A Smart Book

Web Development in Pascal



Primož Gabrijelčič www.smartprogrammer.org

A Smart Book

Web Development in Pascal with Smart Mobile Studio

Primož Gabrijelčič

This book is for sale at http://leanpub.com/asmartbook

This version was published on 2013-09-04



This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

©2012 - 2013 Primož Gabrijelčič

Tweet This Book!

Please help Primož Gabrijelčič by spreading the word about this book on Twitter!

The suggested hashtag for this book is #asmartbook.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search/#asmartbook

Also By Primož Gabrijelčič

Parallel Programming with OmniThreadLibrary

Параллельное программирование с OmniThreadLibrary

Programación Paralela con OmniThreadLibrary

Contents

C	redits					. i
In	troduction					. ii
	Sample Co	de				. iii
	Formatting	Conventions				. iv
	Work in P	ogress				. v
	Advertiser	nent				. vi
Sa	ample Book				• ,	. vii
1	Smart Pas	cal				. 1
-		Types				
	1.1.1	Base Types				
	1,1,1	1.1.1.1 Integer				
		1.1.1.2 Float				
		1.1.1.3 Boolean				
		1.1.1.4 String				
		1.1.1.5 Variant				
		1.1.1.6 Type Mappings				
	1.1.2	Enumerations				
	1.1.3	Sets				
	1.1.4	Arrays				
		1.1.4.1 Static Arrays				
		1.1.4.2 Dynamic Arrays				
	1.1.5	Records				
		1.1.5.1 Anonymous Records [1.0.1.120]				. 8
		1.1.5.2 Default Field Values $\begin{bmatrix} 1 & 0 & 1 & 120 \end{bmatrix}$				
	1.1.6	Classes				. 9
		1.1.6.1 Default Field Values $\begin{bmatrix} 1 & 0 & 1 & 120 \end{bmatrix}$. 10
		1.1.6.2 Partial Classes				
		1.1.6.3 External Classes				
	1.1.7	Interfaces			. ,	. 11
	1.1.8	Helpers				
	1.1.9	Delegates				

CONTENTS

			13
		1.1.10.1 Lambdas [1.1.0.911]	14
		1.1.10.1.1 Lambda Functions	14
		1.1.10.1.2 Lambda Statements	15
	1.2	Statements	17
	1.3	Operators	18
		1.3.1 Operator Overloading $\begin{bmatrix} 1 & 0 & 1 & 1220 \end{bmatrix}$	20
		1.3.2 Helper Methods $\begin{bmatrix} 1 & 0 & 1 & 120 \end{bmatrix}$	21
	1.4	Comments	22
	1.5	Memory Model	23
	1.6	Code Structure	24
		1.6.1 Unit Namespaces [1.1.0.911]	25
	1.7	Property Expressions [1.1.0.911]	26
	1.8	Properties with Anonymous Storage [1.1.0.911]	29
	1.9	Contracts	30
		Directives	31
	1.10	1.10.1 Conditional Compilation	31
		1.10.2 Conditional Compilation Symbols	32
		1.10.3 Code Inclusions	32
		1.10.4 Errors and Messages	33
	1 11	Predefined Constants	34
		Built-in Functions	35
	1,12	1.12.1 Mathematical Functions	35
		1.12.2 String Functions	39
		1.12.3 Date/Time Functions [1.0.1.120]	42
	1 13	Including JavaScript Code	45
	1.13	1.13.1 Name Conflicts and Obfuscation Support	45
		1.13.2 You May Be Better Without It	46
		1.13.3 Implicit Parameters Structure	46
		1.13.4 Variables	46
		1.13.5 Handling Callbacks with Variant Methods	47
		1.13.6 Handling Callbacks in an asm section	48
		C	48
		1.13.7 Limitations	40
2	Prog	gramming with Smart	49
	2.1	Message Dialogs	51
		2.1.1 ShowMessage	51
		2.1.2 ShowDialog	52
	2.2	Modal Dialogs [1.1.0.911]	56
		2.2.1 ShowModal	56
		2.2.2 HideModal	58
	2.3	Running Code on Application Shutdown	59
	-		-

CONTENTS

3	Sma	rt Controls	2
	3.1	Overview	3
	3.2	TW3TagObj	6
	3.3	TW3Component	7
		3.3.1 Public Methods [110911]	7
	3.4	EditBox	9
		3.4.1 Sample Application	0
	3.5	Listbox [1.10.911]	3
		3.5.1 Listbox Styles	
		3.5.2 Sample Application	6
	3.6	Scrollbox [1.1.0.911]	
		3.6.1 Sample Application	
	3.7	Toolbar	
		3.7.1 Sample Application	
			Ĭ
4	Sma	rt RTL	6
	4.1	RTL 8	7
	4.2	RTL\PhoneGap	2
	4.3	RTL\Steema	3
	4.4	RTL\System	4
	4.5	RTL\w3c	5
	4.6	Libraries	6
	4.7	Libraries\Box2D	7
	4.8	Libraries\GLScene	8
	4.9	Libraries\RemObjects	9
	4.10	Libraries\SmartUnit	0
	4.11	Graphics	1
		4.11.1 Colour	1
		4.11.2 Line	1
		4.11.3 Circle	4
		4.11.4 Rectangle	6
		4.11.5 Curve	7
		4.11.6 Transparency	
		4.11.7 Gradient	0
		4.11.8 Image	3
		4.11.9 Text	6
	4.12	Regular Expressions	9
		4.12.1 Example	
		4.12.2 A Look Inside	
		4.12.3 Creating a Regular Expression Object	
		4.12.4 Match	
		4.12.5 Search	
		4.12.6 Replace	

CONTENTS

4.12.7 Split	24
4.12.8 Exec	
4.12.9 Test	
4.12.10RegExDemo Example	26

Credits

The author would like to thank Jon Lennart Aasenden and Eric Grange for making the wonderful program described in this book and for publishing many posts explaining the Smart Mobile Studio and its inner working.

I'd also like to thank Eric Grange and Andre Mussche for providing invaluable feedback that helped me refine the content.

Title page images are taken from the Smart Mobile Studio demos. From left to right: Minesweeper (by Christian-W. Budde), LSystemFiddle (by Eric Grange), MandelbrotExplorer (by Primož Gabrijelèiè), and TeeChart (by Steema Software).

Introduction

This is a book about Smart Mobile Studio¹, a development environment that takes Pascal source and spits out HTML+CSS+JavaScript which can be executed in all most common browsers on desktop and mobile devices.

The material in the book assumes that you have a basic knowledge about the Pascal programming language. Knowledge of object-oriented principles and Object Pascal syntax (Delphi, FreePascal) will also come handy.

If you are new to the Pascal world, a good introduction is *Essential Pascal*² by Marco Cantu.

There's also a good overview of Object Pascal at Delphi wikia3.

¹www.smartmobilestudio.com

²http://www.marcocantu.com/epascal/

³http://delphi.wikia.com/wiki/Object_Pascal

Introduction iii

Sample Code

Sample programs for this book can be downloaded from the *a-smart-book*⁴ Google Code project.

⁴http://code.google.com/p/a-smart-book/

Introduction iv

Formatting Conventions

This book covers all versions of Smart Mobile Studio released to the public.

In some extreme cases, Smart Mobile Studio may be out for few days before the book is updated but as the author actively participates in alpha and beta tests, this is more an exception than the rule.

When a part of the book covers only some versions of Smart Mobile Studio, a $^{[version \, tag]}$ in superscript indicates relevant versions.

A single version ($^{[1.0.1]}$ f.i.) indicates that the topic in question was introduced in this version and that it is still supported in the current release.

A range of two versions ($^{[1.0.1-1.9]}$ f.i.) indicates that the topic was introduced in the first version (1.0.1 in this example) and that it was supported up to the second version (1.9). After that, support for this topic was removed from the Smart or it was changed so much that additional section was added to describe the new functionality.

In some cases, a [unreleased] tag indicates functionality that will be available in the next version of Smart Mobile Studio.

Introduction V

Work in Progress

This book is a work in progress - it is being published as it is written. When you buy this book, you are not only buying it 'as is' - in accordance with the Lean Publishing Manifesto⁵ you will get all subsequent ebook versions for free, forever.

Visit the Smart Programmer⁶ blog to check the status of the book and to influence the writing process by voting on the importance of particular topics.

⁵http://leanpub.com/manifesto

⁶http://www.smartprogrammer.org

Introduction vi

Advertisement

I teach various Delphi and Smart Mobile Studio topics. I'm also available for Delphi or Smart Mobile Studio consulting. Find more at http://www.glagolite.si/training.

Sample Book

This is just the sample of the real book. It includes two full chapters - a very long (and important) chapter on *Smart Pascal*, the Object Pascal dialect used to program Smart Mobile Studio and a chapter on regular expressions - and selections from other chapters.

To check the status of the full book, visit the Smart Programmer⁷ blog.

You can buy the book on the LeanPub⁸.

⁷http://www.smartprogrammer.org

⁸http://leanpub.com/asmartbook

The programming language used in the Smart Mobile Studio is a dialect of the Object Pascal¹. The frontend (parser) comes from the DWScript² project which implements a language based on the Delphi³ programming environment with additions from FreePascal⁴ and Oxygene⁵. This chapter documents the differences between the Smart Pascal and Delphi Pascal.

 $^{{\}rm ^1}http://en.wikipedia.org/wiki/Object_Pascal$

²http://code.google.com/p/dwscript/

³http://embarcadero.com/Delphi

⁴http://www.freepascal.org/

⁵http://www.remobjects.com/oxygene/

1.1 Data Types

Smart Pascal supports most of the data types from Delphi. Some base types are missing because there is not direct JavaScript support for them and some complex type were added.

1.1.1 Base Types

Smart Pascal implements following base types.

1.1.1.1 Integer

Represents integer numbers. *Integer* maps to JavaScript Number object and is therefore internally always represented as a floating-point number. The compiler will strive to keep the Number object with an integer value, though since Number is a double-precision float that value can exceed the 32 bits range and reach up to 54 bits for regular operands (+, -, <, etc.). Due to JavaScript limitations, value will be clamped to 32 bits for bitwise operands.

Integers can be represented in a decimal notation (123, -456) or hexadecimal notation (\$BAD, 0xFACE). [1.1.0.911] Since version 1.1, integers can also be represented in a binary notation (0b10101).

1.1.1.2 Float

Represents double-precision floating-point numbers. *Float* maps to JavaScript Number object.

Floats can be represented in normal notation (3.1415) or in exponential notation (1.23e45).

1.1.1.3 Boolean

Stores True or False. Maps to JavaScript Boolean object.

When casting a Boolean as an Integer, True maps to 1 and False maps to 0. When casting an Integer as a Boolean, 0 maps to False and all other values map to True.

1.1.1.4 String

Stores mutable (garbage collected), UTF-16 strings. String maps to JavaScript String object.

Strings can be delimited by a single quote ('abc') or double quote ("ABC").

Explicit Unicode characters can be specified by using # followed by an integer codepoint (decimal or hexadecimal). Characters specified this way are always understood as Unicode codepoint.

Following statement would output 'Hello' followed by CR+LF (ASCII code 13 and 10), followed by 'World'.

```
writeln('Hello'#13#$0D'World');
```

In a single-quoted string, a single quote can be expressed by doubling it ('can''quote this!'). In a double-quoted string, a double-quote can be expressed by doubling it ("A""QUOTE"). Double-quoted strings can span multiple lines.

This is an equivalent of the example few lines above.

```
writeln("Hello
World");
```

Strings can also be defined with #" or #". In this case the compiler will then ignore common indentation, and will additionally ignore an empty first line. Following example is equivalent to both examples above.

```
writeln(#"
Hello
World");
```

^[1.1.0.911] The String type has built-in Low/High/Length pseudo-methods, similar to those of the array types. You can use str.Low, str.High and str.Length as an alternative to Low(str), High(str) & Length(str).

[1.1.2] In operator can be used to check for substrings (if 'java' in 'javascript' then).

[1.1.2] Helpers can operate directly on string constants ("Smart".Reverse).

1.1.1.5 Variant

Stores any Smart Pascal base or complex type, JavaScript object or array.

Variants holding a JavaScript object cannot be directly casted from/to TObject. One of the reason for this is the obfuscation but even with obfuscation disabled, Smart compiler has to rename inherited methods, which are not supported in the JavaScript. Similar scoping issues happen in variety of other situations (nested procedures f.i.).

^[1.0.1.120] Published record properties can be automatically assigned to a field of a variant object (and **not** directly to the variant object itself). For example, the code below will output {"field":{"B":42,"S":"Smart!"}}

```
1
    type
2
      TRec = record
3
      public
 4
      A: integer;
5
    published
      B: integer;
 6
       S: string;
     end;
8
9
   procedure TApplication.PopulateConsole;
10
12
   r: TRec;
13
     v: Variant;
14
      s: string;
15 begin
16
     r.A := 17; r.B := 42; r.S := 'Smart!';
     v := TVariant.CreateObject;
17
     v.field := r;
18
      asm @s = JSON.stringify(@v); end;
19
     console.Writeln(s);
20
21
   end;
```

1.1.1.6 Type Mappings

Unit w3system defines following type mappings for better compatibility with existing Pascal codebase.

```
TDateTime = Float;
Real = Float;
Double = Float;
Extended = Float;
```

1.1.2 Enumerations

Enumerations are simple integer-based types, which may have explicit or automatically assigned values. Enumerations can be cast to and from Integer.

Explicitly scoped enumerations are supported using the enum⁶ keyword:

⁶http://wiki.oxygenelanguage.com/en/Enum_(keyword)

```
type MyEnumeration = enum (First, Second, Third, Tenth = 10, Eleventh);
```

With this syntax, enumeration elements must be referred with the enumeration type as scope, as in MyEnumeration.First. If values aren't specified explicitly, they start from 0 and increase by 1 from the previous member.

Another variant of enumerations is when using the flags⁷ keyword, in that case, values cannot be explicit, and they start from 1 and are multiplied by 2 from the previous member (1, 2, 4, 8 etc.)

```
type MyFlags = flags (Alpha, Beta, Gamma);
```

With this syntax too the elements can only be accessed through explicit scoping as in MyFlags. Beta. Finally, the classic Object Pascal enumeration syntax is supported:

```
type TMyEnumeration = (firstEnum, secondEnum, thirdEnum);
type TMyExplicitEnum = (eeOne = 1, eeTwo = 3, eeFive = 5);
```

With this classic syntax, an enumeration element can be referred directly (eeOne) or prefixed with the enumeration type (TMyExplicitEnum.eeOne).

1.1.3 **Sets**

[1.1.2] Smart supports a limited version of sets. Only sets of enumerations are supported. Only in operator is supported. Compiler supports Include and Exclude methods, which can be used in two ways, either as Include(mySet, value) or as mySet.Include(value).

1.1.4 Arrays

Arrays come in two varieties – static arrays where the length of the array is known during the compilation and dynamic arrays which can be resized during the program execution.

1.1.4.1 Static Arrays

Static arrays are supported as value types, they have a fixed size with user-specified bounds:

```
type TZeroToTen = array [0..10] of Integer;
type TTenToTwenty = array [10..20] of String;
```

Multi-dimensional arrays are supported, with two forms:

⁷http://wiki.oxygenelanguage.com/en/Flags_(keyword)

```
type TCompactForm = array [1..3, 1..5] of Float;
type TVerboseForm = array [1..3] of array [1..5] of Float;
```

They support the special functions/methods Low, High, Length and Count. You can use them as either functions or methods. Low(array) and array. Low are equivalent. Length and Count are equivalent.

You can use the in and not in operators to test the presence of an element in a static array. In this way, arrays can be a partial replacement for sets, which are currently not supported by the Smart Pascal.

Constant static arrays can be initialized using square brackets []:

```
const a: array [0..2] of String = ['zero', 'one', 'two'];
```

They can also be defined inline for assignments to static or dynamic arrays:

```
var staticArray: array [0..2] of String;
var dynamicArray: array of String;

staticArray := ['zero', 'one', 'two'];
dynamicArray := ['one', 'two'];
```

[1.1.0.911] Since version 1.1, static arrays can be initialized using standard Delphi bracket syntax:

```
const a: array [0..2] of String = ('zero', 'one', 'two');
```

1.1.4.2 Dynamic Arrays

Dynamic arrays are supported as reference types. They are declared without bounds. Their lower bound is always zero.

In addition to Low, High, Length and Count, they also support the following pseudo-methods:

- Add(item [,...]) / Push(item [,...])

 Increases Length by one and adds one or more item at the end of the array, can also add arrays (concatenation).
- Clear()

Empties the array (equivalent to SetLength(0)).

• Copy(startIndex[, count])

Creates a new dynamic array containing count items from startIndex. If count isn't specified, copies to the end of the array.

• Delete(index[, count])

Deletes the item at the specified index and reduces Length by count (default one).

• IndexOf([startIndex,]item)

Returns the index of an item, returns a negative value if not found. Starts the search at offset startItem (default zero).

• Insert(index, item)

Inserts an item at the specified index.

• Peek()

Returns the last item.

• Pop()

Returns the last item and removes it from the array.

• SetLength(newLength)

Defines the length of a dynamic array. If a array is resized, old items are preserved as much as possible (if the array is shortened, extraneous items are removed).

• Swap(index1, index2)

Swaps two items of specified indexes.

• Reverse()

Reverses the order of the items.

The reasons those are called *pseudo*-methods are:

- 1. They have compiler-magic for superpower overloads: .Add() f.i. accepts any number of parameters, and those parameters can be items, static arrays or dynamic arrays, and that's checked at compile-time.
- 2. You can't acquire function pointers from them.

You can use the in and not in operators to test the presence of an element in a dynamic array.

Dynamic arrays can be initialized from inline or constant static arrays:

```
var dynamicArray : array of String;

dynamicArray := ['one', 'two', 'three'];
```

[1.1.2] Dynamic arrays now have a sort methods which takes a comparison function; that maps directly to JavaScript array sort method. String, Integer and Float arrays also implement a sort method without a parameter which sorts the array in the natural order.

[1.1.2] Dynamic arrays now have a map method which maps directly to the JavaScript map method.

1.1.5 Records

Records are supported using the classic Delphi syntax. They're value types. See also Classes Methods and properties of records are supported too, including class methods.

1.1.5.1 Anonymous Records [1.0.1.120]

Anonymous records allow you to declare arbitrary inline JavaScript objects

```
var
1
2
     v: Variant;
3
  v.rec := record
5
     Field = 'hello';
      Value : Integer = someVar+someFuncCall(anotherVar);
6
      'dotted.weird$name' := #"
7
          Lorem ipsum
          dolor sit amet"
9
  end;
```

Field types can be type-inferenced or provided explicitly. Initialization values can be constants of dynamic values. Field names can be specified as strings to provide for naming conventions, valid in JavaScript but not in Object Pascal.

The above would be expressed in JSON as

```
1 {
2    "Field" : "hello",
3    "Value" : someVar+someFuncCall(anotherVar),
4    "dotted.weird$name" : "Lorem ipsum\ndolor sit amet"
5 }
```

1.1.5.2 Default Field Values [1.0.1.120]

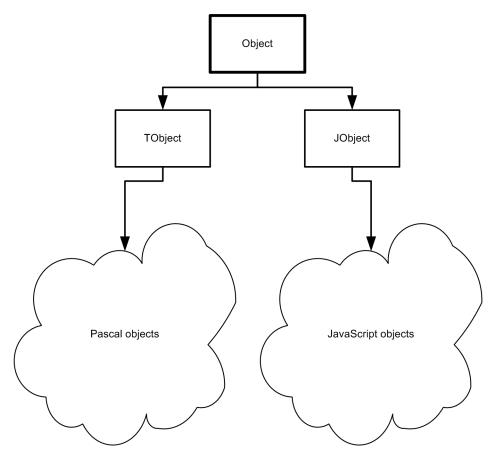
Record fields can have default values. Types for fields with default values can be type-inferenced.

```
1 type
2  TMyRecord = record
3  Field1 := 'hello';
4  Field2 : Integer = 123;
5 end:
```

1.1.6 Classes

Classes all derive from the root TObject class and follow the classic Delphi syntax. They're reference types. Although the JavaScript Objects don't support inheritance and virtual methods, all of object-related paradigms are supported in Smart Pascal.

[1.1.0.911] Version 1.1 introduced new *super-parent* class, Object, which is parent of the TObject class and of a newly introduced JObject class. The Object class is empty – it introduces no fields, methods or properties. The JObject class is a parent class for external classes.



Named constructors are supported, as well as class methods, meta-classes, virtual methods, properties and destructors. Properties can optionally be array properties, and can feature an explicit index.

You can also declare class methods with method as in the Oxygene language⁸, in addition to

 $^{^{8}} http://wiki.oxygenelanguage.com/en/Methods \\$

procedure and function.

Classic constructor syntax is supported, but you can also specify a default constructor and use the new keyword to instantiate classes.

Both lines of code below are equivalent:

```
obj := TMyObject.Create(parameter);
post := new TMyObject(parameter);
```

Visibilities are private, protected, public and published. Private and protected in Smart correspond to strict private and strict protected in Delphi.

Classes can implement Interfaces. (See example in the *Interfaces* section.)

1.1.6.1 Default Field Values [1.0.1.120]

Class fields can have default values. Types for fields with default values can be type-inferenced.

```
type
TMyClass = class
Field1 := 'hello';
Field2 : Integer = 123;
end;
```

1.1.6.2 Partial Classes

Partial classes allow splitting the definition and implementation of a class over multiple files, or over multiple sections of the same file. Partial declarations must have the same modifiers (abstract, sealed, ...), must specify the same ancestor (or none), however each partial declaration is allowed to introduce new interfaces and their implementations.

Partial classes can be declared as class partial (TAncestor) as well as with the Oxygene partial class (TAncestor) syntax9.

1.1.6.3 External Classes

Classes can be marked as external, in which case they're meant to expose classes that are not implemented in the script, and unlike interfaces, then can define fields.

Following example (taken from the w3regex RTL unit) wraps JavaScript class RegExp¹⁰.

 $^{^9} http://wiki.oxygenelanguage.com/en/Partial_Classes$

¹⁰http://www.w3schools.com/jsref/jsref_obj_regexp.asp

```
type
1
      JRegExp = class external 'RegExp'
2
        constructor Create(regularExpression, flags: String); overload;
3
        constructor Create(regularExpression: String); overload;
4
5
        function Exec(s: String): TStrArray; external 'exec';
6
        function Test(s: String): Boolean; external 'test';
8
9
        global : Boolean;
10
        ignoreCase : Boolean;
        lastIndex : Integer;
11
12
        multiline : Boolean;
13
        source : String;
14
      end;
```

[1.0.1.120] An external name can be provided for an external field.

[1.1.0.911] External classes can be subclassed. All external classes are parented from the JObject class.

```
1 FField: Integer; external "Hello";
```

1.1.7 Interfaces

Interfaces are supported using the classic Delphi syntax (except the GUID part which is not supported).

Interfaces can define properties, which are syntax sugar to their methods.

Example from the w3motion RTL unit:

```
1
    type
      IW3MeasurementBuffer = interface
2
        procedure Add(x, y, z: float);
3
        procedure Clear;
4
        function Count: integer;
5
        procedure Get(item: integer; var x, y, z: float);
6
7
      end;
8
      TW3MeasurementBuffer = class(IW3MeasurementBuffer)
9
10
      public
11
        constructor Create(bufferSize: integer);
12
        procedure Add(x, y, z: float);
13
```

```
procedure Clear;
function Count: integer;
procedure Get(item: integer; var x, y, z: float);
end;
```

The class implementing an interface can be derived from any parent. There is no TInterfacedObject class.

1.1.8 Helpers

Helpers in Smart Pascal are a generalization of Delphi's class helpers and can be applied to (almost) any type, even to built-in types. You can also have multiple helpers per type.

The syntax is as following:

You can have helpers for base types, classes, records, interfaces or arrays.

For instance if you write...

```
type
1
2
      TFloatHelper = helper for Float
        const PI = 3.141592;
3
        function ToString : String;
 4
5
      end;
6
    function TFloatHelper.ToString : String;
7
8
       Result := FloatToStr(Self);
9
   end;
10
```

...you can then write expressions Float.Pi or myFloatVar.ToString.

The literal form is also accepted (TFloatHelper.Pi and TFloatHelper.ToString(myFloatVar)).

It is possible to have class methods in helpers; for types with a meta-symbol, Self will be that meta-symbol (f.i. for a TObject helper, Self would be TClass), for others (like Float), Self will not be

defined. For records, Self is passed a a const parameter therefore record helpers cannot modify the record being operated upon.

[1.1.0.911] Smart also supports a simplified helper syntax where you just declare a method and mark it as a helper. You can for example declare a following helper...

```
function Concat(s: string; prefix: string): string; helper Prepend;
begin

Result := prefix + s;
end;

...and then use it on any data of the string type.

var s := 'Hello world';
Console.WriteLn(s);
Console.Writeln(s.Prepend('ABC:'));
Console.Writeln(('my string').Prepend('*** '));
```

[1.1.2] Added a bunch of helpers for String, Integer, Float and Boolean type. See https://code.google.com/p/dwscript/wiki/BaseTypes for a full list.

1.1.9 Delegates

Function pointers are unified, there is no distinction between standalone function and method pointers.

For instance the following code is valid as long as the function or method prototype matches.

```
type TMyFunction = function (i: Integer): String;

var v: TMyFunction;

v := IntToStr;

v := someObject.SomeMethod;

v := someInterface.SomeMethod;
```

You can use the @ operator to explicitly access a function reference to remove ambiguity.

1.1.10 Closures

Closures (also known as *anonymous methods*¹¹) are supported with the same syntax as in Delphi.

¹¹http://en.wikipedia.org/wiki/Anonymous_function

```
1 myObject.MyEvent :=
2 procedure (Sender: TObject);
3 begin
4 ...
5 end;
```

You are allowed to use a named local procedure as a closure, with optional capture of local variables, allowing for neater layout of code.

```
begin
1
2
       procedure MyLocalProc(Sender : TObject);
3
4
       begin
5
          . . .
6
       end;
7
       myObject.MyEvent := MyLocalProc;
8
9
10
    end;
```

The function pointers and closures are unified, you don't have to distinguish between a procedure and a procedure of object, and you don't have to distinguish a reference to procedure either. In other words, if you declare

```
type TNotifyEvent = procedure (Sender: TObject);
```

as long as the parameters match (and result type for a function), the above type will accept standalone functions, object methods, interface methods, and now closures (and even record methods, which are just syntax sugar for standalone function with an implicit parameter).

1.1.10.1 Lambdas [1.1.0.911]

Version 1.1 introduced more compact syntax for closures, similar to the C#'s syntax with an extra lambda keyword thrown at the front. Two different types of lambda closures are supported – functions and statements.

1.1.10.1.1 Lambda Functions Lambda functions are used when the code expects an anonymous function and when the code implementing this function is short. The syntax for lambda functions is:

```
1 lambda (parameter_list) => expression
```

The lambda keyword is followed by an optional parameter list, => and an expression returning a result of the appropriate type.

For example, the following code creates an event that is called every five seconds.

```
var repeater := TW3EventRepeater.Create(
function (Sender: TObject): boolean
begin
Result := MyFunction;
end,
5000);
```

Anonymous method calls some function in the code and returns its result. (False will trigger another event after the timeout (5000 ms) and True will stop the repeater.)

Let's rewrite this code using a lambda function.

```
var repeater :=
TW3EventRepeater.Create(lambda (Sender) => MyFunction, 5000);
```

As you can see, there's no need to declare the parameter (Sender) type and the function result type; Smart will detect them automatically. Even more, as we don't use the Sender parameter, we can drop the parameter list and use an even shorter form.

```
var repeater :=
TW3EventRepeater.Create(lambda => MyFunction, 5000);
```

1.1.10.1.2 Lambda Statements The statement form is used when you want to put more than one statement inside the lambda. The syntax for lambda statement is:

```
1 lambda (parameter_list)
2 statement;
3 [statement;]
4 [...]
5 [statement;]
6 end
```

A lambda statement can also implement a function; in that case you should return the result in a normal Delphi way by assigning to a Result pseudo-variable.

Following example sets up a repeater that calls a method MyProc every three seconds.

```
var repeater :=
TW3EventRepeater.Create(lambda => MyProc; Result := false; end, 5000);
```

You can use variables inside a lambda statement, but only if they are declared inline.

Lambdas are an excellent addition to the language, as they provide for a more compact code. For example, in Smart 1.0 the W3Layout unit used the following code:

```
ResizeChildren(FClientArea.Height, [TAlign.Top, TAlign.Bottom],
1
     function (layout: TLayoutImpl): Variant
2
3
     begin
       Result := layout.GetConfig.GetHeight;
4
5
     procedure (layout: TLayoutImpl; value: integer)
6
     begin
7
       layout.GetConfig.Height(value);
8
9
   In Smart 1.1, this was simplified to:
   ResizeChildren(FClientArea.Height, [TAlign.Top, TAlign.Bottom],
1
     lambda (layout) => layout.GetConfig.GetHeight,
2
     lambda (layout, value) layout.GetConfig.Height(value) end);
3
   Lambdas are also very helpful when defining event handlers.
```

W3btnOK.OnClick := lambda Application.HideModal(mrOK); end;

1.2 Statements

All standard structured Pascal statements are supported with the exception of with, goto and label. Supported statements are:

- break
- case of

Unlike Delphi, case . . of can operate on any data type.

- continue
- exit
- for in do

Support is limited to enumerations and arrays.

[1.1.2] for in can be used on variants and sets.

• for to/downto do

For loops also support step parameter, which must be strictly positive. It is added or subtracted depending on loop direction and is evaluated only once at the beginning (like the other for loop members).

- if then else
- raise
- repeat until
- try except/finally
- while do

1.3 Operators

Following operators are supported.

	Unary operators
-	Negation
not	Logical or bitwise NOT
<i>(a)</i>	Explicit function pointer reference

U	or bitwise NO I
@ Explic	it function pointer reference
	Expression operators
+	Addition & string concatenation
-	Subtraction
*	Multiplication
div	Euclidean division
mod	Remainder of Euclidean division
sar	Bitwise shift arithmetic right
shr	Bitwise shift right
shl	Bitwise shift left
in	Present in an array, set
not in	Absence in an array, set
and	Boolean or bitwise AND
or	Boolean or bitwise OR
xor	Boolean or bitwise XOR
implies	Logical IMPLIES
is	Class, interface type test
as	Class, interfaces safe cast
implements	Tests if a class implements an interface
=	Equality test
<>	Difference test
<	Lesser test
<=	Lesser or equal test
>	Greater test
>=	Greater or equal test
if-then-else	[1.1.0.911] Conditional operator

 $^{[1.1.0.911]}$ Conditional operator (also called *ternary operator*, *inline if* and *ternary if* 12) returns *then* expression if conditional (if) expression is True and *else* expression otherwise.

An example below returns a name of the component if the component reference represents a valid object and empty string if the component reference is nil.

¹²http://en.wikipedia.org/wiki/%3F:

```
function ComponentName(comp: TW3Component): string;
begin
Result := if assigned(comp) then comp.Name else '';
end;
```

The important fact is that only the appropriate then/else expression is evaluated – if comp is nil, then comp. Name is never evaluated.

Assignment operators

- := Assignment
- += Addition & concatenation compound operator
- -= Subtraction compound operator
- *= Multiplication compound operator
- /= Division compound operator

These operators don't have a language-defined meaning, but can be overloaded.

Unused operators

```
<<
>>
^
```

1.3.1 Operator Overloading [1.0.1.120]

Smart Pascal supports operator overloading. It can be applied to every type, classes included. You can overload all binary operators (+, *, etc.). The declaration syntax is as follows:

```
operator <op> ( <typeLeft>, <typeRight> ) : <typeResult> uses <someFunc>;
```

Operator overloading is considered a form of syntax-sugar, and treated as such, ie. you have to provide a regular function to handle the processing, the operator is just an alias for that function.

There are three extra operators – <<, >> and ^, which can additionally be used at statement level (unlike regular binary operators). By default those three operators do nothing, they're available for operator-overloading purposes only.

Smart Pascal also supports class operator overloading. The syntax differs from Delphi, FreePascal & Prism in that they are not special methods/procedures, but syntax-sugar aliases, in a fashion similar to what properties achieve.

```
class operator <operator> <rightType> uses <method>
```

To illustrate this with TList, you could declare.

```
class operator += TObject uses Add;
class operator -= TObject uses Remove;
```

After that, myList += item would be equivalent to myList.Add(item).

Class operators support type overloading. It is thus possible to declare

```
1  TMyClass = class
2    procedure AddInteger(i: Integer);
3    procedure AddString(s: String);
4    class operator += Integer uses AddInteger;
6    class operator += String uses AddString;
7  end;
```

The method will be selected depending on the operand type

```
1 myObject += 10; // will use AddInteger
2 myObject += 'ten'; // will use AddString;
```

1.3.2 Helper Methods [1.0.1.120]

Helper methods are accepted for operator overloading.

```
type
1
     Vec2 = array [0..1] of Float;
2
3
   type
      TVec2Helper = helper for Vec2
4
        function Add(const v : Vec2) : Vec2; overload;
5
        begin
6
7
          Result:=[Self[0]+v[0], Self[1]+v[1]];
        end;
8
        function Add(const f : Float) : Vec2; overload;
9
10
        begin
          Result:=[Self[0]+f, Self[1]+f];
11
12
        end;
13
      end;
14
    operator + (Vec2, Vec2) : Vec2 uses TVec2Helper.Add;
15
16
    operator + (Vec2, Float) : Vec2 uses TVec2Helper.Add;
17
   var v1 : Vec2 := [1, 2];
18
   var v2 : Vec2 := [10, 20];
19
20
21
   var v := v1 + v2;
22
23 PrintLn(v[0]);
24 PrintLn(v[1]);
```

1.4 Comments

Pascal and C/C++ style comments are supported:

- // denotes a single line comment
- (* and *) mark a multi-line comment
- { and } mark a multi-line comment
- /* and */ mark a multi-line comment

Comments cannot be nested.

1.5 Memory Model

Smart Pascal is compiled into JavaScript which executes in a garbage-collected execution environment. Therefore, objects in Smart Pascal cannot be leaked and you can leave the destruction to the JavaScript engine.

Invoking destructor explicitly (i.e. calling Free or assigning nil to an interface) is still valid. This approach can be used to release memory for large objects immediately.

An attempt to access a destroyed object will fail with an exception.

1.6 Code Structure

DWScript supports two source structure styles – mixed-mode and classic unit/program.

In *mixed-mode* style, interface and implementation are not used. Source code structure still follows Object Pascal rules with the following specifics:

- Variables, constants and procedures can be declared inline.
- Begin/end is optional for main program code.
- Var, const and type keywords have to be explicit before any var or type declaration in the main program (var/const blocks are allowed in procedure, before the first begin).
- Methods can be implemented inline (in the class declaration).
- There are also no *type sections*, each type declaration must have its own 'type' keyword, and you can even have inline code there.

This coding style is useful for simple units/classes.

```
1
    unit Whatever;
 2
3
    type
       TMyClass = class
 4
           procedure Hello;
5
          begin
 6
              writeln('hello');
 7
8
           procedure World;
9
10
       end;
11
    procedure TMyClass.World;
12
    begin
13
       writeln('world');
14
15
    end;
```

In the *classic* style, units are implemented with interface and implementation sections, just like in classic Object Pascal.

In this style, methods can still be implemented inline (in the class declaration), but only for classes, declared in the implementation section of the *classic* style unit.

In both modes, variables are guaranteed to always be initialized.

Variable declaration can feature an assignment; variable type can be inferred, for instance

```
var i := 20;
is equivalent to

var i: Integer;
i := 20;
as well as

var i: Integer := 20;

[1.1.0.911] Variables can also be declared inline in for statements.

for var i := 1 to 10 do
```

1.6.1 Unit Namespaces [1.1.0.911]

Unit namespace feature in Smart Pascal accomplishes several functions:

- Supporting "classic" namespace as found in Java or .Net.
- Supporting aggregate namespaces and conditional units.
- Supporting unit deprecation and splitting with backward-compatibility.

Unit namespaces are small files containing just a unit namespace and a uses directives. Unit namespaces don't expose or implement anything.

```
unit namespace Foo.Bar; deprecated 'deprecation message';

uses Foo.Bar.One, Foo.Bar.Two, WhateverUnit;
```

The deprecated part is optional.

With the above unit namespace, if in regular code you have uses Foo.Bar, OtherUnit; it will be equivalent to uses Foo.Bar.One, Foo.Bar.Two, WhateverUnit, OtherUnit;.

 $^{^{13}} http://delphitools.info/2012/11/19/new-for-var-syntax-in-dwscript/$

1.7 Property Expressions [1.1.0.911]

Smart Pascal supports properties just the same as Delphi and Free Pascal do. Properties are supported in classes, records, interfaces and in helpers (helper classes and records). In addition to the standard syntax, Smart Pascal supports *property expressions* which help you write more compact code.

Property expressions can be used in two ways – by providing a *backing storage* (typically a field or property containing the data) or by providing a *modifying expression* (an expression that modifies and returns/stores data). In both cases, property expressions are introduced by an opening parenthesis (() and ended by a closing parenthesis ()).

The simplest way to explain the functioning of this language extension is through examples. The following example defines a form containing two labels, W31b1Task and W31b1Detail. The form also defines two properties. Property Task provides access to the W31b1Task.Caption and property Detail provides access to the W31b1Detail.Caption.

```
1
    type
      TToDoListTemplate=class(TW3form)
2
      private
3
        {$I 'ToDoListTemplate:intf'}
 4
5
      protected
        procedure InitializeObject; override;
6
      public
7
8
        property Task: string read (W31b1Task.Caption)
          write (W3lblTask.Caption);
9
        property Detail: string read (W31b1Detail.Caption)
10
          write (W3lblDetail.Caption);
11
12
      end;
```

Equivalent code in a "classic" style would be noticeably longer.

```
type
1
      TToDoListTemplate=class(TW3form)
2
3
      private
        {$I 'ToDoListTemplate:intf'}
4
        function GetDetail: string;
5
        function GetTask: string;
6
        procedure SetDetail(value: string);
7
8
        procedure SetTask(value: string);
9
      protected
        procedure InitializeObject; override;
10
11
12
        property Task: string read GetTask write SetTask;
```

```
property Detail: string read GetDetail write SetDetail;
13
14
      end;
15
16
     function TToDoListTemplate.GetDetail: string;
17
       Result := W31b1Detail.Caption;
18
19
     end;
20
21
     function TToDoListTemplate.GetTask: string;
22
       Result := W31b1Task.Caption;
23
24
     end;
25
     procedure TToDoListTemplate.SetDetail(value: string);
26
     begin
27
28
       W31b1Detail.Caption := value;
29
     end;
30
     procedure TToDoListTemplate.SetTask(value: string);
31
32
       W31blTask.Caption := value;
33
     end;
```

In the example above, (W31b1Task.Caption) and (W31b1Detail.Caption) provide a backing storage; a place where the data is stored. We can, however, do more and write a full expression inside the property expression-wrapping brackets. In case of a *getter* (read access) this expression must return a result and in case of the *setter* (write access) this expression should be a statement setting the value of the property. The *setter* can use a pseudo-parameter Value holding the value that was assigned to the property in the code.



I've said *should be a statement* because the compiler is not really fussy and will handle any expression in the setter. This allows for weird tricks where the assignment to a property produces a side effect.

The second example defines a TAngle record which holds one value (angle) stored in radians¹⁴ and exposes it through two properties, Radians returning the value in radians and Degrees returning the value in degrees. Conversion between radians and degrees is done inside property expressions.

¹⁴http://en.wikipedia.org/wiki/Radian

```
type
TAngle = record
FAngleRad: float;
property Radians: float read FAngleRad write FAngleRad;
property Degrees: float read (Radians/Pi*180)
write (Radians := Value/180*Pi);
end;
```

Equivalent code in "classic" style would again be longer.

```
1
    type
2
      TAngle = record
        FAngleRad: float;
3
        function GetDegrees: float;
4
        procedure SetDegrees(Value: float);
5
        property Radians: float read FAngleRad write FAngleRad;
6
        property Degrees: float read GetDegrees write SetDegrees;
7
8
      end;
9
    function TAngle.GetDegrees: float;
10
    begin
11
12
      Result := Radians/Pi*180;
   end;
13
14
    procedure TAngle.SetDegrees(Value: float);
16
    begin
      Radians := Value/180*Pi;
17
18 end;
```

1.8 Properties with Anonymous Storage [1.1.0.911]

When a property is backed directly by a field (without a getter and/or a setter) and this field is never accessed directly, the code can be further simplified by dropping the field declaration altogether.

An example of such code would be:

```
1 type
2   TMyClass = class
3   private
4    FData: integer;
5   public
6    property Data: integer read FData write FData;
7   end;
```

This code can be simplified to:

```
type
TMyClass = class
public
property Data: integer;
end;
```

Smart will still create a field inside the TMyClass object, but this field will be anonymous and not accessible directly from the code.



This type of properties cannot be used in external classes and classes inheriting from the JObject class.

Using this feature the example from the previous section can be simplified even further.

```
type
TAngle = record
property Radians: float;
property Degrees: float read (Radians/Pi*180)
write (Radians := Value/180*Pi);
end;
```

1.9 Contracts

Contracts programming 15 is supported with a syntax similar to the Oxygene language, procedures can have require 16 and ensure 17 sections. The ensure sections also support the old 18 keyword.

Method and procedure contracts are supported, but contracts in the class definition aren't supported yet.

¹⁵http://wiki.oxygenelanguage.com/en/Class_Contracts

¹⁶http://wiki.oxygenelanguage.com/en/Require_(keyword)

¹⁷http://wiki.oxygenelanguage.com/en/Ensure_(keyword)

 $^{^{18}} http://wiki.oxygenelanguage.com/en/Old_(keyword)$

1.10 Directives

Smart Pascal supports *compiler directives* which can be included in the source code to control compiler behaviour. Three classes of compiler directives are supported: conditional compilation directives, code inclusion directives and error/message control directives.

1.10.1 Conditional Compilation

• {\$DEFINE NAME_TOKEN}

Defines NAME_TOKEN symbol.

• {\$UNDEF NAME_TOKEN}

Undefines NAME_TOKEN symbol.

• {\$IFDEF NAME_TOKEN}

Compiles following block of code (ending with {\$ELSE} or {\$ENDIF}) if NAME_TOKEN symbol is defined.

• {\$IFNDEF NAME_TOKEN}

Compiles following block of code (ending with {\$ELSE} or {\$ENDIF}) if NAME_TOKEN is not defined.

• {\$IF expression}

Evaluates a Boolean expression and compiles the following block of code (ending with {\$ELSE} or {\$ENDIF} if the expression evaluates to True. That expression can use any constant defined previously, as well as any stateless built-in function. It can also invoke the following special functions:

- Defined(nameToken: String)

Returns True if the token is defined when the compilation reaches the expression.

[1.0] This special functions can also be used in regular code. In this case, it will be evaluated at runtime.

 $^{[1.0.1.120]}$ The meaning of this function when used in the regular code has been changed. See the *Built-in Functions* section below. Function ConditionalDefined() works the same as {\$IF Defined()}.

Declared(symbolName: String)
 Returns True if there is a symbol name in scope when compilation reaches the expression.
 This special functions can also be used in regular code. In this case, it will be evaluated at runtime.

• {\$ELSE}

Toggles the current *compilation state*. If the code was not compiled before, it will be compiled after this directive and vice versa. This directive is valid only if placed after {\$IFDEF}, {\$IFNDEF}, or {\$IF} directive and before corresponding {\$ENDIF} directive.

• {\$ENDIF}

Closes the conditional compilation block.

Conditional compilation directives may be nested.

1.10.2 Conditional Compilation Symbols

[1.1.2] Conditional compilation symbol DWSCRIPT is always defined.

1.10.3 Code Inclusions

- {\$INCLUDE 'filename'} or {\$I 'filename'} Includes the specified file in the source.
- {\$INCLUDE_ONCE 'filename'}

Includes the specified file in the source only if it hasn't already been included (directly or through inclusion).

• {\$R 'filename'} [1.0.1.120]

Include a resource file. It is included in the index.html file in place of the %rescode%> marker.

The \$I (or \$INCLUDE, they are equivalent) can be used in a special form as a macro replacement directive.

• {\$I %FUNCTION%}

Includes a string literal containing the name of the function or method where the directive is.

• {\$I %FILE%}

Includes a string literal containing the current file name where the directive is.

• {\$I %LINE%}

Includes a string literal containing the line number where the directive is.

• {\$I %DATE%}

Includes a string literal containing the date of compilation (yyyy-mm-dd).

• {\$I %TIME%}

Includes a string literal containing the time of compilation (hh:nn:ss).

1.10.4 Errors and Messages

• {\$WARNINGS ON|OFF}

Enable or disable compiler warnings.

• {\$HINTS ON|OFF|NORMAL|STRICT}

Controls compiler hints.

- ON: Enables hints.
- OFF: Disables hints.
- NORMAL: Enables hints at the normal level.
- STRICT: Enables hints at the strict level.
- {\$HINT 'message'}

Emits message as a hint.

• {\$WARNING 'message'}

Emits message as a warning.

• {\$ERROR 'message'}

Emits message as an error (prevents execution).

• {\$FATAL 'message'}

Emits message as a fatal error (aborts compilation).

1.11 Predefined Constants

 $^{[1.1.2]}$ CompilerVersion constant is always defined. It holds a floating point value that holds the compiler version in form YYYYMMDD.XX, with XX being the minor version. This will be updated each time the language changes.

1.12 Built-in Functions

Special built-in functions are built into the compiler. In most cases, they work the same as the equivalent Delphi function.

- Abs
- Assert
- Assigned
- ullet ConditionalDefined $^{[1.0.1.120]}$

Checks if symbol is defined at runtime. Same as {\$IF Defined()}, but executed at runtime.

• Dec

Decrements a number. In Smart Pascal, this is a function returning a new (decremented) value.

- Declared
- Defined [1.0-1.0]

Checks if a symbol is defined or if an expression is defined.

• Defined [1.0.1.120]

Checks whether any value (object, interface, variant that refers to a JavaScript object etc) is defined.

- High
- Inc

Increments a number. In Smart Pascal, this is a function returning a new (incremented) value.

- Length
- Low
- Ord
- Pred
- SizeOf
- Sqr
- Succ
- Swap [1.0.1.122]

Accepts two parameters of any type and swaps their content. Both parameters must be of the same type and they must both be writeable variables, fields or array items.

1.12.1 Mathematical Functions

Following built-in functions work with numbers.

• Odd(i: Integer): Boolean Returns True if i is odd.

Cos(a: Float): FloatReturns the cosine of an angle in radians.Sin(a: Float): Float

Returns the sine of an angle in radians.

• Tan(a: Float): Float

Returns the tangent of an angle in radians.

• Cotan(a: Float): Float

Returns the cotangent of an angle in radians.

• Cosh(a: Float): Float

Returns the hyperbolic cosine of a.

• Sinh(a: Float): Float

Returns the hyperbolic sine of a.

• Tanh(a: Float): Float

Returns the hyperbolic tangent of a.

• ArcSin(v: Float): Float

Returns the inverse sine in radians.

• ArcCos(v: Float): Float

Returns the inverse cosine in radians.

• ArcTan(v: Float): Float

Returns the inverse tangent in radians.

• ArcTan2(y, x: Float): Float

Returns the inverse tangent of the quotient of its arguments.

• ArcSinh(v: Float): Float

Returns the inverse hyperbolic sine in radians.

• ArcCosh(v: Float): Float

Returns the inverse hyperbolic cosine in radians.

• ArcTanh(v: Float): Float

Returns the inverse hyperbolic tangent in radians.

• Hypot(x, y: Float): Float

Returns the hypotenuse of a right-angle triangle (e.g. Sqrt(x*x + y*y)).

• Factorial(v: Integer): Float

Returns the product of all positive integers less than or equal to v.

• Exp(v: Float): Float

Returns e^v.

```
• Ln(v: Float): Float
  Returns natural logarithm of v.
• Log2(v: Float): Float
  Returns base-2 logarithm of v.
• Log10(v: Float): Float
  Returns base-10 logarithm of v.
• LogN(n, x: Float): Float
  Returns base-n logarithm of v.
• Power(base, exponent: Float): Float
  Returns base raised to the exponent exponent.
• IntPower(base: FLoat, exponent: Integer): Float
  Returns base raised to the exponent exponent.
• Sqrt(v: Float): Float
  Returns square root of v.
• Int(v: Float): Float
  Returns integer part of the value as a floating-point number.
• 'Floor(v: Float): Float'
  Returns fractional part of the value.
• Ceil(v: Float): Integer
  Returns the smallest integer greater than or equal to a number.
• Floor(v: Float): Integer
  Returns the largest integer less than or equal to a number.
• Trunc(v: Float): Integer
  Returns integer part of the number.
• Round(v: Float): Integer
  Rounds the number to the nearest integer.
• DegToRad(a: Float): Float
  Converts degrees to radians.
• RadToDeg(a: Float): Float
  Converts radians to degrees.
• Sign(v: Float): Integer
  Returns the sign of the number (-1, 0, +1).
```

• Max(v1, v2: Float): Float

Returns larger of two numbers.

• Max(v1, v2: Integer): Integer or MaxInt(v1, v2: Integer): Integer Returns larger of two numbers.

• Min(v1, v2: Float): Float

Returns smaller of two numbers.

• Min(v1, v2: Integer): Integer or MinInt(v1, v2: Integer): Integer Returns smaller of two numbers.

• Clamp(v, min, max: Float): Float

Limits value to an interval.

 $\bullet \ {\tt ClampInt}({\tt v}, \ {\tt min}, \ {\tt max} \colon \ {\tt Integer}) \colon \ {\tt Integer}$

Limits value to an interval.

• Pi: Float

Returns the floating-point approximation of the pi constant.

• Gcd(a, b: Integer): Integer

Returns the greatest common denominator of two values.

• Lcm(a, b: Integer): Integer

Returns the least common multiplier of two values.

• IsPrime(n: Integer): Boolean

Returns True if n is a prime number.

• LeastFactor(n: Integer): Integer

Returns the smallest prime that divides the parameter.

• Random: Float

Returns random number from interval [0, 1).

• RandomInt(range: Integer)

Returns random number from interface [0, range).

• RandG(mean, stdDev: Float): Float

Generates random numbers with normal distribution using Marsaglia-Bray algorithm.

• Randomize

Sets the random generator seed to a random value.

• RandSeed: Integer

Returns the random generator seed.

• SetRandSeed(seed: Integer)

Sets the random generator seed to a given value.

1.12.2 String Functions

Following built-in functions work with strings.

• Chr(i: Integer): String

Returns the string corresponding to the given Unicode codepoint (supports the whole Unicode range).

• IntToStr(i: Integer): String

Returns a string representing i in decimal notation.

• StrToInt(str: String): Integer

Converts the decimal representation of a number from a string to a number.

• StrToInt(str: String; def: Integer): Integer

Converts the decimal representation of a number from a string to a number; returns def if value cannot be converted.

• VarToIntDef(val: Variant; def: Integer): Integer

Converts variant value to a number; returns def if value cannot be converted.

• IntToHex(v, digits: Integer): String

Returns a string representing i in a hexadecimal notation.

• HexToInt(hexa: String): Integer

Converts the hexadecimal representation of a number from a string to a number.

• IntToBin(v, digits: Integer): String

Returns a string representing i in a binary notation.

• BoolToStr(b: Boolean): String

Converts a boolean value to a string. Returns True or False.

• StrToBool(str: String): Boolean

Converts a string value to a boolean. Following values will return True: 'True', 'T', 'Yes',

• FloatToStr(f: Float; p: Integer = 99): String

Returns a string representation of a value f with optional precision p.

• StrToFloat(str: String): Float

Converts the floating point representation of a number from a string to a number.

• StrToFloatDef(str: String; def: Float): Float

Converts the floating point representation of a number from a string to a number; returns def if value cannot be converted.

• VarToFloatDef(val: Variant; def: Integer): Integer

Converts variant value to a floating-point number; returns def if value cannot be converted.

• Format(fmt: String; args: array of const)

Formats variable number of parameters into a string according to the format specifier.

• CharAt(s: String; c: Integer): String

Returns character at a specified position.

• Delete(var s: String; index, len: Integer)

Deletes part of the string.

• Insert(src: String; var s: String; index: integer)

Inserts substring into a string at the specified position.

• LowerCase(str: String): String

Converts string to a lower case.

• AnsiLowerCase(str: String): String

Converts string to a lower case according to the current locale.

• UpperCase(str: String): String

Converts string to an upper case.

• AnsiUpperCase(str: String): String

Converts string to an upper case according to the current locale.

• Pos(subStr, str: String): Integer

Returns location of a substring inside a string or 0 if substring was not found.

• PosEx(subStr, str: String; offset: Integer): Integer

Returns location of a substring inside a string, starting the search at the specified offset.

• RevPos(subStr, str: String): Integer

Returns location of a rightmost occurrence of a substring inside a string.

• SetLength(var s: String; newLength: integer)

Sets length of a string.

• TrimLeft(str: String): String

Removes all whitespace from the beginning of a string.

• TrimRight(str: String): String)

Removes all whitespace from the end of a string.

• Trim(str: String): String

Removes all whitespace from both the beginning and the end of a string.

• SameText(str1, str2: String): Boolean

Checks whether two strings have the same content.

• CompareText(str1, str2: String): Integer

Compares two strings without case sensitivity and returns -1 if first is lexically smaller than the second, 0 if they are equal and 1 otherwise.

- AnsiCompareText(str1, str2: String): Integer
 - Compares two strings without case sensitivity according to the current locale and returns -1 if first is lexically smaller than the second, 0 if they are equal and 1 otherwise.
- CompareStr(str1, str2: String): Integer
 - Compares two strings with case sensitivity and returns -1 if first is lexically smaller than the second, 0 if they are equal and 1 otherwise.
- AnsiCompareStr(str1, str2: String): Integer
 - Compares two strings with case sensitivity according to the current locale and returns -1 if first is lexically smaller than the second, 0 if they are equal and 1 otherwise.
- IsDelimiter(delims, str: String; index: integer): Boolean
 - Returns True if str[index] matches one of the delims delimiters.
- LastDelimiter(delims, str: String): Integer
 - Returns position of the rightmost delimiter in the string.
- FindDelimiter(delims, str: String): Integer
 - Returns position of the leftmost delimiter in the string.
- QuotedStr(str: String; quoteChar: String = "'"): String
 - Wraps a string into quote characters and duplicates all occurrences of the same inside the string.
- Copy(str: String; index, len: Integer): String
 - Returns len characters from string str starting at position index.
- LeftStr(str: String; count: Integer): String
 - Returns first count characters from the string.
- RightStr(str: String; count: Integer): String
 - Returns last count characters from the string.
- 'MidStr(str: String; start, count: Integer): String'
- Returns count characters from the middle of the string.
- SubStr(str: String; start: Integer): String
 - Returns string without the first start-1 characters.
- SubString(str: String; start, end: Integer): String
 - Returns characters from index start to index end.
- StringOfChar(str: String; count: Integer): String
 - Returns a string with a specified number of repeating characters.
- StringOfString(str: String; count: Integer): String
 - Returns a string with a specified number of repeating strings.
- StrBeginsWith(str, beginStr: String): Boolean
 - Returns True if a string begins with a substring.

```
• StrEndsWith(str, endStr: String): Boolean Returns True if a string ends with a substring.
```

• StrAfter(str, delimiter: String): String

Returns part of the string after the delimiter.

• StrBefore(str, delimiter: String): String Returns part of the string before the delimiter.

 $\bullet \ {\tt StrSplit}({\tt str}, \ {\tt delimiter} \colon {\tt String}) \colon {\tt array} \ {\tt of} \ {\tt string}$

Returns the string splitted on the delimiters as an array of string.

 \bullet StrJoin(strs: array of string; delimiter: String): String

Joins the strings in the array, concatenating them with delimiter in between each item and returns the resulting string.

• ReverseString(str: String): String

Returns a string that is a reversed version of str.

1.12.3 Date/Time Functions [1.0.1.120]

These functions use a Float type to represent a date/time, similar to TDateTime (1 unit per day, same time origin as in Delphi). The w3system unit defines a type alias type TDateTime = Float.

• Now(): Float

Returns the current date time.

• Date(): Float

Returns the current date.

• Time(): Float

Returns the current time.

• UTCDateTime(): Float

Returns the current UTC date time.

• DateTimeToStr(dt: Float): String

Converts a date/time to a string representation.

• StrToDateTime(str: String): Float

Parses a string representing a date/time.

• StrToDateTimeDef(str: String; def: Float): Float

Parses a string representing a date/time, returns def if the string isn't a valid date/time.

• DateToStr(dt: Float): String

Converts a date to a string representation.

• StrToDate(str: String): Float
Parses a string representing a date.

• StrToDateDef(str: String; def: Float): Float

Parses a string representing a date, returns def if the string isn't a valid date.

• DateToISO8601(dt: Float): String

Converts a date to its ISO8601 string representation.

• DateTimeToISO8601(dt: Float): String

Converts a date/time to its ISO8601 string representation.

• TimeToStr(dt: Float): String

Converts a time to a string representation.

• StrToTime(str: String): Float

Parses a string representing a time.

• StrToTimeDef(str: String; def: Float): Float

Parses a string representing a time, returns def if the string isn't a valid time.

• DayOfWeek(dt: Float): Integer

Returns the day of the week between 1 and 7 with Sunday as the first day.

• DayOfTheWeek(dt: Float): Integer

Returns the day of the week between 1 and 7 with Monday as the first day.

• FormatDateTime(frm: String; dt: Float): String

Converts date/time into a string according to the given format specifier.

• IsLeapYear(year: Integer): Boolean

Returns True if a year is a leap year.

• IncMonth(dt: Float; nb: Integer): Float

Returns a date/time offset by nb months (nb can be negative or positive).

• DecodeDate(dt: Float; var y, m, d: Integer)

Splits date/time into year, month and day parts.

• EncodeDate(y, m, d: Integer): Integer

Converts year, month and day into a date.

• DecodeTime(dt: Float; var h, m, s, ms: Integer)

Splits date/time into hour, minute, second and millisecond parts.

• EncodeTime(h, m, s, ms: Integer): Float

Converts hour, minute, second and millisecond into a time.

• FirstDayOfYear(dt: Float): Float

Returns date representing the first day of the year with the dt date.

• FirstDayOfNextYear(dt: Float): Float

Returns date representing the first day of the year following the year with the dt date.

• FirstDayOfMonth(dt: Float): Float

Returns date representing the first day of the month with the dt date.

• FirstDayOfNextMonth(dt: Float): Float

Returns date representing the first day of the month following the month with the dt date.

• FirstDayOfWeek(dt: Float): Float

Returns date representing the first day of the with with the dt date.

• DayOfYear(dt: Float): Integer

Returns the index of the given date inside the year. January 1st has index 1.

• MonthOfYear(dt: Float): Integer

Returns the integer between 1 and 12 representing the month of the given date.

• DayOfMonth(dt: Float): Integer

Returns the integer between 1 and 31 representing the day of the month for the given date.

- DateToWeekNumber(dt: Float): Integer
- WeekNumber(dt: Float): Integer
- DateToYearOfWeek(dt: Float): Integer
- YearOfWeek(dt: Float): Integer

45

1.13 Including JavaScript Code

Smart Pascal allows you to directly include JavaScript code in the source. Delphi's asm statement was repurposed for this task.

```
function w3_getIsIPad:Boolean;
1
2
   var
     mTemp: Variant;
3
4
   begin
     asm
5
       @mTemp = navigator.userAgent.match(/iPad/i);
6
       if (@mTemp) @Result = true;
7
8
9
   end;
```

There are few good practices and limitations that you have to keep in mind while coding asm sections.

1.13.1 Name Conflicts and Obfuscation Support

The first thing to have in mind is that JavaScript reserved words may be used in Smart Pascal as identifiers. Those can be language keywords (this, delete, etc.) or common DOM objects and properties (document, window). The compiler automatically protects you from such conflicts by transparently renaming your identifiers (currently by adding a \$+number at the end).

Then there is the obfuscator, which will basically rename everything to short, meaningless names. That's good for more than obfuscation: it reduces the size of the JavaScript, improves the parsing and lookup-based performance in the browser.

The consequence is that in an asm section, you should prefix all Pascal identifiers with an @, so the compiler can correctly compile your asm. For instance in:

```
var
window: String;

asm
end;

@window = window.name
end;
```

The @window refers to the window string variable (which the compiler will rename), while window name will be compiled "as is", as it reads the name property of the global window JavaScript object.

1.13.2 You May Be Better Without It

There are many cases in which you don't need asm, as the language supports a Variant type which is a raw JavaScript object, and upon which you can call methods, read properties directly or via indexes.

For instance, with v a Variant, the following code:

```
v := v.getNext();
v['hello'] := v.space + 'world';
will get compiled (almost) straight into
v = v.getNext();
v['hello'] = v.space + 'world';
```

When using Variant, you don't have strong compile-time checks (it's just you versus JavaScript). Property and function names are case-sensitive, so use them with care. This is similar in syntax and essence to using OLE Variants and Delphi.

On the other hand, you have compiler support, and you get automatic casts when assigning a variant to a strong type (Integer, String, etc.), and you also get name conflict protection & obfuscation support without having to @ your Pascal references.

1.13.3 Implicit Parameters Structure

Don't rely on implicit parameters structure because it may change in future compiler revisions!

For instance, methods are currently invoked with an implicit Self parameters, and the others behind, so currently arguments[0] is Self, and everything else if after that. But don't rely on it.

Future compiler revisions may change that parameter's name, may obfuscate it, may remove it entirely in favour of an implicit this, may inline your function, etc.

So if you need explicit parameters, declare them. If you're in a method and need to access the object (Self), use @Self. If you need to access a field of the current object use @Self.FieldName, etc. That will keep working.

1.13.4 Variables

Avoid declaring variables in asm sections. Declare them in the parent function/method instead, and reference them with the @ prefix.

There are three main reasons for that. The first is that doing so means they'll be case-insensitive, the second is that it will allow the obfuscator to obfuscate them, and the third is that you'll get compiler warnings if you declare a variable but do not use it (or if you forgot to @-prefix it).

So don't write that:

```
1
   asm
2
      var myTemp;
3
      myTemp = ...whatever...;
4
   end:
   But write this instead:
   var myTemp: Variant;
1
2
3
   asm
      @myTemp = ...whatever...;
5
   end;
```

1.13.5 Handling Callbacks with Variant Methods

A common occurrence is to register a callback to a JavaScript object, when that object is hosted in a Variant, that's fairly simple to achieve:

```
procedure DoImageLoaded;
begin
...
end;

var
myImage: Variant; // will refer to an image object

myImage.onload(@DoImageLoaded);
```

There we use the @ operator Pascal-side, to make it explicit that we want a function pointer, and not call the function. The @ isn't necessary when the function is declared Pascal-side, as the compiler can figure it out, but when invoking a Variant method, it doesn't know the parameters type.

Note that since function pointers are unified, you can get a function pointer from an object method or an interface method in the same fashion:

```
myImage.onload(@myObject.DoImageLoaded);
myImage.onload(@myInterface.DoImageLoaded);
```

1.13.6 Handling Callbacks in an asm section

If you want to register the callback in an asm section, the situation is a little more complex, as @myObject.myMethod will refer to the function prototype, outside of its context. It means it's okay for standalone functions or procedures, but may not do what you're expecting for object or interface methods.

The solution is to acquire the function pointer outside of the asm section:

```
1  var
2  myCallback: Variant;
3
4  myCallback := @myObject.DoImageLoaded;
5  asm
6  @myImage.onload(@myCallback);
7  end;
```

1.13.7 Limitations

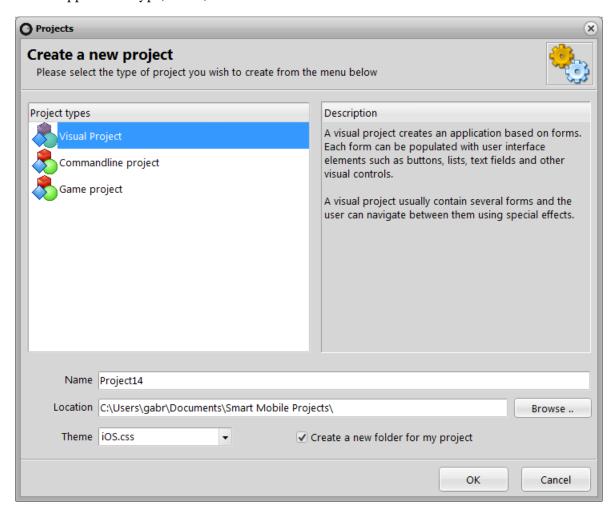
The parser for asm sections doesn't really understand JavaScript – it's still treating JavaScript as a weird invalid form of Pascal meaning that {} will be parsed as a comment so @ prefixes inside {} will be ignored.

Some weird operator combinations (but valid JavaScript) may throw off the parser, if that happens, place that code in between curlies ({}), and post a bug report.

In this chapter you'll learn basics of programming with the Smart Mobile Studio. The chapter will focus more on the application architecture and programming patterns than on the Pascal language and on the IDE (see chapters *Smart Pascal* for the overview of the language and *Smart IDE* for the description of the programming environment).

Smart supports three different types of projects – Visual applications (i.e. Delphi-like form-based applications), Command-line applications (DOS-like applications) and Game applications.

To start creating new application, click the *New project* button. A dialog window will ask you to select the application type, name, folder and theme.



• Name

Name of the application storage file (.opp; see *File Management*). This will also be the name of the main application unit – the one that initializes and starts the application. The name must start with a letter and can contain letters and numbers.

• Location

Base folder for the application. You can click *Browse* and select another folder.

• Theme

Style sheet that will be used in the application. Currently, only iOS-like theme *iOS.css* is available.

• Create a new folder for my project

If checked, Smart will create subfolder with the application name in the *Location* folder and put application files there. If unchecked, files will be stored directly in the *Location* folder. My recommendation is to keep this setting checked.

When you click *OK*, Smart will create initial application files for you. Exact number of units in the project and their content will depend on the application type.

2.1 Message Dialogs

Since the initial release, Smart supported two ways to display dialog on the screen. Synchronous and unthemed ShowMessage is useful for simple notifications, especially while developing the application, while the asynchronous and themed ShowDialog offers greater flexibility and nicer look for the price of more complicated usage. In release [1.1.0.911] this was extended with ability to user user-designed forms as a modal dialogs. This feature is described in the next section - Modal Dialogs.

Sample code for this section is included in the MessageDialogs project.

2.1.1 ShowMessage

Smart implements two procedures for displaying a message on the screen.

```
procedure w3_ShowMessage(aText: String);
procedure ShowMessage(aText:String);
```

Actually, these two procedures are completely the same. If you look at the implementation (in the *w3system* unit), you'll see the proof.

```
procedure w3_ShowMessage(aText:String);
 1
 2
    begin
 3
      asm
        alert(@aText);
 4
5
      end;
    end;
 6
7
    procedure ShowMessage(aText:String);
8
    Begin
9
10
        alert(@aText);
11
12
      end;
13
    end;
```



[1.0.0-1.0.0] In the initial release of Smart Mobile Studio there was only w3_ShowMessage. ShowMessage was added in the first hotfix update.

As you can see from the code, ShowMessage merely calls the JavaScript alert function which displays a system dialog with an 'OK' button, rendered with the operating system 'theme'. In other words,

on Windows and OS X this dialog will not suit your Smart application, which will be (unless you have edited the stylesheet) visually similar to iOS applications.

This simple method displays a message with the ShowMessage call. The look of the dialog will vary across different operating systems.

```
procedure TForm1.TestShowMessage(Sender: TObject);
begin
ShowMessage('Hello from the ShowMessage dialog!');
end;
```



Result of the ShowMessage call (left - Windows; right - iOS)

ShowMessage is very useful for debugging because it is *synchronous* – when the dialog is displayed, your program will be stopped until the user clicks the OK button.

2.1.2 ShowDialog

Smart also implements a more advanced dialog with a caption, message, different buttons and themed display (in other words – it will look similar to the rest of the application regardless of the operating system that runs the application). There is, however, a downside. ShowDialog is asynchronous. In other words, when the code calls ShowDialog, execution will immediately continue with the next line of code. So how do you get the notification that the dialog was closed? You have to write an event handler, which is called when the user clicks a dialog button.

There are two procedures and two properties dedicated to showing a dialog. All are defined in the TCustomApplication class which lives in the *w3application* unit. An instance of this class is accessible through the global Application object.

```
procedure ShowDialog(aCaption: String; aText: String;
aOptions: TW3AlertOptions);
procedure CloseDialog;
property DialogActive: Boolean;
property OnDialogSelect: TW3AlertSelectEvent;
```

• ShowDialog

Displays a dialog with specified caption, message and buttons (the aOptions parameter).

• CloseDialog

Closes the active dialog.

• DialogActive

Returns True while the dialog is displayed.

• OnDialogSelect

Contains the event handler which will be called when user clicks a button in the dialog. Dialog will be closed *after* the event handler completes execution.

Dialog-related types are declared in the *w3dialogs* unit which is not automatically added to the uses list. In order to use the ShowDialog you'll have to add *w3dialogs* to the uses list yourself.

```
TW3AlertOptions = (aoYes, aoNo, aoYesNo, aoOK, aoCancel, aoOKCancel);
TW3AlertResult = (roYes, roNo, roOK, roCancel);
TW3AlertSelectEvent = procedure(Sender: TObject; aResult: TW3AlertResult);
```

• TW3AlertOptions

Possible combinations of buttons that can be displayed in a dialog.

• TW3AlertResult

The button that was pressed by the user.

• TW3AlertSelectEvent

Type of the OnDialogSelect event.

Following code fragment shows how to use ShowDialog.

```
procedure TForm1.ProcessDialogResult(Sender: T0bject;
1
      aResult: TW3AlertResult);
2
3
   begin
4
      if aResult = roYes then
        ShowMessage('You''re welcome.');
5
    end;
6
    procedure TForm1.TestShowDialog(Sender: TObject);
8
9
    begin
      Application.OnDialogSelect := ProcessDialogResult;
10
      Application.ShowDialog('ShowDialog Demo', 'Happy now?', aoYesNo);
12
   end;
```

The TestShowDialog method is called when the user clicks the button in the application. First it sets up the OnDialogSelect event which will be called when the user clicks a button in the dialog and then it displays the dialog with caption 'ShowDialog Demo', text 'Happy now?' and 'Yes'/'No' buttons.

When the user clicks either the 'Yes' or 'No', ProcessDialogResult is called. It checks the aResult parameter and displays a message if the user clicked 'Yes'.



Styled ShowDialog dialog displayed in a Windows browser

If you try this code (or just run the sample project *MessageDialogs*) you'll notice that the rest of the application is disabled while the dialog is displayed. If that is so, what use is the CloseDialog method?

Well, you have to remember that JavaScript is asynchronous. Even if the dialog is displayed, events may still be executed.

For example, the sample code implements a timer which closes the dialog after five seconds.

```
procedure TForm1 .ProcessDialogResult(Sender: T0bject;
1
      aResult: TW3AlertResult);
2
3
   begin
      FTimer := nil;
4
      if aResult = roYes then
5
        ShowMessage('You''re welcome.');
6
    end;
8
9
    procedure TForm1.TestShowDialog(Sender: TObject);
10
      Application.OnDialogSelect := ProcessDialogResult;
11
      Application.ShowDialog('ShowDialog Demo', 'Happy now?', aoYesNo);
12
      FTimer := TW3EventRepeater.Create(
13
        function(Sender: TObject): boolean
14
15
        begin
16
          Application.CloseDialog;
          Result := true; //don't repeat
17
18
        end,
        5000);
19
20
   end;
```

In addition to setting up OnDialogSelect event and calling ShowDialog, the TestShowDialog creates a TW3EventRepeater object (implemented in the *w3time* unit) which will call the anonymous function after five seconds (5000 milliseconds). The anonymous function calls CloseDialog to close the dialog and returns True which tells the TW3EventRepeater that it shouldn't call that anonymous function again. This is a simple way to implement a one-short timer.

ProcessDialogResult will destroy the FTimer event so that it is not called if the dialog was closed by the user.

2.2 Modal Dialogs [1.1.0.911]

While the user interface in Smart is centred around full-screen forms, sometimes we want to present a user with a smaller (not full screen) dialog. Since Smart 1.1, this is possible to do with APIs Application. ShowModal and Application. HideModal. ¹

Sample code for this section is included in the Smart demo *ModalDialogs.*

2.2.1 ShowModal

To display a modal dialog, an application must call the Application. ShowModal API. This method accepts six parameters:

- The name of the form that contains the dialog. This is the name used in the ApplicationStarting method to register the form.
- The name of the panel that will function as the main dialog element. You can define multiple dialogs in the same form by designing them each in its own panel. Each ShowModal call displays only the selected panel. Every component that is not a child of the panel will be ignored in the process.
- The name of the control that will receive focus when the dialog is displayed. You can leave this field empty (set to an empty string) which will cause all controls to be unfocused.
- A method accepting a single parameter of type TW3CustomForm. This method is called before the dialog is displayed and can initialize controls in the dialog. When the method is called, the parameter will contain the form object containing the dialog panel. (This also holds for the next two ShowModal parameters.)
- A method accepting a single parameter of type TW3CustomForm. This method is called when the dialog is closed with the mrOK status. (See *HideModal* below).
- A method accepting a single parameter of type TW3CustomForm. This method is called when the dialog is closed with the mrCancel status (see *HideModal* below). This method is optional if it is left out and the user cancels the dialog, nothing special will happen (besides the dialog disappearing, of course).

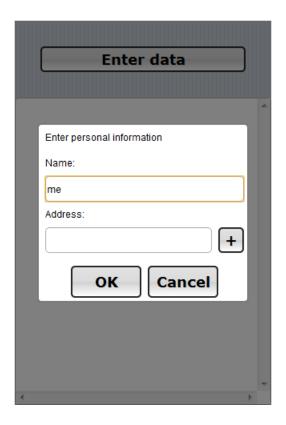
A modal dialog is always displayed centred over the owner form, regardless of the panel's position in the designer.

An example from the *ModalDialogs* demo shows how to use the ShowModal call. In this case, the dialog is stored in the W3pnlDialog on the *DlgPersonalInfo* form. After the dialog is displayed, the W3inpName control will be focused. The initialization method initializes two edit boxes in the dialog to an empty string. The "ok" method shows data from the dialog in a memo field. The "cancel" method just displays the status.

¹Smart 1.1 contains an RTL bug which makes the modal dialog functions unusable. You can easily fix it by editing one RTL file. The process is described in my post New in Smart 1.1: Bugs and Omissions. This bug was fixed in the 1.1.1 hotfix update.

```
procedure TForm1.InitializeObject;
1
   begin
2
      inherited;
3
      {$I 'Form1:impl'}
4
      W3Button1.OnClick := lambda
5
        Application.ShowModal('DlgPersonalInfo', 'W3pnlDialog', 'W3inpName',
6
          lambda (dialog)
7
            TDlgPersonalInfo(dialog).FullName := '';
8
            TDlgPersonalInfo(dialog).Address := '';
9
10
          end,
          lambda (dialog)
11
            AddLine('Name: ' + TDlgPersonalInfo(dialog).FullName);
12
            AddLine('Address: ' + TDlgPersonalInfo(dialog).Address);
13
          end,
14
          lambda (dialog)
15
            AddLine('Data entry was cancelled');
16
17
18
      end;
   end;
19
```

When a modal dialog is displayed, the rest of the form is hidden behind an opaque layer which also makes it non-responsive to clicks and touches.



2.2.2 HideModal

To close a dialog, application must call Application. HideModal with either a mrOK or mrCancel parameter.

An example from the *ModalDialogs* demo:

```
procedure TDlgPersonalInfo.InitializeObject;
2
   begin
      inherited;
 3
      {$I 'DlgPersonalInfo:impl'}
 4
5
      W3btnOK.OnClick := lambda Application.HideModal(mrOK); end;
 6
      W3btnCancel.OnClick := lambda Application.HideModal(mrCancel); end;
7
      W3btnLookupAddress.OnClick :=
        lambda
8
          Application.ShowModal('DlgSelectAddress', 'W3pnlDialog', '', nil,
9
10
            lambda (dialog)
              W3inpAddress.Text := 'selected';
11
            end);
12
13
        end;
14
    end;
```

As we see from this example, a modal dialog can display another modal dialog.

2.3 Running Code on Application Shutdown

When you are writing a Smart application, you have to keep in mind that it's not a 'real' application but a web page running in a browser. As such, the user can at any time close the tab with your application or close a browser. The browser may also crash and the whole operating system hosting the browser that runs your application may die unexpectedly.

In short, there are many ways of terminating your application and you cannot be sure that you'll be notified when the application is being closed. You cannot always expect that your application will be given a signal before it is being closed. The best way to run a code on application shutdown is not to do it. [That is, restructure your code so it doesn't have to be run on application shutdown. Execute it before that.]

If you really want to run code on application shutdown, Smart offers you few ways to do it. You can override TForm.FinalizeObject or TApplication.ApplicationClosing. You can also write an event handler for TApplication.OnBeforeUnload and for TApplication.OnUnload, which roughly correspond to Delphi's OnCloseQuery and OnClose events. The only problem is that your application will not run on all major browsers correctly so you'll have to find a workaround for some browsers or just remove support for them.

To test the browser support for executing shutdown code, I have written a small program (ApplicationShutdown) that logs various times during application startup and shutdown into cookie storage. Next time you start the application times from the previous run will be shown. Following points in the code are logged:

- TApplication.ApplicationStarting
- TForm.InitializeObject
- TForm.FinalizeObject
- TApplication.ApplicationClosing
- TApplication.OnBeforeUnload
- TApplication.OnUnload

As we could expect, the first two points which are executed while application is being started are always logged. Other four (called during the application shutdown) are correctly called only on some browsers.

Following browsers were tested:

- Chrome 21.0: Shutdown code is executed on first reload (F5) but not on subsequent ones. Shutdown code is not executed if tab or browser is closed.
- Firefox 14.0.1: Shutdown code is always executed.
- Opera 12.01: Shutdown code is never executed.
- Safari 5.1.7 on Windows: Shutdown code is always executed.

• Smart 1.0.122 (internal browser): Shutdown code is always executed except that OnBeforeUnload is never called.

• Safari on iOS 5.1: Shutdown code is always executed except that OnBeforeUnload is never called.

Following two screens show results from Opera (left) and Safari (right).

ApplicationStarting	ApplicationStarting
2012-08-19 08:40:01	2012-08-19 08:41:01
InitializeObject	InitializeObject
2012-08-19 08:40:01	2012-08-19 08:41:01
FinalizeObject	FinalizeObject
	2012-08-19 08:41:45
ApplicationClosing	ApplicationClosing
	2012-08-19 08:41:45
OnBeforeUnload	OnBeforeUnload
	2012-08-19 08:41:44
OnUnload	OnUnload
	2012-08-19 08:41:45

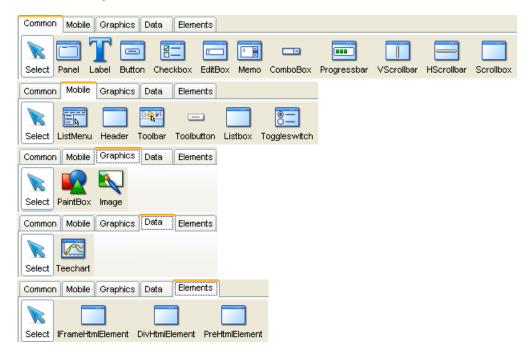
This is just an excerpt from the full book. Get it at http://leanpub.com/asmartbook!

Smart Mobile Studio comes with a basic control set suitable for variety of client applications. Currently you cannot add new controls to the designer; you can only create and use them in runtime. Most of this chapter focuses on built-in controls while the last section shows how to create and use custom controls. Smart also includes one control that cannot be instantiated in design view – TW3ScrollControl.

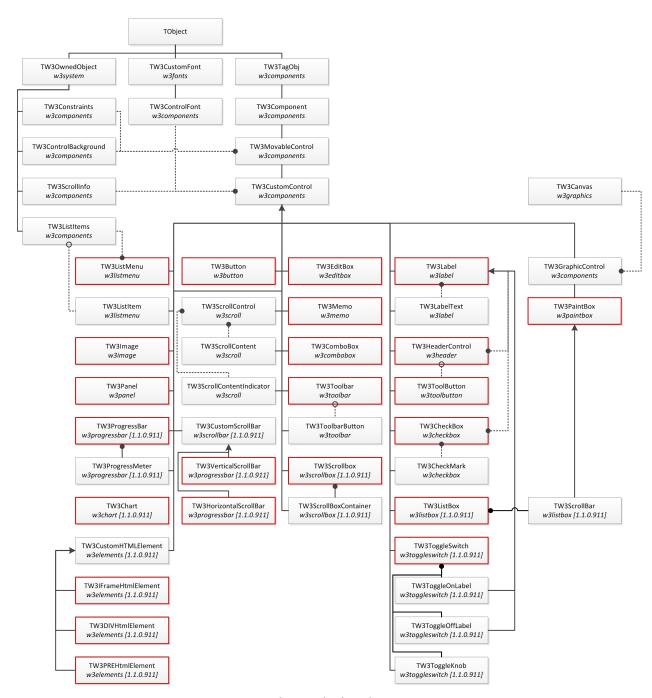
This chapter focuses on controls that are included with the Smart Mobile Studio, their usage and configurability. Actual mechanics of working with the Designer view are described in chapter Working with Designer.

3.1 Overview

Smart Mobile Studio comes with 23 preinstalled components: Panel, Label, Button, Checkbox, EditBox, Memo, ComboBox, Progress bar, Vertical scrollbar, Vertical scrollbar, ScrollBox, ListMenu, Header, Toolbar, Toolbutton, ListBox, Toggle switch, PaintBox, Image, TeeChart, IFrameHtmlElement, DivHtmlElement, and PreHtmlElement.



The chart below shows relations between classes that implement visual controls in Smart.



Visual controls class diagram

Classes with thicker red border represent visual components available in the designer. Solid lines indicate inheritance. Dashed lines indicate composition; circle indicates the owner. One-to-one relations are marked with a full circle and one-to-many relation with an open circle. *Italic* text denotes the name of the unit with the class definition. A version number in square brackets (e.g. [1.1.0.911]) indicates the Smart version where the component was introduced.

Tab Order

While users of Smart-generated application can move between controls with the <tab> key (HTML takes care of that), Smart currently doesn't have support for setting tab order. At the moment, tab order equals creation order. The only way to modify the tab order is to remove and reinsert controls (Cut and Paste) in correct order.

3.2 TW3TagObj

At the top of the control hierarchy is the TW3TagObj class. This is merely a TObject descendant that publishes a TagValue field. In this field programmer can store any integer value.

property TagValue: Integer read FTagValue write FTagValue;

3.3 TW3Component

Next in hierarchy is the TW3Component. This class describes a non-visual control (a.k.a. a *component*). It contains a public Parent property which holds the reference to the control's parent and a published Name property where you can set the name for the control. This name is used to access control's properties and methods from the code.

```
public
property Parent: TW3Component read FParent;
published
property Name: String read FName write setName;
```

3.3.1 Public Methods [1.1.0.911]

```
public

function ChildByName(const compName: String): TW3Component;

function ChildByHandle(const aHandle: THandle): TW3Component;

procedure EnumChildren(childProc: procedure (child: TW3Component));

function TopLevelParent: TW3Component;
```

• ChildByName

Locates a child component by its name.

• ChildByHandle

Locates a child component by the handle.

• EnumChildren

Calls a worker procedure (passed as a parameter) for each child component.

• TopLevelParent

Returns the top parent control (parent of a parent of a ...).

This is just an excerpt from the full book. Get it at http://leanpub.com/asmartbook!

3.4 EditBox

The *EditBox* control is implemented in the TW3EditBox class, which is defined in the *w3editbox* unit. It can be used to display an edit field.

A sample code for this section is included in the *ControlsEditBox* project.

TW3EditBox introcudes numerous published properties.

```
TW3InputType = (itNone,
1
2
                    itColor,
3
                    itDate,
                    itDateTime,
4
                    itDateTimeLocal,
5
                    itEmail,
6
                    itMonth,
7
8
                    itNumber,
9
                    itRange,
10
                    itSearch,
11
                    itTel,
                    itTime,
12
                    itUrl,
13
                    itWeek,
14
                    itPassword);
15
    property TextAlign: TTextAlign read getTextAlign write setTextAlign;
   property InputType: TW3InputType read getType write setType;
   property AutoCapitalize: Boolean read getAutoCapitalize
3
   write setAutoCapitalize;
4
5 property AutoCorrect: Boolean read getAutoCorrect write setAutoCorrect;
6 property PlaceHolder: String read getPlaceHolder write setPlaceHolder;
    property Text: String read getText write setText;
```

• TextAlign

Specifies text alignment (left, centre, right).

• InputType

Specifies the type of the input. Not all input types are supported in all browsers. Well-supported types are itNone, itNumber and itPassword.

• AutoCapitalize

Sets the value for the autocapitalize attribute. Supported in Safari on iOS.

• AutoCorrect

Sets the value for the autocorrect attribute. Supported in Safari on iOS.

• PlaceHolder

Sets the placeholder text. This text is displayed in the edit box if it has no user-provided content.

• Text

Contains the value entered in the edit box.

Useful inherited properties:

- Angle
- BorderRadius
- Color Specifies colour for the background colour. Default (c1None) makes editbox white.
- Enabled
- Transparent
- Visible
- StyleClass
- Name, Left, Top, Width, Height

Useful events:

• OnKeyUp Triggered when a key is released. New Text value is available at that moment.

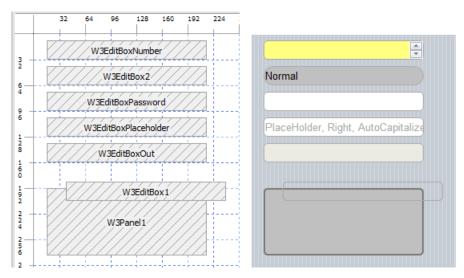


OnChanged event looks like a natural candidate to do any processing when the editbox content changes, but it is not called at all when a user types into an editbox.

3.4.1 Sample Application

The demo program *ControlsEditBox* contains bunch of editboxes and a panel. Following properties are set to non-default values.

```
W3EditBoxNumber:
1
      Color = $0080FFFF
 2
 3
   W3EditBox2:
 4
      BorderRadius = 15
5
      Color = clSilver
 6
      Text = 'Normal'
8
9
   W3EditBoxOut:
      Enabled = False
10
```



ControlsEditBox demo (left: design, right: runtime)

The program sets various properties and makes W3EditBox1 transparent. Then it assigns OnKeyUp event to all edit boxes except W3EditBoxOut. This event will copy current value of the Text property from focused editbox to W3EditBoxOut.

```
procedure TForm1.InitializeObject;
2
      procedure CopyEditBox(Sender: TObject; aKeyCode: integer);
 3
 4
        W3EditBoxOut.Text := TW3EditBox(Sender).Text;
5
      end;
6
7
8
    begin
      inherited;
9
      {$I 'Form1:impl'}
10
      W3EditBoxNumber.InputType := itNumber;
11
      W3EditBoxPassword.InputType := itPassword;
12
```

```
13
      W3EditBoxPlaceholder.AutoCapitalize := true;
      W3EditBoxPlaceholder.TextAlign := taRight;
14
      W3EditBoxPlaceholder.PlaceHolder := 'PlaceHolder, Right, AutoCapitalize';
15
      W3EditBox1.Transparent := true;
16
17
      W3EditBoxNumber.OnKeyUp := CopyEditBox;
18
      W3EditBox2.OnKeyUp := CopyEditBox;
19
      W3EditBoxPassword.OnKeyUp := CopyEditBox;
20
      W3EditBoxPlaceholder.OnKeyUp := CopyEditBox;
21
      W3EditBox1.OnKeyUp := CopyEditBox;
22
   end;
23
```

In the browser, we can see the visual difference between itNumber and other input types. Also, W3EditBox1 is transparent and W3EditBoxOut is disabled, which makes them visually different.

3.5 Listbox [1.1.0.911]

The *Listbox* control is implemented in the TW3ListBox class, which is defined in the *w3listbox* unit. It represents a flexible listbox control where each item can contain further graphic controls.

A sample code for this section is included in the *ControlsListbox* project. Further demos are included with the Smart distribution (*ListboxControl* and *ColorListbox*).

```
TW3ListBox = class
1
2
   public
3
      constructor Create(AOwner: TW3Component); override;
4
      function Add: integer; overload;
      function Add(s: string): integer; overload;
5
      procedure Clear;
6
      procedure Delete(itemIndex: integer);
7
8
      function IndexOf(item: TW3CustomControl): integer;
      procedure Insert(itemIndex: integer); overload;
9
      procedure Insert(itemIndex: integer; s: string); overload;
10
11
      procedure Resize; override;
      property AllowMoving: boolean;
12
      property Count: integer;
13
      property EnableAnimation: boolean;
14
      property HighlightedIndex: integer;
15
      property HighlightedItem: TW3CustomControl;
16
      property Items[idx: integer]: TW3CustomControl; default;
17
18
      property ItemClass: TW3CustomControlClass;
      property ItemHeight: integer;
19
      property InnerSpacing: integer;
20
      property MovingItem: TW3CustomControl;
21
      property SelectedIndex: integer;
22
      property SelectedItem: TW3CustomControl;
23
24
      property Styles: TW3ListBoxStyles;
      property Text[idx: integer]: string;
25
      property OnHighlighted: TItemSelectedEvent;
26
      property OnMouseDown: TMouseDownEvent;
27
      property OnMouseEnter: TMouseEnterEvent;
28
      property OnMouseExit: TMouseExitEvent;
29
      property OnMouseMove: TMouseMoveEvent;
30
31
      property OnMouseUp: TMouseUpEvent;
      property OnMouseWheel: TMouseWheelEvent;
32
      property OnSelected: TItemSelectedEvent;
33
      property OnTouchBegin: TTouchBeginEvent;
34
35
      property OnTouchMove: TTouchMoveEvent;
```

```
property OnTouchEnd: TTouchEndEvent;
form end;
```

AllowMoving

If set to True (default is False), user can move listbox items around with a mouse or a finger. The mouse/finger must be pressed down and not moving for at least 750 ms, before the item can be moved.

• Count

Returns the number of items in the listbox.

• EnableAnimation

If True (default), the component will animate item movement whenever an item is inserted or deleted. If False, items will just snap into place.

• HighlightedIndex

Returns the index (0-based) of the item under the mouse cursor or -1 if no item is highlighted.

• HighlightedItem

Contains the item which is currently highlighted or nil if no item is highlighted.

• Items

Allows read-only access to listbox items. In Smart, a listbox item is actually one of the controls so you can still modify the item's appearance by changing its properties.

• ItemClass

Contains a class name for items in the listbox. By default, this property is set to TW3Pane1, meaning that all items added to the listbox will actually be panel controls.

• ItemHeight

Specifies height for listbox items. Default is 32.

• InnerSpacing

Contains a spacing (in pixels) between the listbox boundaries and listbox items.

• MovingItem

Contains the item which is currently moved with a mouse or a finger. Contains nil if no item is being moved.

• SelectedIndex

Returns the index (0-based) of the item that is selected (was clicked with a mouse or touched with a finger). Returns -1 if no item is selected.

• SelectedItem

Contains the item which is currently selected or nil if no item is selected.

• Styles

Contains properties (described below) that change the listbox appearance.

• Text

Returns the Caption of the first TW3Label found in the item idx.

• OnHighlighted

Triggered when an item is highlighted.

• OnSelected

Triggered when an item is selected.



[1.1.0.911-1.1.1.5] In Smart 1.1.0.911, TW3ListBox component is missing a Clear method. My blog post New in Smart 1.1: Bugs and Omissions¹ describes how to fix that. This was corrected in the 1.1.1 hotfix release.

3.5.1 Listbox Styles

The properties of the TW3ListBoxStyles change the listbox appearance. They are accessed through the TW3ListBox.Styles property.

```
TW3ListBoxStyles = class
1
   public
2
3
      property Highlighted: string read FHighlighted write FHighlighted;
      property HighlightedColor: TColor
5
        read FHighlightedColor write FHighlightedColor;
      property HighlightVisible: boolean
 6
        read FHighlightVisible write FHighlightVisible;
7
8
      property Moving: string read FMoving write FMoving;
9
      property MovingColor: TColor read FMovingColor write FMovingColor;
      property Item: string read FItem write FItem;
10
      property ItemColor: TColor read FItemColor write FItemColor;
11
      property Selected: string read FSelected write FSelected;
12
      property SelectedColor: TColor read FSelectedColor write FSelectedColor;
13
14
      property SelectionVisible: boolean
        read FSelectionVisible write FSelectionVisible;
15
16
      property Text: string read FText write FText;
17
   end;
```

• Highlighted

Contains the style (default TW3Pane1) that is applied to a highlighted item.

¹http://www.smartprogrammer.org/2013/04/new-in-smart-11-bugs-and-omissions.html

• HighlightedColor

Contains the colour (default clNone) that is applied to a highlighted item.

• HighlightVisible

Specifies whether the Highlighted and HighlightedColor styles are used whenever an item is to be highlighted (default True).

Moving

Contains the style (default TW3Pane1) that is applied to an item being moved around with a mouse or a finger.

• MovingColor

Contains the colour (default clGrey) that is applied to an item being moved around.

• Item

Contains the style (default TW3Pane1) that is applied to a "normal" (not selected, highlighted or moving) listbox item.

• ItemColor

Contains the colour (default clNone) that is applied to a normal item.

• Selected

Contains the style (default TW3Panel) that is applied to a selected item.

• SelectedColor

Contains the colour (default clNone) that is applied to a selected item.

• SelectionVisible

Specifies whether the Selected and SelectedColor styles are used whenever an item is selected (default True).

• Text

Contains the style (default TW3Label) that is applied to a text item (created with the Add(s: string) function).

Useful inherited properties:

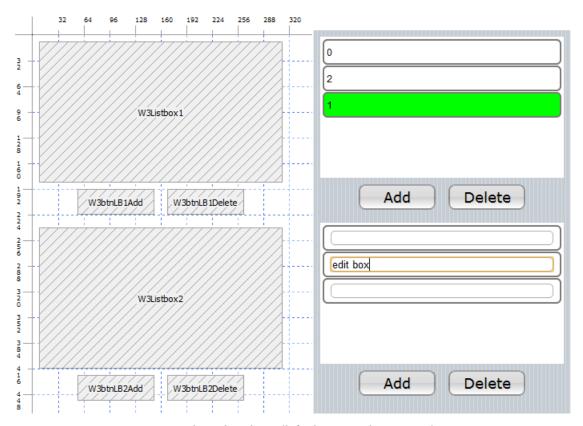
- Enabled
- Visible
- StyleClass
- Name, Left, Top, Width, Height

3.5.2 Sample Application

The demo program *ControlsListbox* contains two listboxes. No properties are set to non-default values. One of the listbox controls is, however, extensively configured in the code.

```
W3listbox1.Styles.SelectedColor := clLime;
W3listbox1.Styles.HighlightedColor := clAquamarine;
W3listbox1.Styles.MovingColor := clBurlyWood;
W3listbox1.AllowMoving := true;
```

This configuration sets the colours for *selected*, *highlighted* and *moving* items and allows item moving.



ControlsListbox demo (left: design, right: runtime)

The code also sets OnClick events for four buttons.

```
W3btnLB1Add.OnClick := lambda W3listbox1.Add(IntToStr(W3listbox1.Count)); end;
W3btnLB1Delete.OnClick := lambda DeleteFrom(W3listbox1); end;
W3btnLB2Add.OnClick := lambda AddToListbox2; end;
W3btnLB2Delete.OnClick := lambda DeleteFrom(W3listbox2); end;
```

The W3btnLB1Add button simply adds a text item. This overload of TW3ListBox. Add creates a default item control (which is the default TW3Panel in this example) and then creates a centred TW3Label child. The Caption property of this label is set to the text parameter, passed to the Add call.

The W3btnLB2Add button calls the other Add overload which just creates a default item control. The program can then create any child control(s) inside the item control.

To demonstrate this, the demo code creates a TW3EditBox child inside each item of the second listbox.

```
procedure TForm1.AddToListbox2;
begin
   var item := W3listbox2.Items[W3listbox2.Add];

var edit := TW3EditBox.Create(item);
edit.Left := 8;
edit.Top := 5;
edit.Width := item.ClientWidth - 16;
end;
```

The delete buttons both call the same code which deletes the selected item.

```
procedure TForm1.DeleteFrom(listbox: TW3ListBox);
begin
if listbox.SelectedIndex >= 0 then
listbox.Delete(listbox.SelectedIndex);
end;
```

3.6 Scrollbox [1.1.0.911]

The *ScrollBox* control is implemented in the TW3ScrollBox class, which is defined in the *w3scrollbox* unit. In many ways it can be considered as a replacement for the *Scroll Control* container, which was not improved since its release.

A scrollbox represents a *view* into a larger area (the scrollbox *content*) and automatically displays *scroll bars* so that the user can view any part of its content.

A sample code for this section is included in the *ControlsScrollBox* project.

TW3ScrollBox control introduces one public method (Update) and one public property (Content).

```
procedure Update;
property Content: TW3ScrollBoxContainer read FContent;
```

• Update

Repaints the scroll bars and the content.

• Content

Provides a parent for all controls that the scrollbox should manage.

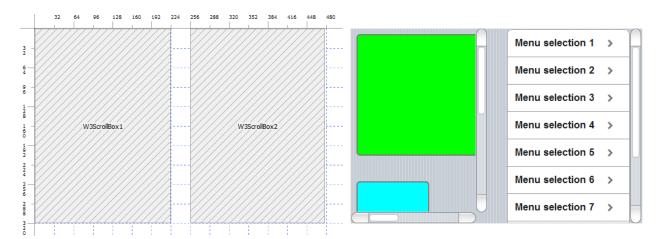
Useful inherited properties:

- Visible
- Name, Left, Top, Width, Height

3.6.1 Sample Application

Although the scrollbox control is placed on the toolbar and you can place it on the form in the designer, you cannot do the same with the scrollbox's content. You will have to create all child controls in the code. Don't forget that you will also have to add an appropriate unit to the uses lists (a *w3button* unit if you are using a TW3Panel and so on).

The demo program *ControlsScrollBox* contains two scrollboxes. No properties are set to non-default values.



ControlsScrollBox demo (left: design, right: runtime)

The code creates two panels inside the left scrollbox (W3ScrollBox1). The important detail is that the parent of those two panels is set to W3ScrollBox1.Content. (This is achieved by passing the W3ScrollBox1.Content to the TW3Panel constructor.)

InitializeObject also creates a TW3ListMenu control inside the W3ScrollBox2.Content and adds ten menu items to the menu.

```
procedure TForm1.InitializeObject;
1
    begin
2
      inherited;
3
4
      {$I 'Form1:impl'}
      var panel1 := TW3Panel.Create(W3ScrollBox1.Content);
5
      panel1.SetBounds(8, 8, 600, 200);
6
      panel1.Color := clLime;
7
      var panel2 := TW3Panel.Create(W3ScrollBox1.Content);
8
      panel2.SetBounds(8, 250, 120, 400);
9
      panel2.Color := clAqua;
10
      FListMenu := TW3ListMenu.Create(W3ScrollBox2.Content);
11
12
      for var i := 1 to 10 do
        FListMenu.Items.Add.Text := Format('Menu selection %d', [i]);
13
14
    end;
```

To resize the menu to the size of the scrollbox (sans the vertical scroll bar), FListMenu.Width is set in the Resize method.

```
procedure TForm1.Resize;
begin
inherited;
FListMenu.Width := W3ScrollBox2.Width - CNT_SCROLLBAR_SIZE - 2;
end;
```

If you run the program, you'll see that the left scrollbox displays two scroll bars with which you can scroll its contents, but the right scrollbox displays only the vertical scroll bar. The reason for this is that the scrollbox content in the second case is narrow enough to fit in the view completely.

3.7 Toolbar

The *Toolbar* control is implemented in the TW3Toolbar class, which is defined in the *w3toolbar* unit. Its purpose is to display a toolbar with buttons.

A sample code for this section is included in the *ControlsToolbar* project.

TW3Too1bar introduces public method Add and three published properties.

```
public

function add: TW3ToolbarButton;

published

property ButtonSpace: Integer read FBtnSpacing write setBtnSpacing;

property ButtonWidth: Integer read FBtnWidth write setBtnWidth;

property ButtonHeight: Integer read FBtnHeight write setBtnHeight;
```

Add

Adds a new button to the toolbar. Buttons are added from left to right.

• ButtonSpace

Specifies distance between buttons in pixels.

• ButtonWidth

Specifies a width for each button.

• ButtonHeight

Specifies a height for each button. This property **must** be set in the Resize method or toolbar will not be displayed correctly.



Toolbar **must** be resized once to work correctly. This resizing must occur in the Resize method.

Method Add creates new object of class TW3Too1barButton. This class publishes two properties.

```
property Glyph: TW3Image read FGlyph;
property Caption: String read FCaption write setCaption;
```

• Glyph

Image displayed on the button.

• Caption

Caption displayed on the button.

Button can show a caption or a caption and an image.



A caption must be set (at least to a space) for a glyph to be visible. This is most probably a bug, not a design choice.

Useful inherited properties:

- Visible
- Name, Left, Top, Width, Height

Toolbar gets its look from the CSS styling. It is not useful to change default values of its properties (except, of course, the name and bounds).



It is not possible to disable a toolbar. Setting Enabled := False does nothing. It is possible to disable individual buttons by setting TW3ToolbarButton. Enabled to False, but the result is not visually pleasing as disabled buttons don't look like buttons anymore.

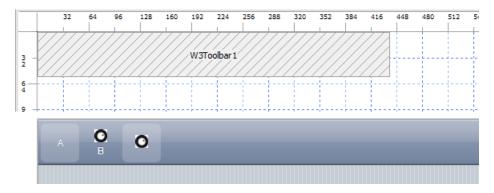
Useful events:

• TW3ToolbarButton.OnClick

Write the OnClick event for a button to detect when the button has been pressed.

3.7.1 Sample Application

The demo program *ControlsToolbar* contains one toolbar. All properties are left at default values.



ControlsToolbar demo (top: design, bottom: runtime)

The program creates three buttons, all with the same OnClick handler which merely reports the caption of the clicked button. First button has a caption, second a caption and a glyph, and the third one only a glyph (but the caption has to be set to a space for the glyph to be visible). The second button is disabled.

```
procedure TForm1.InitializeObject;
2
      procedure ReportClick(Sender: TObject);
3
 4
        ShowMessage('You clicked: ' + (Sender as TW3ToolbarButton).Caption);
5
      end;
6
8
   var
9
      btn: TW3ToolbarButton;
10
11 begin
      inherited;
12
     {$I 'Form1:impl'}
13
      btn := W3Toolbar1.Add;
14
      btn.Caption := 'A';
15
16
      btn.OnClick := ReportClick;
      btn := W3Toolbar1.Add;
17
      btn.Caption := 'B';
18
      btn.Glyph.LoadFromURL('res\smart.png');
19
      btn.Enabled := false;
20
      btn.OnClick := ReportClick;
21
      btn := W3Toolbar1.Add;
22
23
      btn.Caption := ' ';
      btn.Glyph.LoadFromURL('res\smart.png');
24
      btn.OnClick := ReportClick;
25
26
    end;
27
    procedure TForm1.Resize;
29 begin
30
      inherited;
     W3Toolbar1.Width := ClientWidth;
31
     W3Toolbar1.ButtonWidth := 48;
32
     W3Toolbar1.ButtonHeight := 48;
33
      W3Toolbar1.ButtonSpace := 3;
34
   end;
35
```

The program also sets toolbar width to the width of the form and assigns Button* properties in the Resize handler.

This is just an excerpt from the full book. Get it at http://leanpub.com/asmartbook!

Smart Mobile Studio comes with an extensive collection of run-time libraries (RTL). They are used just like in all modern Pascal-like languages, by listing libraries in the uses statement. Run-time libraries contain various interface and helper functions that help you write programs suitable for deployment in browsers and on mobile devices.

Run-time libraries are stored in the %ProgramData%\Optimale Systemer AS\Smart Mobile Studio\RTL folder and its subfolders. Parts of the RTL that are based on external libraries are stored in the %ProgramData%\Optimale Systemer AS\Smart Mobile Studio\Libraries. If you have to make a change to an RTL unit, the best approach is to copy the file to the application folder and modify the copy.

This chapter focuses on some of the RTL units that we deem especially important for writing weband mobile device-friendly applications. Each topic is covered in a separate section.

The rest of this chapter lists all RTL units alphabetically and provides a short description for each. Section names represent names of folders where units are stored.

4.1 RTL

• W3Animation [1.1.0.911]

Contains classes that simplify animating user interface elements.

• W3Application

Contains classes implementing the global Application object. This functionality is common to all application types implemented in Smart Mobile Studio.

• W3BmpFont

Implements support for storing and rendering bitmap fonts. An example is included in the demo *Spartacus*.

• W3Borders

Implements support for painting borders of a GUI control.

• W3BufferedImage [1.0.1.120]

Implements a *background image*, TW3BufferedImage. The image contains a canvas which is not visible on the screen. You can paint on this canvas using standard graphics mechanisms and at a later stage draw the background image on a visible canvas. [^buffered-image-eric] [^buffered-image-eric]: Eric Grange wrote an article about the TW3BufferedImage at http://delphitools.info/2012/05/24/buffered-image-for-smartms/.

• W3Buffers

Allows you to create memory buffers, more or less the same as AllocMem does under Delphi.

• W3Button [1.1.0.911]

Contains implementation of a button control.

• W3CheckBox [1.1.0.911]

Contains implementation of a checkbox control.

 $\bullet \ \ W3Collapsible Panel \ ^{[1.1.0.911]}$

Contains implementation of a collapsible panel component, which is "experimental and unfinished at the moment" (quote from the source file).

• W3Colors [1.1.0.911]

Colour manipulation class capable of converting between colour models (RGB, HSL) and adjusting colours.

• W3ComboBox [1.1.0.911]

Contains implementation of a combo box control.

• W3Components

Contains classes from which the graphical user interface controls are derived. This unit is described in chapter Smart Controls.

W3Console

Contains a simple control that implements basic *console* functionality – you can write a line to the screen and clear the contents. This control is used in Command-line applications.

• W3ConsoleApp

Contains the base class implementing a *Command-line application*. This unit is described in chapter Command-line Applications.

• W3CookieStorage [1.0.1.120]

Implements support for web browser cookies. It is further described in section Local Storage and Cookies.

• W3CssNames

Implements backend support for assigning CSS classes to graphical user interface controls.

• W3Ctrls

[-1.0.1.120] Contains graphical user interface controls (edit box, label, button etc.). This unit is described in chapter Smart Controls.

 $^{[1.0.1.911]}$ Provides a unit namespace wrapping all units that implement user interface controls.

• W3DbSql [1.1.0.911]

Implements the Web SQL Database¹ API.[^WebSql-blog] [^WebSql-blog]: An article about the webSQL API appeared on the Smart Mobile Studio blog (http://smartmobilestudio.com/2013/01/08/working-with-websql/).

• W3Dialogs

Contains implementation of an Application. ShowDialog method. An example of using this method is shown in chapter Message Dialogs.

• W3Dictionaries

Contains various *dictionary* classes used to store key-referenced values. An example can be found in the *MultiFingerPaint* demo.

W3EditBox [1.1.0.911]

Contains implementation of an edit box control.

• W3Effects

Implements classes that wrap CSS animations. *MegaDemo* demo application uses this unit in the *Rotation demo* (unit demos, class TRotateDemoForm).

• W3Elements [1.1.0.911]

Contains implementation of an iframe element, div element and pre element controls.

- W3FlowingCtrls [1.1.0.911]
- W3Fonts

Implements TW3CustomFont class which manages HTML font properties. It is used inside graphical user interface controls.

¹http://en.wikipedia.org/wiki/Web_SQL_Database

W3Forms

Implements the *form* concept in a HTML world. Forms and navigation between them are described in section Forms and Navigation.

• W3Game

Contains graphical user interface control implementing behaviour needed in a typical *Game application*. This unit is described in chapter Writing Games.

• W3GameApp

Contains the base class implementing *Game application*. This unit is described in chapter Writing Games.

• W3Geolocation [1.1.0.911]

Implements geolocation support.

• W3Graphics

Contains classes that allow you to draw on the screen. They are further described in section Graphics.

• W3Header [1.1.0.911]

Contains implementation of a header control.

• W3Image [1.1.0.911]

Contains implementation of an image control.

• W3Inet

Contains a wrapper for the JavaScript Xm1HttpRequest class and JSONP² queries. Section Networking covers this unit in more detail.

• W3Label [1.1.0.911]

Contains implementation of a label control.

• W3Layout

Contains classes implementing the *Layout Manager* – a subsystem that helps with automatic positioning of controls on the screen. It is further described in the section Layout Manager.

• W3Listbox [1.1.0.911]

Contains implementation of a listbox control.

• W3ListMenu ^[1.1.0.911

Contains implementation of a list menu control.

• W3Lists

Contains implementation of TStringList and TObjectList classes that allow you to store strings or objects (respectively) in an integer-indexed list.

• W3MediaElements [1.1.0.911]

Contains support for media (audio and video) playback.

²http://en.wikipedia.org/wiki/JSONP

• W3Memo [1.1.0.911]

Contains implementation of a memo control.

• W3Motion

Implements classes to support iOS acceleration hardware. See the demo program *AccelerationDemo* for an example. Section Accelerometer describes this unit in more detail.

• W3MouseCapture [1.1.0.911]

Contains support for global (full-window) mouse capture.

• W3MouseTouch [1.1.0.911]

Extends TW3CustomControl with handlers that combine mouse and touch input.

• W3PageControl [1.1.0.911]

Contains implementation of a page control component, which is "experimental and unfinished at the moment" (quote from the source file).

• W3Panel [1.1.0.911]

Contains implementation of a panel control.

• W3Polygons

Contains helper methods that manipulate an array of points as if it represents vertexes in a polygon.

• W3ProgressBar [1.1.0.911]

Contains implementation of a progress bar control.

• W3RegEx

Contains wrapper for the JavaScript RegExp class and regex helpers for the string object. An example of regular expression usage can be found in demo *RegExDemo*. It is further described in section Regular Expressions.

• W3Scroll [-1.1.0.210]

Contains a class implementing a *scrolling control*. This control is described in the section Scroll Control.

• W3ScrollBar [1.1.0.911]

Contains implementation of a vertical and horizontal scrollbar controls.

• W3ScrollBox [1.1.0.911]

Contains a class implementing a scrollbox control.

• W3Spinner [1.1.0.911]

Contains a full-screen spinner ("please wait, application is busy") component.

• W3Splitter [1.1.0.911]

Contains implementation of a splitter component, which is "experimental and unfinished at the moment" (quote from the source file).

• W3Sprite3D

Contains the TW3Sprite class, which implements CSS-supported *sprite* – a movable image that 'floats' on its own Z position (that is, in front or behind the application and/or other sprites). An example of TW3Sprite usage can be found in the *Spartacus* demo.

• W3SpriteSheet

Contains *sprite sheet* support. A sprite sheet is an image that contains multiple small images which can be independently presented on the graphical interface.

• W3Storage

Contains interface to the browser's per-application storage mechanism. It is further described in section Local Storage and Cookies.

• W3Styles [1.1.0.911]

A wrapper around w3c.CSS and w3c.StyleSheets units that simplifies CSS manipulation.

• W3System

Contains various helper procedures and functions.

• W3Table [1.1.0.911]

A wrapper around the HTML TABLE, TR and TD tags.

• W3Time

Contains classes that wrap the JavaScript setInterval function to implement timer functionality. F.i. *MandelbrotExplorer* demo uses this unit to set up a timer FTimer to redraw next part of the screen in the MandelbrotCanvas unit.

• W3ToggleSwitch [1.1.0.911]

Contains implementation of a toggle switch control.

• W3Toolbar [1.1.0.911]

Contains implementation of a toolbar control.

• W3ToolButton [1.1.0.911]

Contains implementation of a toolbutton control.

• W3Touch

Implements support for *touch* and *gestures*. Examples can be found in demos *MultiFingerPaint* and *GestureDemo*. If is further described in section Touch and Gesture.

• W3Utils

Contains various utilities working on integer and string arrays.

4.2 RTL\PhoneGap

• PhoneGapAPI [1.1.0.911]

Contains pascal translation of the PhoneGAP API.

4.3 RTL\Steema

• W3Chart [1.1.0.911]

Contains implementation of a teechart control.

4.4 RTL\System

4.5 RTL\w3c

 $\bullet \ \ w3c. Ambient Light Events \ ^{[1.1.0.911]}$

Implements interfaces to ambient light events.

• w3c.Canvas [1.0.1.120-]

 $Implements\ interface\ to\ the\ JavaScript\ {\tt HTMLCanvasElement}\ class.$

• w3c.Canvas2DContext $^{[1.0.1.120-]}$

Implements interfaces to JavaScript canvas classes.

• w3c.CSS [1.1.0.911]

Implements interfaces to JavaScript CSS manipulation classes.

• w3c.Date [1.1.0.911]

Implements interface to JavaScript Date class.

• w3c.DOM [1.0.1.120-]

Implements interfaces to JavaScript DOM classes.

• w3c.ProximityEvents [1.1.0.911]

Implements interfaces to proximity events.

• w3c.StyleSheets [1.1.0.911]

Implements interfaces to JavaScript style sheet manipulation classes.

• w3c.TypedArray [1.0.1.120-]

Implements API for interoperability with arrays of native binary data³.

• w3c.WebAudio [1.1.0.911]

Implements interfaces to JavaScript WebAudio classes.

• w3c.WebGL [1.0.1.120-]

Implements interfaces to JavaScript WebGL classes.

³Typed Array Specification, http://www.khronos.org/registry/typedarray/specs/latest/

4.6 Libraries

• Complex [1.1.0.911]

Complex number support.

• Shims.InternetExplorer [1.1.0.911]

 $Adds \ {\tt indexOf} \ support \ in \ arrays \ for \ Internet \ Explorer.$

97

4.7 Libraries\Box2D

• Box2DWrapper

A wrapper around the $Box2D^4$ 2-dimensional physics simulator engine.

4http://box2d.org/

4.8 Libraries\GLScene

Conversion of the GLScene $^{\scriptscriptstyle 5}$ OpenGL 3D library that works with WebGL engine. Currently contains following units:

- GLS.Base
- GLS.Mesh
- GLS.Scene
- GLS.Vectors

 $^{^{5}} http://glscene.sourceforge.net/wikka/HomePage$

4.9 Libraries\RemObjects

• RemObjectsSDK

Wrapper classes for the RemObjects JavaScript client-side library.

4.10 Libraries\SmartUnit

Unit testing framework for Smart.

• SampleTestUnit [1.1.0.911]
Sample unit.

• TestFramework [1.1.0.911]

Testing framework.

• TextTestRunner [1.1.0.911]

Test runner that works as a console application.

4.11 Graphics

In Smart, painting directly to the display is implemented using the HTML5 canvas⁶ element. It exposes rectangular area on which the application can draw using simple primitives – lines, circles, rectangles, text and others.

This behaviour is encapsulated in the TW3Canvas class which is implemented in the *w3graphics* unit. TW3Canvas is a big class, too big to be fully covered by this book. I will focus on the most important methods and properties which will get you started and then you can continue exploring the code to find about more advanced (and complicated) members.

To create a 'paintable' area in a Visual application, you should use the TW3PaintBox control which exposes the Canvas: TW3Canvas property. Game applications get access to the canvas through the Canvas: TW3Canvas property of the PaintView method.

Code examples from this chapter are collected in the sample application Graphics.

4.11.1 Colour

In the HTML5 Canvas infrastructure colours are represented as strings. There are many ways to specify a colour – '#RRGGBB' ('standard' HTML way, for example '#FF0000'), 'rgb(r,g,b)', 'rgba(r,g,b,a)' (colour + transparency channel), named colours ('orange'), 'hsl(h,s,l)' and more⁷. There's also a built-in function ColorToWebStr, which converts TColor type into a '#RRGGBB' representation.

Examples in this chapter use different ways to represent colours – just to show you the extensive array of possibilities.

4.11.2 Line

Painting a line with HTML5 Canvas is quite a complicated operation. First you have to set StrokeStyle – a property that defines how the line will be painted. For the time being you can think of it as representing a colour. The example below sets StrokeStyle to black. Then you start a *path*, which is a sequence of lines and curves by calling BeginPath. In our example the path will contain a bunch of vertical lines. Each of them is a result of two operations – MoveToF moves the path's *current position* to some point and LineToF draws a straight line from the current position to another point on the canvas. At the end, Stroke is called to display the path on the canvas.

 $^{^6}http://developer.mozilla.org/en-US/docs/Canvas_tutorial$

⁷http://www.w3.org/TR/2003/CR-css3-color-20030514/



All coordinates in HTML5 Canvas are **floating-point** values. When vertical and horizontal coordinate are needed, first parameter represents the horizontal and second parameter the vertical one. Top left corner of the canvas has coordinates (0,0). The rest of the canvas can be described using positive coordinates (coordinate values increase to the right and to the bottom).

Let me repeat this again – nothing is painted until the code calls the Stroke method. Because of this approach, you can create multiple paths in memory and display them when needed.

```
1
    procedure TPaintForm.PaintLines;
2
    var
      i: integer;
3
    begin
4
5
      ClearCanvas;
      Canvas.StrokeStyle := 'rgb(0,0,0)';
6
7
      Canvas BeginPath;
      for i := 3 to Canvas.Context.Width - 3 step 4 do begin
8
        Canvas.MoveToF(i, 3);
9
        Canvas.LineToF(i, Canvas.Context.Height - 3);
10
      end;
11
      Canvas.Stroke;
12
    end;
13
```

If you run this method (click on the *Lines* button in the Graphics application) you'll notice that the lines are somehow blurry. This does not point to a deficiency in the Smart RTL or in the HTML5 Canvas implementation, but to a rather interesting detail about the coordinate system in the HTML5 Canvas.



A coordinate 0,0 in the HTML5 Canvas system does not represent top left pixel, but the top left corner of that same pixel. Other three corners of that pixel have coordinates (0,1), (1,0) and (1,1).

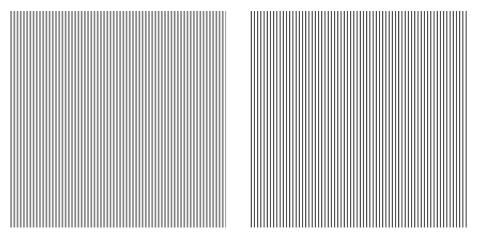
When you paint a line from coordinate (1,1) to (1,10) you are actually painting a vertical line between two columns of pixels. An anti-aliasing (edge-smoothing) algorithm changes one-pixel wide black line into two pixel-wide grey rectangle. To paint a pixel-wide vertical line, you have to paint it through the centre of the pixel – from (1.5,1.5) to (1.5,9.5).

This process is beautifully explained in the HTML5 tutorial⁸ written by Mark Pilgrim.

The code below paints a set of sharp black vertical lines.

```
procedure TPaintForm.PaintLines05;
 1
2
    var
 3
      i: integer;
    begin
 4
      ClearCanvas;
 5
      Canvas.StrokeStyle := 'rgb(0,0,0)';
 6
      Canvas.BeginPath;
 7
 8
      for i := 3 to Canvas.Context.Width - 3 step 4 do begin
9
        Canvas.MoveToF(i+0.5, 3.5);
        Canvas.LineToF(i+0.5, Canvas.Context.Height - 3.5);
10
      end;
11
      Canvas.Stroke;
12
13
    end;
```

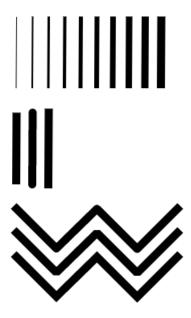
 $^{^8}http://diveintohtml5.info/canvas.html\#pixel-madness$



Vertical lines drawn by the both algorithms - PaintLines on the left and PaintLines05 on the right

Every line has three configurable properties that affect the way it is painted — LineWidth, LineCap and LineJoin. The first one determines line width, second one the way line endings are painted (possible values are 'butt', 'round' and 'square') and the third determines how multiple lines are joined at the common end (possible values are 'round', 'bevel' and 'miter').

The PaintLineProperties method in the sample program displays all possible ways of painting a line.



Line properties - LineWidth (on top), LineCap (in the middle) and LineJoin (at the bottom)

4.11.3 Circle

The ArcF method paints not only arcs but also full circles. ArcF creates a path with *interior* which can be filled with a FillStyle style if the program calls the Fill method. (As in the StrokeStyle

example, let's assume that this style holds a colour reference for now.)

The ArcF method accepts six parameters – coordinates of the centre, radius, starting angle, ending angle and a direction (True - anticlockwise, False - clockwise). Angles are specified in *radian* units; angle 0 indicates the point on the arc/circle that is right to the centre (horizontal distance from that point to the centre is positive and vertical distance is zero). To paint a circle, starting and ending angle must be 2*Pi apart.

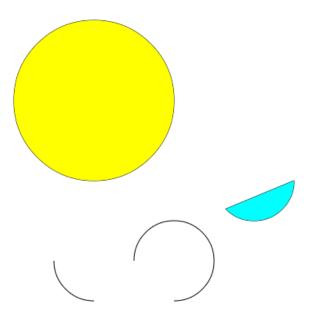
The code below paints a yellow circle and three arcs. The first BeginPath - ArcF - Stroke - Fill sequence draws a circle (start angle = 0, end angle = 2*Pi) and fills it with a yellow colour (Canvas.Fill).

The second sequence draws just a part of the circle from angle 0 to angle Pi*3/4 in a clockwise direction. Because the ClosePath is called, both ends of the arc are connected with a straight line. The interior of the arc is filled with a cyan colour.

The third and the fourth sequence both paint an arc from angle Pi/2 to angle Pi, but in a different manners – former in the anticlockwise and latter in the clockwise direction. Only the arc but not the connecting line is painted because ClosePath was not called.

```
procedure TPaintForm.PaintCircle;
 1
2
    begin
      ClearCanvas;
 3
 4
      Canvas.StrokeStyle := "#000000";
      Canvas.FillStyle := "#FFFF00"; //yellow
5
      Canvas BeginPath;
 6
      Canvas.ArcF(200,200,100,0,PI*2,true);
 7
      Canvas.Stroke;
8
      Canvas.Fill;
9
10
11
      Canvas.FillStyle := "#00FFFF"; //cyan
      Canvas . BeginPath;
12
      Canvas.ArcF(400,300,50,0,PI*3/4,false);
13
      Canvas.ClosePath;
14
      Canvas.Stroke;
15
      Canvas.Fill;
16
17
18
      Canvas.FillStyle := "#FF00FF"; //magenta
      Canvas BeginPath;
19
      Canvas.ArcF(300,400,50,PI/2,PI,true);
20
      Canvas.Stroke;
21
22
      Canvas BeginPath;
23
24
      Canvas. ArcF(200, 400, 50, PI/2, PI, false);
```

```
25 Canvas.Stroke;
26 end;
```



Painting circles and arcs

Since version $^{[1.1.0.911]}$, unit W3Graphics contains a code to paint an ellipse specified by a bounding rectangle. The Ellipse method is used the same as the ArcF method.

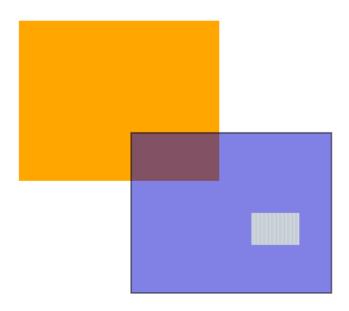
```
1    Canvas.FillStyle := "#808080";
2    Canvas.BeginPath;
3    Canvas.Ellipse(50,350,300,400);
4    Canvas.ClosePath;
5    Canvas.Stroke;
6    Canvas.Fill;
```

4.11.4 Rectangle

HTML5 Canvas also supports drawing rectangles. You can draw only the rectangle outline by calling the StrokeRectF method, fill the rectangle by calling the FillRectF method or clear the rectangle (make it transparent) by calling the ClearRectF method. All three methods also exist in a variation without the trailing F which accepts one TRect parameter instead of four (top, left, width, height).

The code below paints two overlapping rectangles and then makes a hole in the second one. The second rectangle is filled with a semi-transparent colour (the fourth parameter to the rgba call is 0.5, representing 50% transparency), which makes the overlapping part painted in a kind-of-violet colour.

```
procedure TPaintForm.PaintRectangles;
1
    begin
2
3
      ClearCanvas;
      Canvas.FillStyle := "orange";
 4
      Canvas.FillRectF(10, 10, 250, 200);
5
      Canvas.FillStyle := "rgba(0,0,200,0.5)";
6
      Canvas.FillRectF(150, 150, 250, 200);
      Canvas.StrokeRectF(150, 150, 250, 200);
8
9
      Canvas.ClearRectF(300, 250, 60, 40);
10
   end;
```



Painting rectangles

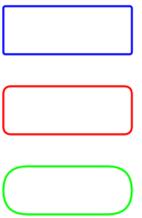
4.11.5 Curve

HTML5 Canvas enables you to paint interpolated curves using quadratic and Bezier interpolation. Curves are hard to explain and even harder to use so I'll just be referring you to the Mozilla Developer Network's tutorial Drawing Shapes'. The two examples in the sample application also come from this tutorial.

There is, however, one very practical use of quadratic interpolation – it allows us to draw rectangles with rounded corners. The following procedure (also taken from the Mozilla's tutorial) will paint a rectangle with specific top-left corner, width, height and corner radius.

⁹https://developer.mozilla.org/en-US/docs/Canvas_tutorial/Drawing_shapes

```
procedure RoundedRect(Canvas: TW3Canvas; x, y, width, height, radius: float);
2 begin
3
      ClearCanvas;
4
      Canvas.BeginPath;
      Canvas.MoveToF(x,y+radius);
5
      Canvas.LineToF(x,y+height-radius);
6
      Canvas.QuadraticCurveToF(x,y+height,x+radius,y+height);
7
      Canvas.LineToF(x+width-radius,y+height);
8
9
      Canvas.QuadraticCurveToF(x+width,y+height,x+width,y+height-radius);
10
      Canvas.LineToF(x+width,y+radius);
      Canvas.QuadraticCurveToF(x+width,y,x+width-radius,y);
11
      Canvas.LineToF(x+radius,y);
12
      Canvas.QuadraticCurveToF(x,y,x,y+radius);
13
      Canvas.Stroke;
14
15
   end;
16
17
    procedure TPaintForm.PaintRoundedRect;
    begin
18
19
      ClearCanvas;
20
      Canvas.LineWidth := 2.5;
      Canvas.StrokeStyle := 'rgb(0, 0, 255)';
21
22
      RoundedRect(Canvas, 20.5, 20.5, 160, 60, 3);
23
      Canvas.StrokeStyle := 'rgb(255, 0, 0)';
      RoundedRect(Canvas, 20.5, 120.5, 160, 60, 10);
24
      Canvas.StrokeStyle := 'hsl(120, 100%, 50%)';
25
      RoundedRect(Canvas, 20.5, 220.5, 160, 60, 30);
26
      Canvas.LineWidth := 1;
27
28
   end;
```



Rounded rectangles with corner radii 3, 10 and 30 (from top to bottom)

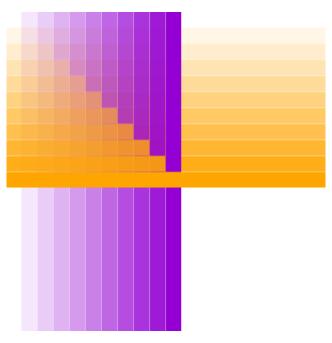
4.11.6 Transparency

We have already seen (in the *Rectangle* section) that a (stroke or fill) style can contain a transparency parameter next to the colour. Transparency can take values from 0 (totally transparent) to 1 (fully opaque) and is provided as a fourth parameter in a rgba(r, g, b, a) colour specification.

Canvas also supports a GlobalAlpha property which sets transparency for all paint operations. GlobalAlpha can also take values from 0 (transparent) to 1 (opaque).

The example below paints two sets of colour stripes with different GlobalAlpha settings. You'll notice that if the GlobalAlpha is greater than about 0.25, it matters a lot which colour is painted on the top.

```
procedure TPaintForm.PaintGlobalAlpha;
1
2
    var
 3
      i: integer;
4
    begin
      ClearCanvas;
5
      for i := \emptyset to 10 do begin
6
        Canvas.GlobalAlpha := i/10;
        Canvas.FillStyle := 'darkviolet';
8
9
        Canvas.StrokeStyle := '#000000';
        Canvas.FillRectF(20*i+0.5, 0.5, 19.5, 399.5);
10
        Canvas.FillStyle := 'orange';
11
        Canvas.FillRectF(0.5, 20*i+0.5, 399.5, 19.5);
12
13
      end;
    end;
```

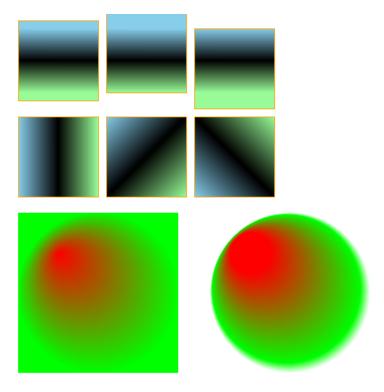


Colour stripes with variable GlobalAlpha

4.11.7 Gradient

Gradients are a powerful tool for creating fill styles in which one colour gradually changes into another. They are, however, also extremely hard to use. I'll try to explain some mysteries in this section.

Canvas supports two kinds of gradients – linear and circular. The difference between them is best explained in the picture.



Six linear and two circular gradients

Linear gradient is created by calling CreateLinearGradientF function. It accepts four parameters – coordinates of the starting point and coordinates of the end point. Colour will be interpolated along the line connecting these two points.

To set up the colour values at arbitrary points along the line, call the AddColorStop function. It accepts the position, which is a number from 0 to 1 (0 representing the starting point and 1 representing the end point) and a TColor value. Values between two colour stops will be calculated automatically by using linear interpolation. At the end, call UseGradient to assign the gradient to the canvases fill style.



To change colour abruptly, assign two colours to the same position.

The tricky part with gradients is that they are not relative to the object you are drawing but absolutely positioned on the canvas itself. In other words - if you want to use gradient to fill rectangle that stretches from point (100, 60) to (200, 120), the gradient must also cover points in this interval. I know this explanation is not very clear, so take a look at the following code fragment from the PaintGradients method.

```
linGrad := Canvas.CreateLinearGradientF(0, 20, 0, 100);
1
      linGrad.AddColorStop(0, clSkyBlue);
2
      linGrad.AddColorStop(0.5, clBlack);
3
      linGrad.AddColorStop(1, RGBToColor($98, $FB, $98));
 4
      Canvas.UseGradient(linGrad);
5
      Canvas.StrokeStyle := 'orange';
6
      Canvas.FillRectF(10.5, 10.5, 100, 100);
      Canvas.StrokeRectF(10.5, 10.5, 100, 100);
8
9
      Canvas.FillRectF(120.5, 0.5, 100, 100);
      Canvas.StrokeRectF(120.5, 0.5, 100, 100);
10
      Canvas.FillRectF(230.5, 20.5, 100, 100);
11
      Canvas.StrokeRectF(230.5, 20.5, 100, 100);
12
```

This code creates linear gradient stretching from (0,20) to (0,100). Then it paints three rectangles.

The first rectangle is defined by corners (10.5, 10.5) and (110.5, 110.5). The top part of the rectangle (from vertical offset 10.5 to vertical offset 20) is not covered by the gradient and because of that it is painted in a solid colour (the colour used in colour stop at offset 0). The bottom part of the rectangle (from vertical offset 100 to 110.5) is not covered by the gradient and because of that it is painted in a solid colour (the colour used in colour stop at offset 1).

The second rectangle is positioned higher (it starts at vertical offset 0.5). Parts from vertical offset 0.5 to offset 20 are all painted in solid colour. Similar goes for the third rectangle which is positioned lower on the screen and which is painted in solid colour from vertical offset 100 to offset 120.5.

Circular gradients works similarly to linear except that the colour is interpolated from one circle to another. Each circle is defined by its centre and radius. If you want the gradient to start (or end) in a single point, set the radius to \emptyset .

To prevent colours to be painted outside of the outermost circle, end the gradient with a transparent colour by passing additional parameter (transparency) to the AddCollorStop method. Code fragment below draws two circular gradients (shown in the picture above) – first one is not 'terminated' with a transparent colour and second one is.

```
circGrad := Canvas.CreateRadialGradientF(60, 300, 25, 110, 350, 100);
1
      circGrad.AddColorStop(0, 255, 0, 0);
2
      circGrad.AddColorStop(1, 0, 255, 0);
3
      Canvas.UseGradient(circGrad);
4
      Canvas.FillRectF(10, 250, 200, 200);
5
6
      circGrad := Canvas.CreateRadialGradientF(300, 300, 25, 350, 350, 100);
7
      circGrad.AddColorStop(0, 255, 0, 0);
8
      circGrad.AddColorStop(0.99, 0, 255, 0);
9
      circGrad.AddColorStop(1, 0, 255, 0, 0);
10
```

```
Canvas.UseGradient(circGrad);
Canvas.FillRectF(250, 250, 200, 200);
```



There's a bug in the CreateRadialGradientF implementation which prevents radial gradients from being created. To fix it, open the w3graphics unit (click on the w3graphics in the uses list of the unit and press Ctrl+Enter). Find the CreateRadialGradientF function. In the line where FDC.createRadialGradient is called, add parameter r0 after y0. Save, close and recompile and radial gradients should work.

```
function TW3Canvas.CreateRadialGradientF(x0,y0,r0,
 2
      x1,y1,r1:Float):TW3CanvasGradient;
 3
 4
      mTemp: THandle;
 5
    begin
 6
      try
 7
           mTemp:=FDC.createRadialGradient(x0,y0,x1,y1,r1); //original code
 8
        mTemp:=FDC.createRadialGradient(x0,y0,r0,x1,y1,r1); //correction
 9
      except
10
        on e: exception do
        Raise Exception.Create('Failed to create gradient object:' +
11
13
      end;
14
      result:=TW3CanvasGradient.Create(mTemp);
15 end;
```

4.11.8 Image

HTML images can be painted on the canvas by calling the DrawImageF method. There is, however, a catch you have to be aware of. An image is not available immediately after the LoadFromURL method is called; you have to wait until the OnLoad event is triggered.



Section Smart Controls, Image describes the TW3Image component.

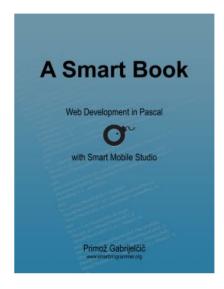
The PaintImage method from the sample code demonstrates this approach.

```
procedure TPaintForm.PaintImage;
1
    var
2
3
      image = TW3Image.Create(nil);
      image2 = TW3Image.Create(nil);
 4
    begin
5
      ClearCanvas;
6
7
      image.OnLoad :=
        procedure (Sender: TObject)
8
9
        begin
          Canvas.DrawImageF(image.Handle, 10, 10);
10
          Canvas.DrawImageF(image.Handle, 300, 10, 100, 100);
11
          Canvas.DrawImageF(image.Handle, 50, 30, 80, 100, 300, 130, 120, 150);
12
13
      image.LoadFromURL('res\title_page_small.jpg');
14
      image2.OnLoad :=
15
16
        procedure (Sender: TObject)
17
          Canvas.DrawImageF(image2.Handle, 300, 300, 125, 162);
18
19
20
      image2.LoadFromURL(COTLBook);
    end;
21
```

First image is loaded from the *res* subfolder. After it is loaded, its content is painted onto the canvas trice.

The first call to DrawImageF simply paints image with top-left corner positioned at point (10,10). The second call positions image at (300,10) and changes width and height to 100. As you can see on the picture below, this distorts the image proportions.

The third call demonstrates the most complicated option – displaying just a part of the original image. First four parameters (left, top, width, height) specify the rectangle from the original image that is to be displayed. Next four parameters (left, top, width, height) specify the rectangle on the canvas where the image will be displayed. Again, the image can be enlarged or reduced (with or without keeping proportions) during the process.









Displaying images

The code fragment above also shows how to load an image from a resource stored directly in the program code. In this example, the resource was created by using the *Tools, Image Converter* menu and is stored in the *PaintResources* unit.

Another possibility you have is to take part of the canvas and convert it into a TW3ImageData object by calling the GetImageData function which takes four parameters defining a rectangle (left, top, width, height). You can then paint this object at another coordinate by calling the PutImageData method. There's also an alternative PutImageData overload which takes another four parameters (left, top, width, height) defining the part of the original image data to display.

```
procedure TPaintForm.PaintImageData;
1
 2
      image = TW3Image.Create(nil);
3
      line: TW3ImageData;
 4
5
      x,y: integer;
    begin
6
7
      ClearCanvas;
8
      image.OnLoad :=
        procedure (Sender: TObject)
9
10
        var
          id: TW3ImageData;
11
```

```
12
        begin
          Canvas.DrawImageF(image.Handle, 10, 10);
13
14
          id := Canvas.GetImageData(50, 30, 120, 160);
          Canvas.PutImageData(id, 300, 10);
15
        end:
16
      image.LoadFromURL(COTLBook);
17
      line := Canvas.CreateImageData(Canvas.Context.Width, 1);
18
      for x := 0 to line.Width - 1 do
19
20
        line.SetPixelEx(x, 0, x mod 256, 128, 128, 255);
21
      for y := 400 to 450 do
        Canvas.PutImageData(line, y-425, y);
22
23
   end;
```

The TW3ImageData object can also be created 'from scratch' (without any content) by calling the CreateImageData method. You can then use the SetPixel to colour pixels inside the TW3ImageData and paint the object on the canvas with 'PutImageData.

Smart contains two RTL units related to images and canvas. *W3bufferedImage* is an implementation of an 'off-screen' (invisible) image containing its own canvas and *w3spriteSheet* is an implementation of a *sprite sheet* image (an image containing many small images), a concept particularly useful in writing games.

Additional reading:

- How do i move graphics from the canvas to an image?^a
- How do i plot pixels on a offscreen bitmap?
- Rethinking navigation, random rant from SMS development

ahttp://smartmobilestudio.com/2012/01/10/how-do-i-move-graphics-from-the-canvas-to-an-image/

^bhttp://smartmobilestudio.com/2012/01/06/how-do-i-plot-pixels-on-a-offscreen-bitmap/

 $^{\rm q} http://smartmobilestudio.com/2012/08/09/rethinking-navigation-random-rant-from-sms-development/2012/08/09/rethinking-navigation-random-rant-from-sms-development/2012/08/09/rethinking-navigation-random-rant-from-sms-development/2012/08/09/rethinking-navigation-random-rant-from-sms-development/2012/08/09/rethinking-navigation-random-rant-from-sms-development/2012/08/09/rethinking-navigation-random-rant-from-sms-development/2012/08/09/rethinking-navigation-random-rant-from-sms-development/2012/08/09/rethinking-navigation-random-rant-from-sms-development/2012/08/09/rethinking-navigation-random-rant-from-sms-development/2012/08/09/rethinking-navigation-random-rant-from-sms-development/2012/08/09/rethinking-navigation-random-rant-from-sms-development/2012/08/09/rethinking-navigation-random-rant-from-sms-development/2012/08/09/rethinking-navigation-random-$

4.11.9 Text

The last canvas element we'll be covering is a text. Canvas supports drawing Unicode text in any installed font. You can draw just an outline (StrokeTextF), fill text with the fill style (FillTextF) or combine both.

Before calling StrokeTextF or FillTextF you have to set the Font property which specifies font typeface (or family), size and style. Some examples are in the code below; see the Mozilla documentation¹⁰ for more examples.

¹⁰http://developer.mozilla.org/en-US/docs/CSS/font

The code fragment below shows three different ways to paint a text. The code displays an outline of the first text ('Stroked'), fills the second text ('Filled') and combines both approaches to display filled text with an outline ('Combined'). In all examples, StrokeTextF and FillTextF take four parameters – the text to be displayed, text coordinates and maximum width the text output can use. Height cannot be limited.

```
Canvas.FillStyle := '#FF0000';
1
      Canvas.StrokeStyle := '#000000';
2
      Canvas.Font := '24pt Helvetica';
3
      Canvas.StrokeTextF('Stroked', 10, 40, Canvas.Context.Width - 20);
4
      Canvas.Font := '30pt Times';
5
      Canvas.FillTextF('Filled', 10, 100, Canvas.Context.Width - 20);
6
      Canvas.Font := 'bold 36pt monospace';
7
      oldWidth := Canvas.LineWidth;
8
      Canvas.LineWidth := 2.5;
9
      Canvas.FillTextF('Combined', 10, 160, Canvas.Context.Width - 20);
10
      Canvas.StrokeTextF('Combined', 10, 160, Canvas.Context.Width - 20);
11
      Canvas.LineWidth := oldWidth;
12
```

The meaning of the text output coordinates is defined by two properties, TextAlign and TextBaseline. The former specifies how the text will be aligned relative to the starting point in the horizontal direction and the latter specifies how the text is positioned relative to the starting point in the vertical direction. Check the Mozilla documentation¹¹ to learn about the possible values and their meaning.

```
Canvas.Font := '24pt Helvetica';
1
      Canvas.FillStyle := '#000000';
2
      Canvas.TextAlign := 'left';
3
      Canvas.FillTextF('left-aligned', 200, 240, Canvas.Context.Width);
4
5
      Canvas.TextAlign := 'centre';
      Canvas.FillTextF('center-aligned', 200, 300, Canvas.Context.Width);
6
      Canvas.TextAlign := 'right';
7
      Canvas.FillTextF('right-aligned', 200, 360, Canvas.Context.Width);
8
      Canvas.TextAlign := 'left';
9
      Canvas.StrokeStyle := '#FF0000';
10
11
      Canvas.BeginPath;
      Canvas.MoveToF(200.5, 220.5); Canvas.LineToF(200.5, 359.5);
12
13
      Canvas.Stroke;
14
      Canvas.FillTextF('left:', 100, 420, Canvas.Context.Width);
15
16
      Canvas.FillTextF('right', 100 + Canvas.MeasureText('left:').Width, 420,
```

¹¹http://developer.mozilla.org/en-US/docs/Drawing_text_using_a_canvas

It is also possible to calculate the width of a text by using the MeasureText function. Unfortunately, it only returns the width, not the height.



Filled

Combined

left-aligned center-aligned right-aligned left:right

Displaying text

4.12 Regular Expressions

In Smart Mobile Studio, regular expressions are implemented in the w3regex unit so you start by adding w3regex to the uses list.

The simplified JavaScript syntax for specifying regular expressions (var re = $/\w+\s/g$) is not supported. You have to create a regular expression object with the standard Object Pascal syntax re := TW3RegEx.Create('\w+\s', 'g');. The w3regex unit does, however, implement many wrappers which simplify the regular expression usage.

4.12.1 Example

Let's suppose you've found a short JavaScript code example and you want to convert it to Smart.

```
var re = /\w+\s/g;
var str = "fee fi fo fum";
var myArray = str.match(re);
```

A direct translation of this code would be

```
var re := TW3Regex.Create('\w+\s', 'g');
var str := 'fee fi fo fum';
var myArray := str.Match(re);
```

You could write this code segment in all kinds of ways. You could, for example, make it into a nicely formatted Pascal.

```
var
re: TW3Regex;
str: string;
myArray: TStrArray;

re := TW3Regex.Create('\w+\s', 'g');
str := 'fee fi fo fum';
myArray := str.Match(re);
```

On the other hand, you can also change this into a one-liner. (Parenthesis' around the string literal are required in this case.)

```
var myArray := ('fee fi fo fum').Match('\w+\s', 'g');
```

4.12.2 A Look Inside

A short overview of the w3regex unit will help you understand how JavaScript regular expression engine is wrapped in Smart and how you can use it in your programs.

The w3regex unit implements three classes - JRegExp, TW3RegEx and TW3StringRegex.

The JRegExp class is just a Pascal representation of the native JavaScript RegExp class. The code uses class external syntax to declare a class, which has no implementation in the Pascal code. This class is just the alias for the native JavaScript class - it just allows our code to refer to the native implementation in a type safe manner.

The TW3RegEx class does three things. Firstly, it holds a reference to the JRegExp instance. When you call TW3RegEx.Create, a new JRegExp instance is created. Secondly, it implements functions (such as Exec, Test, Match ...) which call appropriate JavaScript function. Thirdly, it implements class functions (with same names as the 'normal' functions - Exec, Test ...). Those class functions allow the programmer to construct a regular expression object and immediately use it for regular expression matching. Before the class function returns a result, internally created regular expression object is discarded.

The example above could be rewritten to use the class version of the Match function.

```
var myArray: TStrArray = TW3RegEx.Match('fee fi fo fum', '\w+\s', 'g');
```

Class functions are fine if you are doing only one matching with a specific regular expression. If you would, on the other hand, like to match a same regular expression against many strings, you should create regular expression object in front.

The first example below will execute faster and use less memory than the second example.

```
// example 1, using instance functions
2
   re := TW3Regex.Create('\w+\s', 'g');
   myArray1: TStrArray = re.Match('fee fi fo fum');
4
   myArray2: TStrArray = re.Match('apples and oranges');
5
   myArray3: TStrArray = re.Match('this actually works!');
7
   // example 2, using class functions
9
   myArray1: TStrArray = TW3Reqex.Match('fee fi fo fum', '\w+\s', 'q');
10
   myArray2: TStrArray = TW3Regex.Match('apples and oranges', '\w+\s', 'g');
11
   myArray3: TStrArray = TW3Regex.Match('this actually works!', '\w+\s', 'g');
```

The TW3StringRegex is a 'helper' for the built-in string type. It defines regular expression operations on a string type so that you can write s. Match('[A-Z]*') instead of TW3RegEx. Match(s, '[A-Z]*'). If you are applying regular expression function to a string literal, this literal must be wrapped in parenthesis (('abcdef'). Match('[A-Z]*', 'i')).

4.12.3 Creating a Regular Expression Object

To create a regular expression object, call TW3Regex.Create and pass it a regular expression pattern and an optional string containing modifiers.

!!!Insert a short overview of the JS regular expression syntax

Modifiers are used to perform case-insensitive, global and multiline searches.

Modifier	Description
i	Perform case-insensitive matching.
g	Perform a global match (all possible matches will be found, not just the
	first).
m	Perform a multiline matching.

Please note that in the w3regex unit, modifiers are called flags.

4.12.4 Match

In JavaScript, match function is part of the string object. The w3regex unit implements few different Match overloads in both TW3RegEx and TW3StringRegex classes.

```
TW3RegEx = class
1
      function Match(s: string): TStrArray; overload;
2
      function Match(s: string; var idx: integer): TStrArray; overload;
3
      class function Match(s, regularExpression: string; flags: string = ''):
 4
        TStrArray; overload;
5
      class function Match(s: string; var idx: integer;
6
        regularExpression: string; flags: string = ''): TStrArray; overload;
7
   end;
8
   TW3StringRegex = helper for string
10
      function Match(regEx: TW3RegEx): TStrArray; overload;
11
      function Match(regEx: TW3RegEx; var idx: integer): TStrArray; overload;
12
      function Match(regEx: string): TStrArray; overload;
13
      function Match(regEx: string; var idx: integer): TStrArray; overload;
14
```

```
function Match(regEx: string; flags: string): TStrArray; overload;
function Match(regEx: string; flags: string; var idx: integer):
TStrArray; overload;
end;
```

Match will try to find all matches of a regular expression in the string and will return array containing all matches (if you have specified the g modifier) or only the first match (if you haven't specify the g modifier). If you have passed an integer idx parameter to the function, it will contain the position of the first match, but only if g modifier was not specified.

Example:

```
1 regEx := TW3RegEx.Create('ab+?c', 'g');
2 myArr := regEx.Match('ac_abc_abbc');
```

This code will create an array with two elements.

```
1  myArr[0] = 'abc'
2  myArr[1] = 'abbc'

Example:
1  s := '_ab_abc';
2  myArr := s.Match('ab+?c', idx);
```

This code will create an array with one element - myArr[0] = 'abc'. It will return value 4 in the idx parameter.

4.12.5 Search

In JavaScript, search function is part of the string object. The w3regex unit implements few different Search overloads in both TW3RegEx and TW3StringRegex classes.

```
TW3RegEx = class
1
   public
      function Search(s: string): integer; overload;
      class function Search(s, regularExpression: string; flags: string = ''):
4
        integer; overload;
5
   end;
6
   TW3StringRegex = helper for string
8
9
      function Search(regEx: TW3RegEx): integer; overload;
      function Search(regEx: string; flags: string = ''): integer; overload;
10
    end;
```

The Search function searches the string for a regular expression and returns position of the match or -1 if match is not found.

Example:

```
regEx := TW3RegEx.Create('ab+?c');
pos := regEx.Search('ac_abc_abbc');
```

This code will set the pos variable to 3.

4.12.6 Replace

In JavaScript, replace function is part of the string object. The w3regex unit implements few different Replace overloads in both TW3RegEx and TW3StringRegex classes.

```
TW3ReqEx = class
1
      function Replace(s, replace: string): string; overload;
2
      class function Replace(s, replace, regularExpression: string;
        flags: string = ''): string; overload;
4
5
   end;
6
    TW3StringRegex = helper for string
      function Replace(regEx: TW3RegEx; replace: string): string; overload;
8
      function Replace(regEx: string; replace: string): string; overload;
9
      function Replace(regEx, replace: string; flags: string): string;
10
        overload;
11
12
   end;
```

The Replace function searches the string for a regular expression and replaces matches with the replace parameter. If the g modifier is not specified, only the first match is replaced.

Example:

```
s1 := regEx.Replace('Apples and apples', 'oranges', 'apples', 'gi');
2 s2 := regEx.Replace('Apples and apples', 'oranges', 'apples');
```

This code will set s1 to 'oranges and oranges' and s2 to 'Apples and oranges'.

If the regular expression uses *grouping*, the replacement string can contain references to those groups. To create a group, wrap it in parenthesis. To refer to a group, use the \$n syntax where n is an integer number. \$1 refers to the first group in the regular expression, \$2 to the second one and so on.

Example:

```
1 regEx := TW3RegEx.Create('(\w+)\s(\w+)');
2 s := regEx.Replace('John Smith', '$2, $1');
```

This code will set s to 'Smith, John'.

4.12.7 Split

In JavaScript, split function is part of the string object. The w3regex unit implements few different Split overloads in both TW3RegEx and TW3StringRegex classes.

```
TW3RegEx = class
1
      function Split(s: string): TStrArray; overload;
2
      function Split(s: string; limit: integer): TStrArray; overload;
3
      class function Split(s, regularExpression: string; flags: string = ''):
        TStrArray; overload;
5
      class function Split(s: string; limit: integer;
6
        regularExpression: string; flags: string = ''): TStrArray; overload;
7
8
   end;
9
   TW3StringRegex = helper for string
10
      function Split(regEx: TW3RegEx): TStrArray; overload;
11
      function Split(regEx: string; flags: string = ''): TStrArray; overload;
12
      function Split(regEx: TW3RegEx; limit: integer): TStrArray; overload;
13
      function Split(regEx: string; limit: integer): TStrArray; overload;
14
      function Split(regEx: string; flags: string; limit: integer): TStrArray;
15
        overload;
16
17
   end;
```

The Split function splits a string into an array of substrings. The regular expression defines places where the string is split. You can also specify a *limit* - maximum number of elements to be returned. Additional elements after the split limit are ignored and will not be included in the returned array.

Example:

```
1 myArr := ('1 22 3').Split('\s+');
```

This code will create an array with three elements.

```
1 myArr[0] = '1'
2 myArr[1] = '22'
3 myArr[2] = '3'

Example:
1 myArr := ('1 22 3').Split('\s+', 2);
```

This code will create an array with two elements.

```
1 myArr[0] = '1'
2 myArr[1] = '22'
```

4.12.8 Exec

In JavaScript, 'exec' method is part of the RegExp object. The w3regex unit implements few different Exec overloads in the TW3RegEx class.

```
1 TW3RegEx = class
2    function Exec(s: string): TStrArray; overload;
3    class function Exec(s, regularExpression: string; flags: string = ''):
4    TStrArray; overload;
5    end;
```

The Exec method tries to match the string against the regular expression. It returns the matched text if it finds a match, otherwise it returns an empty array.

If a match is found, Exec returns matched text in the array element with index Ø. Array elements 1 and above will be set to the values that matched groups in the regular expression (if there were any).

Example:

```
1 myArr := TW3RegEx.Exec('ac_abc_abbc', '(a(b+?)c)');
```

This code will create an array with three elements - myArr[0] = 'abc' (matched value), myArr[1] = 'abc' (value of the first, outer group), myArr[2] = 'b' (value of the second, inner group).

If the g modifier is specified, same regular expression can be matched repeatedly against the string and will return all occurences of the matched expression.

```
1 regEx := TW3RegEx.Create('(a(b+?)c)', 'g');
2 myArr1 := regEx.Exec('ac_abc_abbc');
3 myArr2 := regEx.Exec('ac_abc_abbc');
```

This code will create following arrays.

```
1 myArr1 = ['abc', 'abc', 'b']
2 myArr2 = ['abbc', 'abbc', 'bb']
```

4.12.9 Test

In JavaScript, 'test' method is part of the RegExp object. The w3regex unit implements few different Test overloads in the TW3RegEx class.

```
1  TW3RegEx = class
2  function Test(s: string): boolean; overload;
3  class function Test(s, regularExpression: string; flags: string = ''):
4  boolean; overload;
5  end;
```

The Test function checks whether the string matches the regular expression. It returns True if it finds the match.

Example:

```
regEx := TW3RegEx.Create('ab+?c');
t1 := regEx.Test('ac_abc_abbc');
t2 := regEx.Test('acb');
```

This code will set t1 to True and t2 to False.

4.12.10 RegExDemo Example

A regular expression demo is included in the Smart distribution as the RegExDemo project. When you run it, you'll see the output of various regular expression functions. The source for the demo will show how those values were produced.

```
Console Application

[01] 2; 0: abc, 1: abbc
[02] 0;
[03] 1; 0: abc
[04] -1
[05] 1; 0: abc
[06] -1
[07] 1; 0: abc
[08] 1; 0: abc
[09] 4
[10] 1; 0: abc
[11] 1; 0: abc
[12] 2; 0: abc, 1: abbc
[13] 1; 0: abc
[14] 4
[15] 3
[16] 3
[17] -1
[18] 3
[19] 3
[20] 3
[21] 1
[22] 0
[23] 1
[24] Smith, John
[25] JohnSmith
[26] Smith, John
[27] Smith, John
[28] oranges and oranges
[30] 3; 0: 1, 1: 2, 2: 3
[31] 2; 0: 1, 1: 2
[32] 3; 0: 1, 1: 2, 2: 3
[33] 2; 0: 1, 1: 2, 2: 3
[34] 3; 0: 1, 1: 2, 2: 3
[35] 3; 0: 1, 1: 2, 2: 3
[36] 3; 0: abc, 1: abc, 2: b
[38] 3; 0: abc, 1: abc, 2: b
[38] 3; 0: abc, 1: abc, 2: b
[38] 3; 0: abc, 1: abc, 2: b
```