

# **WebHare Enterprise/Professional/Lite**

## HareScript Manual

Date: June 24th 2003  
Number of pages: 30  
Version: 0.1  
Target audience: Template developers  
Module developers

## Table of Contents

---

1.	Introduction .....	1
1.1	What you should already know.....	1
1.2	Prerequisites.....	1
1.3	Manual content .....	1
2.	Essential background information.....	2
2.1	Goals .....	2
2.2	Input and Output.....	2
2.3	Philosophy: Dynamic content creation, static content hosting .....	2
2.4	The WebHare Database.....	3
	Basic database principles and terminology .....	3
	WebHare database design .....	3
2.5	WebHare's conversion routine .....	4
2.6	Conversion Profiles.....	4
2.7	HareScript Templates and Libraries .....	5
3.	HareScript Basics .....	8
3.1	Goals .....	8
3.2	Notation .....	8
3.3	Embedding HareScript .....	8
3.4	Using comments .....	9
	Syntax .....	10
	HareScript commenting standards .....	10
	Commenting rules of thumb.....	11
3.5	Types and Variables .....	12
	The String type.....	12
	The Integer type.....	13
	The Boolean type.....	14
	The DateTime type .....	14
	The Record type .....	14
	The Array type .....	15
3.6	String encoding.....	16
	HTML encoding - The EncodeHTML function .....	17
	URL Encoding - The EncodeURL function .....	17
	Value encoding - The EncodeValue function .....	17
3.7	Modularity .....	17
	Loadlib syntax .....	18
	Public and private identifiers.....	19
	Exporting identifiers .....	20
4.	Working with the WebHare database .....	22
4.1	Goals .....	22
4.2	The records in the Folders table.....	22
4.3	The records in the Files table .....	23
4.4	The records in the Pages table.....	23
4.5	The records in the DocObjects table .....	24
4.6	Retrieving records from the database .....	26
	The select statement .....	26
	The forevery statement.....	28

## 1. Introduction

---

With the release of the second version of WebHare® the HareScript® programming language has been completely renewed and updated, giving you even more power and flexibility in creating WebHare templates and modules.

This manual will explain in detail how to create HareScript templates that can be used in both WebHare CMS (WebHare Professional and WebHare Enterprise) and WebHare Lite.

It can be used by anyone who wants to develop powerful WebHare templates. The manual is primarily intended for use by novice HareScript developers, but can also be useful for advanced scripters.

### 1.1 What you should already know

For optimal use of this manual you should have:

- Good working knowledge of WebHare CMS or WebHare Lite.
- Good working knowledge of the World Wide Web
- Good working knowledge of the HTML mark-up language

Previous experience with scripting languages like JavaScript, PHP or ASP and some knowledge of relational databases and the Structured Query Language (SQL) can be useful, but is not necessary to successfully use this manual.

### 1.2 Prerequisites

Although the theory and examples used in this manual can be used in both WebHare CMS and WebHare Lite, for practical reasons it is presumed you have a working version of WebHare Lite at your disposal. You can download (ADDME:Link) a 60 day trial version of WebHare Lite from our website(ADDME:Link).

The manual will also at times refer to the HareScript Reference. This reference contains information about all additional HareScript elements like macros, functions and variable types, that are not explained in this manual. When using this manual it can be useful to have a copy of the HareScript Reference at hand.

All example templates and libraries used in this manual are available for download at the WebHare support site (<http://www.webhare.net/>). Here you can also find the latest versions of all WebHare CMS/Lite user manuals, and the HareScript Reference.

### 1.3 Manual content

Every chapter in this manual starts with an overview of the learning goals of the specific chapter. After studying the chapter, you should be able to successfully meet these goals.

Every chapter provides theoretical information about the subjects discussed, and real-life examples for you to try out yourself.

Most chapters are concluded with additional reference information.

Chapters 2-4 provide mostly theoretical background information about the HareScript syntax, the most commonly used variable types, functions and macros, and information about the WebHare database structure. It is highly recommended to thoroughly read these chapters before creating your first template.

In chapters 5 you will start working on your first template. Chapters 6-ADDME will guide you step-by-step through the process of creating a template and HareScript libraries, cumulating in a WebHare template you can use to create fully functional websites.

## 2. Essential background information

---

In this chapter some background information about WebHare and the HareScript programming language is provided. This manual will frequently make use of the terms and concepts explained in this chapter, so it is recommended you read this chapter attentively.

### 2.1 Goals

After reading this chapter you will know:

- What is meant by *input* and *output*.
- What is meant by *dynamic* creation of *static* content
- What a database is, and how the WebHare database is designed.
- What HareScript templates and libraries are.
- How WebHare converts Microsoft® Word documents.

### 2.2 Input and Output

In WebHare a clear distinction is made between the *input* and the *output* of the system.

The direct interaction between you (the user) and WebHare is called *input*. Whenever you for example create a folder, edit a file or remove a file or folder, you are inputting information into the system.

When publishing the site, WebHare will automatically create a website based on the files and folders you have created in the WebHare File manager (the input). The publication process consists of creating a folder structure, converting the Word documents to HTML files etc. These files and folders are placed somewhere on a web server.

This collection of folders, HTML files, images and all other kinds of files is called the *output*. The web server hosting the output on the Internet is called the *output server*.

If you are using WebHare Lite, the output will first be created on your local hard disk, so you can easily preview your site. This *Local output* is created automatically. After you've decided the site is exactly the way you want it, you can choose to create the website on the web server that will host it on the Internet. This is called the *Live output*.

### 2.3 Philosophy: Dynamic content creation, static content hosting

Most Content Management Systems (CMS's) store all content in a central database. Every time a user visits the website (requests a page), the requested content is retrieved from the database, embedded in a template, and sent to the user. This means that if your website has to handle 30.000 page requests on an average day, the database has to retrieve the same information 30.000 times.

Most people would agree this isn't a very efficient process. Why should basically static content like an itinerary, that is updated ones every couple of years, be retrieved 30.000 times every day?

Connecting to a database will always be a relatively slow process that should be avoided as much as possible. But besides this efficiency problem there is yet another, even bigger problem with dynamically hosting content. Because all information is only stored in the central database, the website will not be available if the CMS crashes or has to go down for maintenance. Dynamically hosted CMSs can never guarantee 100% information availability.

In contrast to other CMSs, WebHare only dynamically hosts information that really needs to be dynamic. An overview of selected articles in a web shop can never be static, nor can a list of search results. So dynamic hosting is the only possibility for these types of content. But for content like the aforementioned itinerary, a better solution can be chosen.

Both WebHare CMS and WebHare Lite can create *static* content *dynamically*. This is how it works:

Whenever you create some kind of input like a file or folder, this input is stored in WebHare's central database. Metadata like a title, description or some keywords are also stored in this database.

When publishing a website, WebHare will read the necessary information from the database and will create static HTML pages based on this information. It will automatically create navigational structures like menus or a sitemap, and embed this information in a template. The result is a folder structure with lots of linked HTML files, that will reside somewhere on a web server or hard disk (the output).

Whenever you change content in WebHare, the output will automatically change accordingly after republishing the site.

Because WebHare dynamically creates static content, you are free to choose where the content will be made available. The same website can be made available on the Internet, the intranet or even on a CD-ROM. It doesn't matter; the choice is yours.

## 2.4 The WebHare Database

At the heart of both WebHare CMS and WebHare Lite lies a relational database. All information about your WebHare sites, folders and files is stored in this database and can be retrieved using HareScript.

Before continuing to explain how the WebHare database is designed, some basic background information about databases will be provided. If you already know the basic principles and terminology of relational databases, you can skip the next paragraph.

### Basic database principles and terminology

A relational database is best compared to a bookcase. A bookcase contains a number of shelves, each containing some books. These books have a number of properties like the colour, the number of pages and the author of the book.

In database terminology the shelves are called *tables*. The books are called *records*, and the properties of the books are called *cells*.

The following example shows an example *database table*.

ISBN	Title	Author	Colour	Pages
009887	Database design	W.A. Johnson	Green	200
786566	Database design	P.R. Smith	White	200
662213	Love and romance	P.R. Smith	Green	564

Figure 2.1: An example database table

This example database table shows three records (the rows), each having five cells (the columns).

In order for a database to retrieve information fast and efficiently, there has to be at least one unique identifying property value for every record. In this example database, the unique identifier is the ISBN number. This number is unique for every book in the world.

In most database tables this identifier is called the *ID*. Using this ID you can retrieve the appropriate record, and access it's other cells like the title and the author.

### WebHare database design

The WebHare database contains a lot of tables, of which the most commonly used are:

- The Files table. This table contains all information (like the title and description) of every file in WebHare.
- The Folders table. This table contains all information about all the folders in your WebHare site.
- The Pages table. This table contains information about the pages that are created during the conversion of a Word document by WebHare. This table is only available during the conversion process.

- The DocObjects Table. The DocObjects table contains information about every single paragraph, table and table cell in a converted Word document. This table is also only available during the conversion process.

Whenever you create a new folder, the information about this new folder is stored in the Folders table. Editing the description of a Word document means changing description cell of the Word document's record in the Files table.

If you want to find out how many HTML files will be created during the conversion of a particular Word document, this information is available in the Pages table. And if you for example want to know what the text of the second paragraph on the third converted page of a Word document is, you'll need to use the DocObjects table. In paragraph 2.6(ADDME: link) of this manual some additional information about the Pages and DocObjects tables is available.

Information from these and other tables in the WebHare database can be retrieved using HareScript, and can be used by your templates and other HareScript files.

More information about these and other tables that are available in WebHare can be found in chapter ADDME of this manual (ADDME:link), and in the HareScript Reference (ADDME: link).

## 2.5 WebHare's conversion routine

WebHare uses a unique and extremely powerful conversion engine that has been designed with flexibility in mind. Other applications that offer a Word to HTML conversion option use the standard conversion engine provided by Microsoft™. WebHare does not.

The main difference between this 'standard' conversion and the WebHare engine is in the way Word documents are treated.

Most conversion routines look at a Word document as being a linear chain of paragraphs. Every paragraph is a link in this chain. This means that when a Word document is converted, the conversion routine starts with the first paragraph of your document, converts it to HTML and then goes on to the next paragraph, and so on until every paragraph is converted. What goes in will come out, in exactly the same sequence. It doesn't really matter what is being converted. A paragraph is basically the same as a table or an image. They're nothing more than links in the same chain.

In contrast, WebHare's conversion routine looks at a Word document as being an object tree. In WebHare, a Word document is much like a directory tree on your hard drive. It contains lots of objects that can contain other objects, like a folder can contain files or other folders, making up a tree structure.

In WebHare a table is a different object than a paragraph, a table cell is different from a table and a paragraph in a table cell is yet another object, with other properties.

During conversion, every single one of these objects is stored in the DocObjects table. Using HareScript you can access these objects and use them to create any webpage you like.

The possibilities of this object oriented conversion approach are endless. You can for example use HareScript to analyse a table in a Word document, and decide to omit the first row of this table in the HTML file that will be created. Or you can decide to omit the table all together, and just print the paragraphs contained in the table sequentially.

## 2.6 Conversion Profiles

A lot of information about the way WebHare should convert Word documents is stored in so-called *conversion profiles*. By using these profiles you can for example set the font colour and font size to be used in the HTML pages, define paragraph spacing and indentation or tell WebHare to split up a large word document into multiple output HTML files.

As you know, every paragraph in a Word document has a particular style attached to it. The most commonly used styles are the *Heading 1*, *Heading 2* and *Normal* styles. These

styles contain information about the character and paragraph formatting used. For instance, a Heading 1 style can be defined as using font face 'Times New Roman', with font size 14 and a bold character effect.

Most people have their own personal preferences about what a particular paragraph should look like in Word. One person might like the Times New Roman font face; another prefers to use Arial or Helvetica. Although these personal differences might be of little consequence most of the time, you most likely don't want these differences reflected in you website. That's where conversion profiles come in.

Using profiles, you can override the way a particular style in your Word document should be formatted in the webpage. You can for instance tell WebHare to always use font face 'Arial' with font size 12 and text effect 'bold' for every paragraph with style Heading 1.

These and other style-specific settings guarantee consistency in the published website.

But conversion profiles can do more. You can for example set a *Table of Contents level* for every style used. This TOC level will be stored in the DocObjects table. Using HareScript you can then for automatically create a Table of Contents for every converted Word document, containing links to all level 1, level 2 and level 3 paragraphs.

An example of how to use this TOC level property is explained in detail in chapter ADDME.

## **2.7 HareScript Templates and Libraries**

WebHare can publish any type of file. For most file types, like PDF documents, Java applets or Flash animations, publication means that the file is simply copied to the output server.

Other file types, like Word documents and HTML files, are embedded in a template while being published.

In WebHare templates are used to determine the layout (design) of the published website, and to automatically create navigational structures, such as menus or sitemaps, in the site.

WebHare templates consist of HTML source code, supplemented with HareScript source code. You can use any HTML code to create the layout of your pages, and you can use any type of HTML editor such as FrontPage or Dreamweaver to create this layout. You can then embed HareScript code into this HTML code to create for example a button bar or a table of contents. The following figure shows this process.

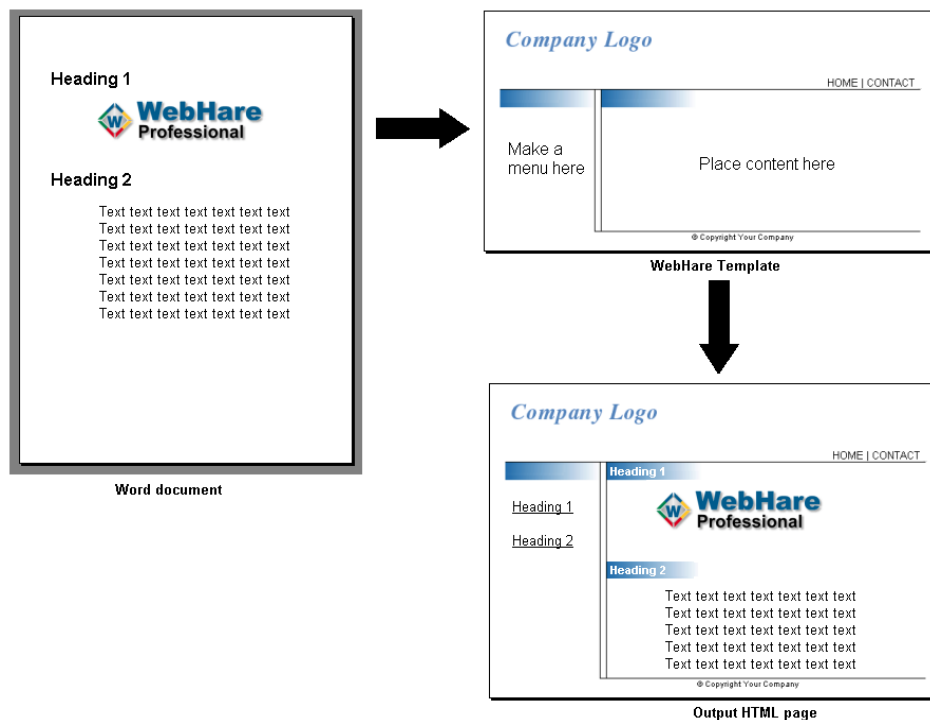


Figure 2.2: WebHare template usage

You can use templates with HareScript to automate almost everything concerning your website. You can for instance:

- Create a Table of Contents for every Word document.
- Automatically generate menus based on the structure of your site. You can even have WebHare create images for every menu item.
- Create a sitemap
- Create dropdown menus, using DHTML and JavaScript.
- Convert your Word documents to HTML, and automatically create a link to download the original Word document.
- Connect to other databases using ODBC or OCI, and use information stored in these databases in your website (WebHare CMS only)
- Automatically print the name and e-mail address of the author of the content, and show the date and time the content was last updated.

Many of these functionalities are available in so called *HareScript libraries*.

A HareScript library is a file that contains generic functions and macros for you to use in your templates. If you want to use a specific function in your template, you only have to load the library containing this function.

You can easily create your own libraries, containing all the functions, macros and variables your specific template requires. But chances are that one of the standard libraries we have provided already contains a function.

The most commonly used built-in libraries are:

- *template-v2.whlib*. This library provides, amongst others, access to all folder, file and site records. It also contains the DocObjects-and Pages tables, and the OpenFile() en CloseFile() macro's. Loading this library is mandatory for every template you create. Without it, nothing will work.
- *publisher.whlib*. The publisher.whlib library provides all functions needed to easily find a file and folder records based on their ID or name.
- *datetime.whlib*. If you for instance want to print the date a file was last updated, the datetime.whlib library is needed. It provides an interface that allows you to easily work with everything pertaining date and timestamps.

- *graphics.whlib*. This library contains functions you can use if you want to generate buttons or any other kind of image from your template.

There are many more libraries available in both WebHare Lite and WebHare CMS, which will not be explained in this manual. For more information about all available libraries, please refer to the HareScript Reference (ADDME: link). You can also surf to the WebHare support site (<http://www.webhare.net>) to check for information on new or updated libraries.

### 3. HareScript Basics

---

#### 3.1 Goals

After studying this chapter you will know:

- The HareScript notation definitions
- How to embed HareScript in HTML
- How to use comments with your HareScript code
- What types of variables are available in HareScript.
- What encoding means, and what types of encoding are available in HareScript.
- What LOADLIBs, FUNCTIONs and MACROs are, and how to use them.

#### 3.2 Notation

Every block of HareScript code needs to be opened by the *HareScript open tag*, and closed by the *HareScript close tag*.

The open tag needs to be noted as follows:

```
<?wh
```

The notation of the close tag is:

```
?>
```

The open tags must be followed by a space, and the close tag must be preceded by a space. All other characters in both the open and close tags must be in lower case. No other characters may be included in these tags.

A block of HareScript code may physically be distributed over more than one line:

```
<?wh
  Here the HareScript sourcecode
  will be placed
?>
```

A semicolon must end every line of source code:

```
<?wh
  This is the first line of HareScript code;
  This is the second line of HareScript code;
  This is the
  third line of HareScript code;
?>
```

#### 3.3 Embedding HareScript

HareScript is a HTML embedded scripting language. This means you can use HareScript source code in your normal HTML code:

```
<html>
  <head>
    <title>
      Example
    </title>
  </head>
```

```
<body>

  <?wh
    Example HareScript source code
  ?>

</body>
</html>
```

In the example above some HareScript source code is embedded in some basic HTML source code.

You can also embed HareScript code in a single HTML tag, as shown in example 3.2:

```
<a href="<?wh HareScript code ?>">My link</a>
```

During publication the HareScript source code will be executed, and the result will be available in the published HTML file.

Let's have a look at the following example code:

```
<html>
  <head>
    <title>
      Example
    </title>
  </head>
  <body>

    <?wh
      PRINT("My first WebHare template");
    ?>

  </body>
</html>
```

The result of publishing a running this HareScript code will be an HTML file containing the following source code:

```
<html>
  <head>
    <title>
      Example
    </title>
  </head>
  <body>

    My first WebHare template

  </body>
</html>
```

As you will notice, only the HareScript code in-between the open `<?wh` tag and the closing `?>` tag is executed. The rest of the code is left untouched. The `PRINT` command causes WebHare to print everything between the double quotes to the output HTML page.

### 3.4 Using comments

Every programmer will agree on the usefulness of using comments in source code. Although comments in source code are never a substitute for good documentation, they can help others to better understand your code, and the decisions you have made while developing the code.

Besides these obvious reasons for commenting there is yet another reason. WebHare (FIXME:developer releases only???) comes with a utility that can automatically generate

source code documentation for you, as long as you comply with the HareScript commenting standards (ADDME: link).

The next paragraphs will describe the basic commenting syntax, provide some information about the HareScript commenting standards, and help you with some commenting rules of thumb.

## Syntax

In HareScript there are two ways to add comments. If you want to add only one line of comment, you start your comment with two slashes:

```
// this is a line of comment
// this is another line of comment
```

To add comments that span more than one line (a *block* of comments), start the comment with `/*` and end it with `*/`, as shown in the next example:

```
/* This is
   a line of comment
   that spans
   more than one line
*/
```

## HareScript commenting standards

In WebHare a HareScript documentation generator (DocGen for short) is available to help you create documentation for all your HareScript code. This utility will analyse the comments you have added to your source code, and create documentation based on these comments.

If you want to use this utility, your comments should comply with some basic rules. These rules are described in greater detail the DocGen manual (ADDME: link). Here only the some basic rules are explained:

1. Start a block of comments with a slash followed by *two* asterisks (`/**`)

```
/**
By using a slash and two asterisks, DocGen will know it must analyse this comment
*/
```

2. For functions and macros, always specify a short description using the `@short` tag, followed by the description.

```
/**
@short This is the short description
*/
```

3. For functions and macros, always specify all parameters using the `@param` tag, followed by the name of the parameter and a short description.

```
/**
@short This is the short description
@param Parameter1 The description of the first parameter
@param Parameter2 The description of the second parameter
*/
```

4. For functions, always specify what is returned, using the `@return` tag followed by a short description

```

/**
@short This is the short description
@param Parameter1 The description of the first parameter
@param Parameter2 The description of the second parameter
@return The description of what the function returns
*/

```

### Commenting rules of thumb

There are a number of rules of thumb you might find useful when writing comments:

1. If the meaning of a line of source code is obvious to everyone, don't add comments:

```

INTEGER i := 0; // initialising i with value 0

```

This comment is not really useful, as anyone will understand the code.

2. Try to explain *why* you are doing something, not only *what* you are doing:

```

FOR (INTEGER i := 0; i < 10; ++i) // looping ten times
{
  PRINT("printing: " || i);
}

```

It's nice to know the programmer of this example code let's us know he's looping ten times, but it would be much nicer to know why he's doing this. Why not eleven times? Why is he looping at all?

3. Don't write prose. Comments are usually scanned, not read. If your code needs a lot of commenting, maybe it's better to split it up in easier to understand parts.

```

/* This macro will, when executed by our template (right now it's called
'default.tpl' but this may change in the near future due to requests from clients
asking us to change the name to something like 'mytemplate.tpl'), scan for the
availability of a file description that contains more than the needed 100
characters, a file description that only consists of 10 or less characters, or a
folder title that contains one or more capital letters, and print the ID of the
file that satisfies these conditions or, if it's a folder, print the folder ID.
*/
MACRO MyFunction()
{
  ...
}

```

This example comment is much too long for anyone to read, let alone understand its relevance.

4. Explain for every function what its purpose is, and what goes in and what comes out. You might want to use the HareScript commenting standards as explained in the previous paragraph.

```

/* This function can return the name of a file, or an empty string */
STRING FUNCTION myFunction(INTEGER FileID)
{
  RECORD TheFile := SELECT name FROM files WHERE ID = FileID;
  IF (TheFile.name != "document.doc")
    RETURN ToUpperCase(TheFile.name);
  ELSE
    RETURN "";
}

```

This example comment should be much more specific for anyone to use the function properly. It's not clear what parameters to specify when using the function, nor is it clear from the comments when a filename or an empty string is returned.

5. When writing comments for a function, try to view the function as a black box that can be used without knowing anything about the code contained in the function. The function's comments should be all that's needed to use the black box.

```
/** @short Strips leading spaces from a string
    @param stripme The string to strip leading spaces from
    @return A string stripped of all it's leading spaces
 */
STRING Function StripSpaces(STRING stripme)
{
    ...
}
```

Even though the exact code of this function isn't available, you can use the function anyway, as you know exactly what you can expect from it.

### 3.5 Types and Variables

A variable is a container for information (*data*). In HareScript, every variable must have a name and an initial value, and the type of data stored in the variable must be defined.

The *name* of the variable is used to reference the variable. A variable name is *case insensitive*, meaning it may contain both uppercase and lowercase characters. In HareScript a variable name may not contain spaces, nor may it start with a number.

For every variable you define in your HareScript code you must also specify the *type of data* it will contain.

HareScript supports many different variable types. For a complete overview of all supported types, please refer to the HareScript Reference (ADDME: link).

In this chapter the following commonly used types are explained:

- The *string* type, used to store characters
- The *integer* type. Integers are used to store whole numbers
- The *boolean* type, used to store truth values (TRUE or FALSE)
- The *datetime* type. A datetime variable can store date and time values.
- The *record* type. A record is a special type of variable, used as a container for other variables
- The *array* type. An array is a list of variables of the same type.

You cannot assign a value of one type to a variable of another type. For instance you cannot store a character in an integer, or a number in a boolean.

The general notation for assigning a value to a variable is:

```
<data type> <variable name> := <value>
```

#### The String type

A string is a set of characters. In HareScript a string may contain at maximum 4096 bytes.

Strings are UTF-8 encoded. With UTF-8 encoding, most characters are stored using only one byte. Some special characters however, like Å, Æ and Ø, are multi-byte characters, meaning they use more than one character position in a string. This means that 4096 characters will not always fit in a string, if some of these characters are multi-byte characters.

If a string is declared without setting an initial value, the string will be empty by default.

Examples of string declarations are:

```
STRING myString1 := "Hello world";
STRING myString2; // no initial value is given, so the string is empty by default
```

There are a lot of ways to manipulate strings in HareScript, amongst others:

- The *Merge operator* ( `||` ), used to merge two strings.
- The *Left* function, which returns the first x characters of a string.
- The *Right* function, used to return the last x characters of a string.
- The *ToString* function, which converts an integer to a string.
- The *SubString* function, which returns characters x to y from a string.

The following example code shows how to manipulate strings in HareScript:

```
// Merge two strings, returning 'Hello Worlds'
STRING MyString3 := "Hello " || "World";

// Returns 'Hello', the first 5 characters of string 'Hello World'
STRING MyString4 := Left("Hello World",5);

// Returns 'World', the last 5 characters of string 'Hello World'
STRING MyString5 := Right("Hello World",5);

// Convert an integer to a string
INTEGER i := 100;
PRINT('<table width=' || ToString(i) || '>');

// get characters 5 to 10
STRING MyString6 := SubString("This is my string",5,10);
```

For more information on manipulating strings in HareScript, please refer to the [HareScript Reference \(ADDME:link\)](#)

### The Integer type

An integer is a number of the set  $Z = \{\dots, -2, -1, 0, 1, 2, \dots\}$ .

Integer values in HareScript must be in the range from  $-2^{31}$  (about -2.000.000.000), to  $2^{31}-1$  (about 2.000.000.000).

If no initial value is specified when declaring an integer, a default value of 0 (zero) is set.

Examples of integer declarations are:

```
INTEGER myInteger1 := 12;
INTEGER MyInteger2 := -99;
INTEGER myInteger3; // no initial value was set, so the integer value defaults to 0
```

HareScript offers a lot of functions and operators you can use to manipulate integers. The most commonly used are:

- The `+` and `-` operators, used to add and subtract integers.
- The `*` and `/` operators, used to multiply and divide integers
- The *ToInteger* function, to convert strings to integers

The following code example shows these operators:

```
// Integer declaration
INTEGER i := 10;
INTEGER j := 5;

INTEGER Total := i + j; // adding 10 + 5 = 15
INTEGER Difference := i - j; // subtracting 10 - 5 = 5
INTEGER Division := i / j; // dividing 10 / 5 = 2
INTEGER Multiple := i * j; // multiplying 10 * 5 = 50
```

```
// printing integer 'multiple'
PRINT(ToString(Multiple));
```

For more information on manipulating integer in HareScript, please look in the HareScript Reference (ADDME:link)

### The Boolean type

Booleans are used to store a truth-value. Booleans can only have one of two values: TRUE or FALSE.

If no initial value is specified when declaring a boolean variable, a default value of FALSE is used.

Examples of boolean declarations in HareScript:

```
BOOLEAN myBoolean1 := TRUE;
BOOLEAN myBoolean2; // no initial value was specified. The value defaults to FALSE
BOOLEAN myBoolean3 := 1 + 1 = 3; // 1 + 1 does not equal 3, so myBoolean3 is FALSE
```

### The DateTime type

Whenever you want to use a time or date in your HareScript code, you have to use the DateTime variable type.

DateTime variables are somewhat special, because you cannot set an initial value for this type of variable without the help of some built-in HareScript functions.

The HareScript datetime.whilib library offers many functions and macros you can use to manipulate dates and timestamps. Some examples:

```
// store the current date and time on the server
DATETIME MyDateTime1 := GetCurrentDateTime();

// Make a timestamp for the 25th of march 2002, at 11:56 AM
DATETIME MyDateTime2 := MakeDateTime(2003,3,35,11,56,0);
```

If you declare a DateTime variable without specifying an initial value, the value is set to *Invalid* by default. You can use the `IsDateValid()` function to check if a date is valid.

```
DATETIME MyDateTime3; // No initial value was specified, default value is Invalid

// The IsDateValid function is used to check if the timestamp is valid
BOOLEAN TheDateIsValid := IsDateValid(MyDateTime3); // FALSE is stored
```

For more information on all available functions and macros you can use when working with dates and timestamps, please refer to the HareScript Reference ADDME:Link).

### The Record type

The record variable has already been briefly introduced in paragraph 2.4 (ADDME:link) of this manual. A book metaphor was used to describe the record variable. Every book has properties (a colour, an ISBN number, the author etc.), which are the *cells* of the record.

Let's have a closer look at this metaphor. Let's assume we have a book called 'database design', written by John Smith, that has 239 pages. When translated to database terminology this would be a record with a title cell, an author cell and a pages cell, as shown in the following table:

Title	Author	Pages
Database design	John Smith	239

When you look at this example record, you will notice that the title and author cells are *strings*, and the pages cell is an *integer*.

Now let's assume we also want to store information about whether we've read the book or not. This would mean we have to have another cell of type *boolean*:

Title	Author	Pages	Read it?
Database design	John Smith	239	TRUE

As you have noticed, a record is different from any other variable in that it functions as a *container for data of various types*.

You can declare a record variable by using either a SELECT command, or by using a built-in function that returns a record. The following examples show how:

```
// Get the record of the file with ID = 17, using a SELECT command
RECORD myFile := SELECT * FROM FILES WHERE ID = 17;

// Get the record using the built-in FindFileByID() function
RECORD myFile := FindFileByID(17);
```

If you declare a record variable without specifying an initial value, the record will be a *non-existing* record. This means the record does not contain any data or cells.

There are many other ways to manipulate a record in HareScript. You can for instance update the value of a cell in a record, add or delete a cell or check whether a cell exists. These and other record manipulation functions are described in detail in the HareScript Reference (ADDME: link).

### The Array type

An array is a list of variables of the same type. When declaring an array, you *must* specify the type of variables you want to store in this list:

```
STRING ARRAY myStrings; // used to store strings
INTEGER ARRAY myIntegers; // used to store integers
RECORD ARRAY myRecords; // used to store records
```

You cannot store variables of a one type in an array of another type. For example you cannot store an integer in a string array.

If no initial value is specified when declaring an array, the array is empty.

When working with arrays of any type, you must remember that the counting of the elements in an array starts at 0. The first element in an array is element 0, the second is element 1 and so on.

There are many ways to manipulate arrays in HareScript, depending of the type of array you are using. An exhaustive overview of all array manipulation option can be found in the HareScript Reference (ADDME: link). In the next example some often-used array manipulations are shown:

```
/* The TOKENIZE function is used to create a string array from a string. The string
array will contain 6 elements: 'a','b','c','d','e' and 'f' */
STRING theString := "a-b-c-d-e-f";
// split the string at every '-' character
STRING ARRAY myStrings := TOKENIZE(theString,"-");

/* insert a new integer into an integer array, at the end of the list */
INTEGER ARRAY myIntegers; // declaring a new integer array
INTEGER theInteger := 15; // declaring a new integer with value 15;
INSERT theInteger INTO myIntegers AT END; // inserting the integer into the array

/* Replace the value of the first element in a string array. This results in an
array with these elements: 'q','b','c','d','e' and 'f' */
string array myStrings := tokenize ("a-b-c-d-e-f","-");
myStrings[0] := "q"; //change the value of the first element to 'q'.
```

Working with arrays, especially *record arrays*, is an integral part of working with HareScript. In the next chapters more information and lots of examples on how to work with arrays is given.

### 3.6 String encoding

Every time you tell your template to print a string from an external source, like a file description or a folder title, you run the risk of printing unsafe characters to your HTML pages.

Let's have a look at the following example HareScript code:

```
<html>
  <head>
    <title>
      Example
    </title>
  </head>
  <body>

    <font color="black">
      <?wh
        PRINT(file.description);
      ?>
    </font>

  </body>
</html>
```

The exact meaning of the HareScript code in this example is not important (it will be explained in more detail in Chapter 4 [ADDME: link](#)), as long as it is clear that a user-specified file description is printed.

You would expect the result of executing this code to be a web page on which the file description is printed in *black*.

Now let's assume the user entered the following string for the file's description:

```
<font color="red">I'm printing HTML code!</font>
```

Because the description is printed exactly as it was entered, the result of executing the code would be a page with the following source code:

```
<html>
  <head>
    <title>
      Example
    </title>
  </head>
  <body>

    <font color="black">
      <font color="red">I'm printing HTML code!</font>
    </font>

  </body>
</html>
```

This result is a web page on which the file's description is printed in *red* instead of *black*, as you specified originally.

Although this is only a very basic illustration, you can probably imagine the potential problems using code like this may cause.

The solution to these kinds of problems is using the correct *encoding*. You should *always* use an encoding when printing strings from external sources.

In HareScript there are a lot of ways to encode strings. The encoding type needed depends on what you want to do with the string. The most commonly used encoding types are:

- HTML encoding, used to encode strings used outside HTML tags.
- URL encoding, for encoding the query part of a URL.
- Value encoding, used to encode attributes of html tags.

These encoding types are explained in more detail in the next paragraphs. For a complete overview of all encoding and decoding types available in HareScript, please refer to the HareScript Reference (ADDME: link).

### HTML encoding - The EncodeHTML function

The general rule for using the EncodeHTML function is that it should always be used when printing strings from an external source in-between HTML tags. The following example illustrates this rule:

```
<title><?wh PRINT(encodeHTML(file.title)); ?></title>
```

Using the EncodeHTML function ensures special characters (such as '<', '>' and '&') and multi-byte characters are converted to proper HTML entities.

For example a less-than (<) character is converted to &#60; and a greater-than character (>) is converted to &#62; when printed to an output file:

```
<?wh PRINT(EncodeHTML("<font face='Arial'>")); ?>
```

This would result in:

```
&#60;font face='Arial'&#62;
```

### URL Encoding - The EncodeURL function

The EncodeURL function should be used to encode the query part of an URL.

There are a lot of characters that are not allowed on a URL, for example a space or a single quote. These and other characters are converted so that the URL only contains valid characters.

```
// returns The_rain.%20In%20Spain%2C%20Ma%92am
encodeurl("The_rain. In Spain, Ma'am");
```

### Value encoding - The EncodeValue function

The EncodeValue function should be used when printing strings from external sources as values of HTML tag attributes. This is explained in the following example:

```
<form name="form" method="get" action="/search/results.shtml">
<input type="hidden" name="user" value="<?wh PRINT(EncodeValue(user)); ?>">
</form>
```

In this example the value of the hidden form field could potentially contain illegal characters. By using the EncodeValue function, these characters are converted, ensuring the correct value is submitted.

## 3.7 Modularity

HareScript is a modular language, meaning you can combine related source code in so-called *libraries* (see also paragraph 2.7 of this manual ADDME: link).

By using libraries you can load functions, macros and variables only if and when you need them. For example: if your template doesn't use any date or timestamps, you don't have to load the *datetime.whlib* library.

Loading a library gives you access to all *public* and *exported* functions, macros and variables (these are collectively called *identifiers*) that are available in the loaded library. In the next paragraphs detailed information about public and exported identifiers is provided. We will start by describing the syntax of loading libraries.

### Loadlib syntax

In HareScript a lot of predefined libraries are available, offering a lot of commonly used functionalities. To make these functionalities available in your own source code, you have to load the respective library, using the LOADLIB statement.

There are four ways to use the LOADLIB statement, depending on the location of the library you want to load:

If you want to load one of the pre-defined libraries that are part of the HareScript system, you must use the following syntax:

```
LOADLIB "wh::[path to library]/<library name>";
```

You can also load one of the pre-defined libraries that are available in any WebHare module. Of course, you can only load libraries from modules that are actually installed on your system. The HareScript reference has more details on the libraries available for your product. You need to use the following syntax to load module libraries:

```
LOADLIB "module::<module name>/<path to module>/<library name>";
```

To load a library from the same site as where the template is available, use this syntax:

```
LOADLIB "currentsite::<path to library>/<library name>";
```

If you want to load a library from another site (WebHare CMS only), you should use this syntax:

```
LOADLIB "site::<site name><path to library>/<library name>";
```

Some examples:

```
// Load the datetime library, that resided in the 'wh' module
LOADLIB "wh::datetime.whlib";

/* Load the HTML output control library. This library can be used to control every
aspect of the HTML pages that WebHare creates during conversion of Word documents.
It is available in the 'output' folder of the publisher module */
LOADLIB "module::publisher/output/html.whlib";

/* load a library called 'myfunctions.whlib' that is placed in a folder called 'my
libs' in the root of your current site */
LOADLIB "currentsite::/my libs/myfunctions.whlib";

/* Load a library from folder '/all libs/specific libs/' in site 'Repository' */
LOADLIB "site::repository/all libs/specific libs/myfunctions.whlib";
```

Loadlib commands *must* be the first commands in your libraries, templates and other HareScript files. Libraries don't have to be loaded in any specific order.

For more information on using the LOADLIB syntax please refer to the HareScript Reference (ADDME: link).

### Public and private identifiers

All functions, macros and variables (the *identifiers*) you define in a library can always be used by other code in the same library, but are *not* by default available to other libraries or templates that load the library.

If you want to use a specific identifier in another file, you have to mark the identifier as *public* by using the PUBLIC keyword when declaring the identifier:

```
// A public integer variable declaration
PUBLIC INTEGER i := 0;

// A public macro declaration
PUBLIC MACRO myMacro()
{
  PRINT("Hello World");
}

// A public function declaration
PUBLIC STRING FUNCTION myFunction()
{
  RETURN "Hello World";
}
```

The following example illustrates the use of public identifiers:

#### mylibrary.whlib:

```
<?wh

// declare two public integers
PUBLIC INTEGER i := 10;
PUBLIC INTEGER j := 20;

?>
```

#### mytemplate.tpl:

```
<?wh

// load the library
LOADLIB "currentsite:/mylibrary.whlib";

/* use integers i and j, that were
declared in the library */
INTEGER total := i + j;

?>
```

>>

Identifiers that are not explicitly marked as public are by default *private*. However for readability purposes, you might want to explicitly mark an identifier as *private*:

```
// The following variable declararions are functionally equivalent
INTEGER i := 0;
PRIVATE INTEGER i := 0;
```

Public identifiers are only available in the library in which they are declared, and in the file that loads the library. This means that if you declare for example a public integer in library1.whlib, and load this library from library2.whlib, the integer will be available in both library1.whlib and library2.whlib.

However if you then load library2.whlib from library3.whlib, the integer will *not* be available anymore:

#### library1.whlib

```
// integer declaration
public integer i := 10;
```

>>

#### library2.whlib

```
//i is available
j := i + 20;
```

>>

#### library3.whlib

```
/* i is not available
anymore! This will result
in an error! */
k := i + 30;
```

In the example above, if you want to be able to use integer i in library3.whlib, you have to export the integer, as described in the next paragraph.

## Exporting identifiers

As described in the previous paragraph, all identifiers that are marked public in a library will be also be available in a file that loads this library; public identifier `x` in library `lib1.whlib` will be available in `lib2.whlib`, if `lib2.whlib` loads `lib1.whlib`.

Now suppose you also want to make this identifier `x` available in library `lib3.whlib`, that does not directly load `lib1.whlib`, but does load `lib2.whlib`. Because the identifier is available in `lib2.whlib` (it loads `lib1.whlib`), we can tell library `lib2.whlib` to export the identifier to `lib3.whlib`.

The syntax to export an identifier is:

```
LOADLIB "<library>" EXPORT identifier1, identifier2, identifier_x;
```

Only public identifiers can be exported.

Exporting identifiers can be quite complex. The next examples explain in some more detail what to expect when exporting identifiers.

Let's suppose we have a library `lib1.whlib` that declares a public integer `i`:

```
<?wh // lib1.whlib
// declaring integer i, initialise it with value 10
PUBLIC INTEGER i := 10;
?>
```

Library `lib1.whlib` is loaded from library `lib2.whlib`.

`lib2.whlib` has access to the integer, so it can change its value:

```
<?wh // lib2.whlib
LOADLIB "currentsite::/lib1.whlib";
i := 20; // the value of i is changed from 10 (as it was in lib1.whlib) to 20
?>
```

Now suppose another library, called `lib3.whlib`, want access to the integer too. We can accomplish this by exporting the integer from `lib2.whlib`:

```
<?wh // lib2.whlib
LOADLIB "currentsite::/lib1.whlib" EXPORT i;
i := 20; // the value of i is changed from 10 (as it was in lib1.whlib) to 20
?>
```

`lib3.whlib` can now get access to the integer by loading `lib2.whlib`:

```
<?wh // lib3.whlib
LOADLIB "currentsite::/lib2.whlib";
PRINT( ToString(i) );
?>
```

In this example the PRINT command is used to print the value of i.

In this case i would have a value of 20, because it was initialised with value 10 in lib1.whlib, and then updated to 20 in lib2.whlib.

Now suppose lib3.whlib does not only load lib2.whlib, but also loads lib1.whlib:

```
<?wh // lib3.whlib  
  
LOADLIB "currentsite::/lib2.whlib";  
LOADLIB "currentsite::/lib1.whlib";  
  
PRINT( ToString(i) );  
  
?>
```

Lib3.whlib would then have direct access to integer i, and its value would be 10.

## 4. Working with the WebHare database

---

In paragraph 2.4 of this manual WebHare's relational database has already been introduced. In this chapter the four most important database tables will be explained in more detail, and the manner in which to retrieve information from these tables is described.

### 4.1 Goals

After studying this chapter you will understand:

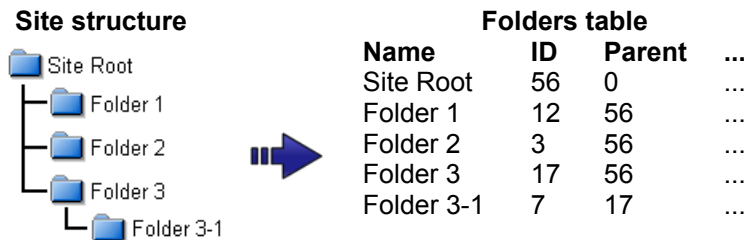
- How records are stored in the Files, Folders, Pages and DocObject tables.
- What the most important cells of the records in each of these tables are.
- How to retrieve these records from the database.
- How to use the `forevery` statement.

### 4.2 The records in the Folders table

The `folders` table stores all information about every folder you create in your WebHare site. Every time you add a new folder to your site, a new record in the Folders table is created. This new record contains a number of cells, of which the most important are:

- The `ID` cell. The ID is a unique number, used to identify the folder.
- The `name` cell, in which the name of the folder is stored.
- The `title` cell, used to store the folder's title.
- The `description` cell, in which the description of the folder is stored.
- The `parent` cell. This cell contains the ID of the folder in which the new folder is created.

Consider the following example site structure:



This site structure is stored in the folders table. Every folder is a record in this table.

In this example, please note the following:

- Only the `Name`, `ID` and `Parent` cells are presented. A folder record in the real folders table contains many more cells.
- The ID of the parent of the root folder of a site is *always* 0 (zero).
- Folder ID's are automatically assigned by the WebHare database. You can in no way influence the ID of a folder, nor is there any way to predict the ID of a folder. In your templates you must always assume folder ID's are assigned randomly. Furthermore, a folder ID can *never* be negative.
- The value of the parent cell of a folder is always equal to the ID of the folder one level upwards in the folder structure. The value of the parent cell of folder 'Folder 3-1' equals the ID of folder 'Folder 3', namely 17.

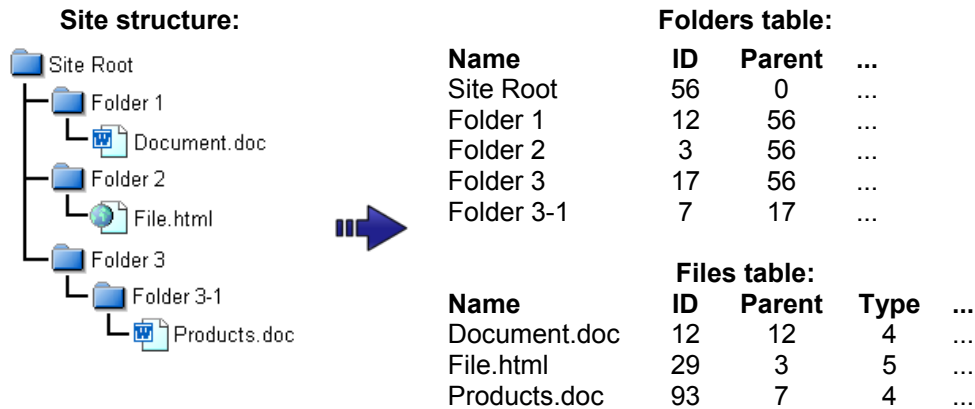
For more information about the folders table please refer to the HareScript Reference (ADDME: link).

### 4.3 The records in the Files table

Besides folders a WebHare site also contains files. These files are all stored in the *files* table of the WebHare database. Every file is a record in this table, and has the following cells, amongst others:

- An *ID* cell, containing a unique number used to identify the file.
- A *name* cell, containing the name of the file.
- A *title* cell, containing the title of the file.
- A *description* cell, used to store a description for the file.
- A *keywords* cell, to store some keywords that are characteristic for the content of the file.
- A *type* cell. The type of a file is an integer representing the type of the file, for instance type value 4 represents a Word document and value 5 represents a HTML file.
- A *parent* cell, containing the ID of the folder in which the file is placed. The value of the parent cell refers to the value of a folder ID cell in the folders table.

Let's refer back to the site structure used to describe the folder table in the previous paragraph, and add some file to it:



In this example, please note the following:

- Only the Name, ID, Parent and Type cells are presented. A file record in the real files table contains many more cells.
- A file ID is automatically assigned by the WebHare database, and can in no way be influenced. Furthermore, file IDs and folder IDs can have the same value. In this example both 'Document.doc' and 'Folder 1' have an ID value of 12.
- File IDs can *never* be 0 (zero) or negative.
- The value of the *parent cell* of a file record refers to the *ID cell* of the folder record in which the file resides. In this example the parent of 'File.html' is 3, which is the ID of folder 'Folder 2'.

More information about the files table and all available cells is available in the HareScript Reference (ADDME: link).

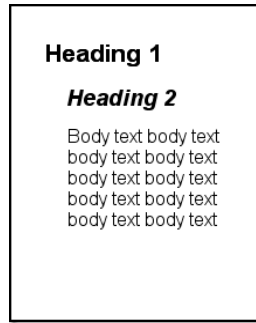
### 4.4 The records in the Pages table

As you know, WebHare can split-up large Word documents into multiple HTML output files when publishing the documents. Each one of these output HTML file will be a record in the *pages* table.

The pages table is a *virtual table*, and is not physically stored in the database. It only exists during publication, and its records refer to the pages that are created for the specific document that is published. For instance if you publish two Word documents, WebHare will create two pages tables, one for each Word document. Once the documents have been published, these tables will not be available anymore.

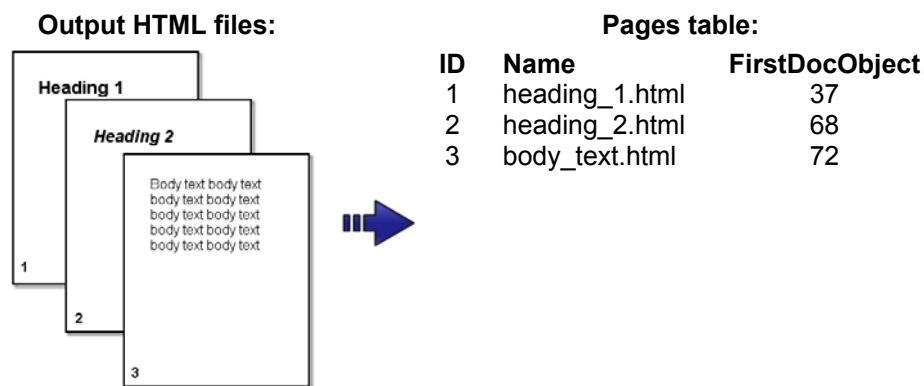
The pages table will *always* contain at least one record, because there will always be at least one output file.

In the following example a Word document is presented, containing three paragraphs. The first paragraph is marked as 'Heading 1' style, the second is a 'Heading 2' and the third is marked with style 'Body text':



We'll assume the word document will be split at both the 'Heading 1' and 'Heading 2' paragraphs, resulting in three output HTML files. The first output file will contain the 'Heading 1' paragraph, the second will contain the 'Heading 2' paragraph and the third contains the 'Body text' paragraph.

The following example pages table would be created during publication of the document:



In this example, please note the following:

- The pages table only contains three cells. There are no other cells than the ID, Name and FirstDocObject cells.
- The ID of the first page that is created is *always* 1.
- Page IDs are sequential. The second page will always have an ID value of 2, the third will be 3 and so on.
- The names of the pages are chosen by WebHare. There is no easy way to influence the naming. It can be done, but this manual will not go into the details on how to influence the page names.
- The value of the FirstDocObject cell refers to the ID of the first DocObject that is available on the page. The DocObjects table will be explained in detail in the next paragraph. In this example the value of the FirstDocObject cell refers to the first paragraph displayed on the page, e.g. 37 refers to paragraph 'Heading 1', 68 refers to 'Heading 2' and 72 refers to the ID of the 'Body text' paragraph.

More information about the pages table is available in the HareScript Reference (ADDME: links). For more information on how to influence the names of the pages please look at the WebHare support site (ADDME: link EN Article!!!).

#### 4.5 The records in the DocObjects table

As already mentioned in paragraph 2.5 (ADDME: link) of this manual, WebHare uses a completely unique conversion engine to publish Word documents. This engine analyses the structure of the Word documents, and creates an object tree based on this tree. Every object is then represented by a record in the *DocObjects* table.

A record in the DocObject table will always contain the following cells, amongst others:

- An *ID* cell. The ID is a unique number used to identify the object

- A *Parent* cell. The value of the parent cell of an object refers to the ID of the object that contains it. For example: The parent of a table cell object refers to the ID of the table. If an object is not contained by any other object, the parent cell value is always 0 (zero).
- A *TocLevel* cell. The TocLevel cell refers to the TOC level value for the paragraph style, which was set in the profile. If no value was set the TOCLevel cell has value 0 (zero).
- A *TocParent* cell. The TocParent value refers to the ID of the first preceding object for which a higher TocLevel was specified.
- A *Page* cell, referring to the ID of the page from the pages table on which the DocObject is available.

This process can best be understood by looking at the following example document:

**Heading 1**  
Normal normal normal  
normal normal normal

**Heading 2**

text in cell 1
text in cell 2

**Heading 3**  
Body text body text  
body text body text  
body text body text

The content of this document starts with a paragraph of style 'Heading 1', followed by a paragraph of style 'Normal'.

Next a 'Heading 2' paragraph, followed by a table with 2 cells is introduced. Every cell contains one paragraph.

Below the table a 'Heading 3' paragraph followed by a paragraph of style 'Body text' is written.

In this example we will assume the 'Heading 1' paragraph has a TOC level of 1, the 'Heading 2' paragraph has a TOC level of 2 and 'Heading 3' has TOC level 3. The other paragraphs have a (default) TOC level of 0 (zero). These values have been specified in the profile used to publish the document. This profile also specifies that WebHare should split the document on the 'Heading 1', 'Heading 2' and 'Heading 3' paragraphs.

When WebHare publishes this document, it will create the following example DocObjects table:

TOC tree:	DocObjects Table:																																																								
<pre> ├─ Heading 1 │  └─ Normal │  └─ Heading 2 │     └─ Table │        └─ Cell 1 │           └─ Text in cell 1 │              └─ Cell 2 │                 └─ Text in cell 2 │                    └─ Heading 3 │                       └─ Body text </pre>	➔	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th>ID</th> <th>Parent</th> <th>Toclevel</th> <th>TocParent</th> <th>Page</th> </tr> </thead> <tbody> <tr><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td></tr> <tr><td>3</td><td>0</td><td>0</td><td>1</td><td>1</td></tr> <tr><td>7</td><td>0</td><td>2</td><td>1</td><td>2</td></tr> <tr><td>13</td><td>0</td><td>0</td><td>7</td><td>2</td></tr> <tr><td>14</td><td>13</td><td>0</td><td>7</td><td>2</td></tr> <tr><td>22</td><td>14</td><td>0</td><td>7</td><td>2</td></tr> <tr><td>25</td><td>13</td><td>0</td><td>7</td><td>2</td></tr> <tr><td>29</td><td>25</td><td>0</td><td>7</td><td>2</td></tr> <tr><td>33</td><td>0</td><td>3</td><td>7</td><td>3</td></tr> <tr><td>37</td><td>0</td><td>0</td><td>33</td><td>3</td></tr> </tbody> </table>	ID	Parent	Toclevel	TocParent	Page	1	0	1	0	1	3	0	0	1	1	7	0	2	1	2	13	0	0	7	2	14	13	0	7	2	22	14	0	7	2	25	13	0	7	2	29	25	0	7	2	33	0	3	7	3	37	0	0	33	3
ID	Parent	Toclevel	TocParent	Page																																																					
1	0	1	0	1																																																					
3	0	0	1	1																																																					
7	0	2	1	2																																																					
13	0	0	7	2																																																					
14	13	0	7	2																																																					
22	14	0	7	2																																																					
25	13	0	7	2																																																					
29	25	0	7	2																																																					
33	0	3	7	3																																																					
37	0	0	33	3																																																					

In this example please note that:

- A DocObject record also has a DepthLevel and a Filter cell, as well as the cells shown in this example. The DepthLevel and Filter cells are not explained in this manual.
- Every object in the tree has an entry in the DocObject table, e.g. 'Heading 1' is the record with an ID value of 1, the table is record with ID value 13 and the paragraph in the second table cell is the DocObject record with ID value 29.
- The ID of the first object is always 1.
- The IDs of other objects than the first are chosen by WebHare, and can in no way be influenced. However, objects IDs are always sequential. An object with an ID value of 3 will always precede an object with an ID value of 4.

- The TocParent value for the 'Heading 3' paragraph refers to the ID of the 'Heading 2' paragraph, because it is the first preceding paragraph for which a higher TocLevel was set. The TocParent value of the 'Heading 2' paragraph is 1, referring to the 'Heading 1' paragraph.
- The document has been split on the 'Heading 1', 'Heading 2' and 'Heading 3' paragraphs, resulting in three output HTML pages. The value of the pages cell refers to the ID of the page on which the object is present.

More information about the DocObjects table can be found in the HareScript Reference (ADDME: link).

#### 4.6 Retrieving records from the database

As already briefly mentioned in paragraph 3.5 of this manual, there are many ways to retrieve information from the WebHare database. Before explaining these retrieval methods in more detail, we'll first have another look at the record variable.

As you know, a record contains cells that contain data of various types. For instance a file record contains information about the file's ID (an integer) its name (a string) and its title (also a string). This information can be retrieved by using the *cell operator*.

The syntax for the cell operator is:

```
<record>.<cell>
```

For example if you want to print a file's title, you would use:

```
// find the record of the file with ID 237
RECORD theFile := FindFile(237);
PRINT(theFile.title); // printing the file's title by using the cell operator
```

In this example the FindFile() function is used to retrieve a file record from the database. This function accepts one parameter; the ID of the file record that should be retrieved.

WebHare supports many functions like this, for example:

- FindFileByName(). This function returns a file record by searching the database for a specific filename, in a specific folder.
- FindFolder(). This function accepts a folder ID and returns the corresponding folder record.
- FindDocObject(). The findDocObject function accepts a DocObject ID, and returns the DocObject record.

These built-in functions can be very useful to retrieve a single record, but sometimes you don't have the required information to use them. For instance, you can't use the FindFile() function if you don't know the ID of the file record you want to retrieve.

In an example scenario, suppose we would want to print a list of all files in a specific folder. Also presume we only know the ID of the folder, but not the ID's of the files in this folder. In this case we can't use the FindFile() function. In order to retrieve the required file records, we would have to use a *select* statement.

#### The select statement

The select statement is used to conditionally select records from a database table. You can specify exactly which cells records you need.

In our example scenario, we could use the following code to retrieve the files we need:

```
/* Get a list of all files in the folder.
   We'll assume the folder has an ID of 17 */
RECORD ARRAY allFiles := SELECT name
                           FROM files
                           WHERE parent = 17;
```

Let's have a detailed look at this example code.

Because we need a list of file records, we first declare a record array variable called 'all\_files'. Next we assign the records we need to this record array using the assignment operator (:=).

At first we only need the names of the files, so we're only selecting the 'name' cells. To further specify exactly which files we want a condition has to be specified. This condition (WHERE parent = 17) tells WebHare to only get the file records for which the parent cell has a value of 17. This means we only want the files that are placed in the folder with an ID value of 17.

If we were to omit the WHERE clause, every file from the database would be selected.

Now suppose we also want to use the titles and descriptions for these files. The code would be:

```
RECORD ARRAY all_files := SELECT name, title, description
                           FROM files
                           WHERE parent = 17;
```

As you will notice, the cells we need are separated by commas.

If we would then want to order the list alphabetically based on the file titles, we would write:

```
RECORD ARRAY all_files := SELECT name, title, description
                           FROM files
                           WHERE parent = 17
                           ORDER BY title ASC;
```

The 'ORDER BY title ASC' statement tells WebHare to order the records in the record array in ascending order (A comes before Z), based on their titles. If you want to use a descending order (Z comes before A), we would change the ASC keyword to DESC.

Now, suppose the folder contains two files with the same title. The current 'ORDER BY' statement would not suffice to order these records properly. We would have to add another order condition:

```
RECORD ARRAY all_files := SELECT name, title, description
                           FROM files
                           WHERE parent = 17
                           ORDER BY title ASC, name DESC;
```

Now we're primarily ordering our list based on the titles of the files, and secondarily based on the file names. Because file names must always be unique in a specific folder, this would result in a correctly ordered list.

As a last addition to the select statement we could also limit the number of elements in the list to three:

```
RECORD ARRAY all_files := SELECT name, title, description
                           FROM files
                           WHERE parent = 17
                           ORDER BY title ASC, name DESC
                           LIMIT 3;
```

This would result in a record array that contains a maximum of three file records.

As you have seen, the select statement offers a powerful way to specify the records you need. But having the necessary records is only part of what we want; we also want to print the files on our page. We need the *forevery* statement for this.

## The forevery statement

The forevery statement is used to loop through an array, and execute a command for every element in the array. The general syntax for the forevery statement is:

```
FOREVERY ( [type] variable FROM array )
{
  statement 1;
  statement 2;
  statement x;
}
```

Based on the examples from the previous paragraph, we'll print the list of all files in the folder. The select statement provided us with a list of these files, so all we have to do now is walk through this list and print the necessary information for every element:

```
// Get the list of files we need
RECORD ARRAY all_files := SELECT name, title, description
                           FROM files
                           WHERE parent = 17
                           ORDER BY title ASC, name DESC
                           LIMIT 3;

/* loop through the list, and print the name and title for every file in the list */
FOREVERY (RECORD the_file FROM all_files)
{
  PRINT(the_file.name); // print the file name
  PRINT(the_file.title); // print the file title
}
```

Let's have a look at this code in more detail:

The forevery statement starts a loop through all elements of the 'all\_files' array. If the array contains three elements, the code contained in the forevery statement will be executed three times, once for every element.

Because we want to do something with the elements in the array, we have to keep track of the current element in the loop. This element is stored in the record variable called 'the\_file'.

In plain English this example forevery statement could be translated to:

"Get the first element from the list called 'all\_files'. Store this element in variable 'the\_file'. Print the name and title of this element. Do the same for the second element, and so on for every element in the list."

As you know from paragraph 3.5, you can use many types of arrays in HareScript. The forevery statement can be used for every type of record. The next example shows how to use it to print some strings from a string array:

```
/* split string 'a-b-c-d-e-f' at every -, resulting in a string array with these
elements: 'a','b','c','d','e' and 'f';
STRING ARRAY my_letters := TOKENIZE("a-b-c-d-e-f","-");

// print every string in the string array
FOREVERY (STRING the_letter FROM my_letters)
{
  PRINT(the_letter);
}
```