Scripting Manual

confirmit® 9

This document is only intended for registered Confirmit users. No part of this document may be reproduced, transmitted, or stored in any form to other than registered Confirmit users without the express written permission of Future Information Research Management - FIRM.

The manual covers features of JScript .NET with relevance to writing server-side scripts in Confirmit. As such, the manual is not to be interpreted as a general documentation of JScript .NET.

The manual is developed based on JScript .NET and Confirmit as of November 27th, 2004 (Confirmit version 9.0). New features may be introduced without notice after this date.

FIRM offers scripting assistance in Confirmit (development, support, quality assurance etc) as chargeable consultancy.

All examples depicted herein are fictitious. FIRM does not guarantee that scripts presented will work as intended by the user, and it is the user’s sole responsibility to perform any quality assurance deemed necessary to ensure that the purpose with a script is achieved.

1 Introduction.......... 8

1.1 DIFFERENCES BETWEEN JSCRIPT AND JSCRIPT .NET.......... 8

1.1.1 All Variables Must Be Declared. 9

1.1.2 Functions Become Constants.......... 10

1.1.3 The arguments Object is not Available.......... 10

1.1.4 Using a Sorting Function in the sort Method (on Array).......... 10

1.2 NEW IN THIS VERSION OF THE SCRIPTING MANUAL.......... 10

1.2.1 New Functions in Confirmit 9.0.......... 10

1.2.2 New Chapter: Useful ASP.NET Objects.......... 11

1.2.3 New Chapter: Testing of Scripts.......... 11

2 Where is Scripting Used in Confirmit?.......... 12

2.1 CONDITIONS.......... 12

2.2 FILTERING ANSWER LISTS, SCALES AND LOOPS.......... 13

2.2.1 Precode Masks and Scale Masks.......... 13

2.2.2 Column and Question Masks.......... 14

2.3 TEXT SUBSTITUTION/RESPONSE PIPING.......... 15

2.4 VALIDATION CODE.......... 16

2.5 SCRIPT NODES.......... 18

3 Comments.......... 20

4 Types, Variables and Constants.......... 21

4.1 NAMING.......... 21

4.2 DATA DECLARATION.......... 21

4.3 UNDEFINED VALUES.......... 22

4.4 NULL.......... 22

4.5 TYPES.......... 24

4.5.1 Numeric.......... 24

4.5.2 Boolean.......... 26

4.5.3 Characters and Strings.......... 26

4.6 CONVERSION BETWEEN TYPES/CONVERSION FUNCTIONS.......... 27

4.6.1 Conversion Methods in JScript .NET.......... 28

4.6.2 Conversions Methods in Confirmit.......... 29

5 Operators and Expressions **32**
5.1 TERMINOLOGY 32
5.2 ARITHMETIC OPERATORS 32
5.3 LOGICAL OPERATORS 33
5.4 COMPARISON OPERATORS 34
5.5 STRING OPERATORS 34
5.6 ASSIGNMENT OPERATORS 34
5.7 NEW 35
5.8 THE CONDITIONAL EXPRESSION TERNARY OPERATOR 35
5.9 COERCION 37
5.10 OPERATOR PRECEDENCE 38
5.11 SHORT CIRCUIT EVALUATION 39
6 Simple Statements **40**
6.1 DECLARATIONS 40
6.2 ASSIGNMENT STATEMENTS 40
6.3 THE IF STATEMENT 41
6.3.1 if *41*
6.3.2 if-else *41*
6.3.3 Using Curly Brackets in if Statements *42*
6.4 THE SWITCH STATEMENT 42
7 Arrays **45**
7.1 TYPED ARRAYS 45
7.1.1 Declaring Typed Arrays *45*
7.2 JSCRIPT ARRAYS 46
7.2.1 .Declaring JScript Arrays *46*
7.3 LENGTH 47
7.4 THE LENGTH PROPERTY 47
8 Methods of the Form Objects **48**
8.1 GET AND SET 48
8.2 LABEL 48
8.3 VALUE 48

8.4 VALUELABEL 48
8.5 DOMAINVALUES 49
8.6 DOMAINLABELS 49
8.7 CATEGORIES 49
8.8 CATEGORYLABELS 49
8.9 VALUES 49
8.10 APPLYING THE METHODS ON DIFFERENT TYPES OF QUESTIONS 49
8.10.1 Open Text Question 50
8.10.2 Single Question 51
8.10.3 Multi Question 52
8.10.4 Grid Question 53
8.10.5 Referencing the Elements of a Multi or a Grid 54
8.10.6 Loops 55
8.10.7 3D Grid 56
8.10.8 Implicit Conversion of Arrays to Strings 56
8.11 OVERVIEW – METHODS OF BASIC VARIABLE OBJECTS IN CONFIRMIT 58
9 Loop Statements 59
9.1 THE WHILE STATEMENT 59
9.2 THE DO WHILE STATEMENT 62
9.3 THE FOR STATEMENT 63
9.4 LOOP NODES IN CONFIRMIT 65
9.5 THE BREAK STATEMENT 66
9.6 THE CONTINUE STATEMENT 66
9.7 THE LABEL STATEMENT/NESTED LOOPS 67
10 Functions 71
10.1 BUILT-IN FUNCTIONS IN CONFIRMIT 71
10.1.1 Arithmetic Functions 71
10.1.2 Range 75
10.1.3 Context Information 76
10.1.4 Browser Information 80
10.1.5 Quota check 81

10.1.6 Panel *82*

10.1.7 Classification Functions *83*

10.1.8 General Utilities *88*

10.2 CREATING YOUR OWN FUNCTIONS 91

10.2.1 Defining functions *92*

10.2.2 Function Call *92*

10.2.3 Functions with a Fixed Number of Arguments *93*

10.2.4 Functions with a Variable Number of Arguments *95*

10.2.5 The return Statement *96*

10.2.6 Local Variables *96*

11 Objects 98

11.1.1 Properties *98*

11.1.2 Methods *98*

11.1.3 Constructors: Creating Instances of Objects *98*

12 Confirmit's f Function 100

12.1 CALLING THE F FUNCTION 100

12.2 STORING THE FORM OBJECT IN A VARIABLE 100

12.3 COMPOUNDS 100

12.4 PROPERTIES 102

12.5 METHODS 103

13 Working with Sets 105

13.1 CONSTRUCTOR 105

13.2 FUNCTIONS RETURNING SETS 105

13.2.1 The a Function *105*

13.2.2 set, nset and nnset *105*

13.2.3 The Filter Function *106*

13.2.4 The f Function *106*

13.3 METHODS OF THE SET OBJECT 106

13.3.1 inc *106*

13.3.2 size *107*

13.3.3 union, isect and diff *107*

13.3.4 Combining Set Operators *109*

13.3.5 members *111*

13.4 USER-DEFINED FUNCTIONS IN PRECODE OR SCALE MASKS 113

14 Predefined JScript .NET Objects 115

14.1 THE DATE OBJECT 115

14.1.1 Constructors *115*

14.1.2 Methods *116*

14.1.3 Confirmit Date Functions *123*

14.2 THE MATH OBJECT 128

14.2.1 Properties *128*

14.2.2 Methods *129*

14.3 THE STRING OBJECT 134

14.3.1 Constructors *134*

14.3.2 Properties *134*

14.3.3 Index *134*

14.3.4 Converting to String from Other Types *134*

14.3.5 Methods *134*

14.4 REGULAR EXPRESSIONS 140

14.4.1 Regular Expression Syntax *141*

14.4.2 Order of Precedence *149*

14.4.3 The Regular Expression Object *150*

14.4.4 String Object Methods that Uses Regular Expression Objects *151*

14.5 THE ARRAY OBJECT 153

14.5.1 Combining Arrays *153*

14.5.2 Converting Arrays to Strings *153*

14.5.3 Removing and Adding Elements *154*

14.5.4 Changing the Order of the Elements *154*

14.5.5 slice and splice *157*

15 Customizing Standard Error Messages 158

15.1 FUNCTIONS FOR STANDARD VALIDATION 158

15.2 TEMPLATE BASED ERROR MESSAGES 159

15.2.1 Answer Required Checks *160*

15.2.2 Exclusivity Tests *160*

15.2.3 Other-Specify Checking *161*

15.2.4 Rank Order Tests *162*

15.2.5 Answer Size For Fixed-Width Fields *162*

15.2.6 Numeric Validation *163*

15.2.7 Precision Error Tests *163*

15.2.8 Scale Error Tests *164*

15.2.9 Range Error Tests *164*

16 Useful ASP.NET Intrinsic Objects **165**

16.1 REQUEST 165

16.1.1 Form *165*

16.1.2 Querystring *165*

16.1.3 Cookies *165*

16.1.4 ServerVariables *166*

16.2 RESPONSE 168

16.2.1 Write *168*

16.2.2 Cookies *168*

17 Testing Survey Scripts **171**

17.1 CHECK SCRIPT CODE 171

17.2 RANDOM DATA GENERATOR 171

17.3 TIPS FOR DEBUGGING 172

18 Programming Conventions **174**

18.1 COMMENTS 174

18.2 NAMING CONVENTIONS 174

18.3 SPACES AND LINE BREAKS 174

18.4 CURLY BRACKETS 174

18.5 SEMI COLON 175

18.6 THE STEP BY STEP APPROACH 176

18.7 WRITING EFFICIENT CODE 176

Appendix A: Answers to Exercises **177**

Appendix B: Further Reading 180

Appendix C: Confirmit Language Codes 181

Appendix D: Codepage 187

EXAMPLES 189

INDEX 192

1 Introduction

Most Confirmit questionnaires contain some amount of script code. Scripts are used:

- In conditions for controlling the flow through the questionnaire (skipping logic) and controlling if questions are to be displayed or not (question masks and column masks (in 3D grids))
- for filtering the lists displayed in questions and the iterations in loops based on previous answers (precode and scale masks),
- in form elements for text substitution (response piping),
- for custom validation of user input and
- in general-purpose code contained in script nodes.

Confirmit uses Microsoft's JScript .NET scripting engine to evaluate all questionnaire expressions and to execute scripts. The run-time environment of the interview engine supplies a number of functions and objects that provide references to and let you manipulate survey variables. This documentation will both cover some of the fundamentals of JScript .NET and the functions and objects provided by Confirmit.

JScript was introduced with Internet Explorer 3. It is Microsoft's version of Netscape's **JavaScript** language. JavaScript came with Navigator 2 and replaced the language **LiveScript**, which was developed to add a basic scripting capability to both Navigator and Netscape's Web-server line of products. Netscape and Microsoft submitted their scripting languages to the European Computer Manufacturers Association (ECMA) for standardization. ECMA released the standard ECMA-262, which describes the **ECMAScript** language. Microsoft worked closely with ECMA and achieved ECMAScript compliance with JScript .NET 3.1.

JScript .NET is the next generation of an implementation by Microsoft of the ECMA-262 language. It has been developed in conjunction with ECMAScript Edition 4. It extends the language to a true object-oriented scripting language. The main differences between JScript .NET and JScript 5.6 relevant to using JScript in Confirmit surveys are described in Chapter 1.1 Differences between JScript and JScript .NET.

JScript .NET is not a condensed version of another programming language, nor is it a simplification of anything. It is a modern scripting language with a wide variety of applications.

JScript .NET is JScript extended with features of class-based languages. All script code in Confirmit will however be wrapped as functions inside a class. This means that you can not define your own classes in Confirmit scripts.

This manual is only covering server-side programming, i.e. scripts that are running on the server. We will not be looking at client-side scripts (scripts that run on the respondent's browser – typically written in JavaScript), so for example scripting of HTML elements is outside of the scope of this documentation.

The next two chapters are most relevant for readers that have experience in JScript and/or are familiar with the previous editions of the Confirmit Manual. If you have no experience in JScript you may move directly to Chapter 2 Where is Scripting Used in Confirmit?.

1.1 Differences between JScript and JScript .NET

Confirmit started using JScript .NET instead of JScript in version 8.5. Most features of previous versions of JScript are included in JScript .NET. However, you may need to make changes to some of your scripts if you are moving them to the new interview engine introduced in Confirmit 8.5. **It is strongly recommended to test your existing, live surveys in test mode before generating WI files in production mode in the new interview engine.**

One of the major improvements that JScript .NET offers when writing scripts in Confirmit, is that it has true compiled code. This has been used to provide script error messages when using the "Check Script Code" functionality and when generating interview files instead of at runtime, which will save programmers a lot of time that they previously had to spend in checking the code by running test interviews. Combined with much faster web interview generation, this will give programmers substantial productivity gains. (Some testing will still be needed since not all errors can be detected while compiling).

To benefit fully from the performance improvements in JScript .NET, the JScript .NET in Confirmit surveys will be compiled in *fast* mode. Since fast mode places some restrictions on the type of code allowed, programs can be more efficient and execute faster. However, some features available in previous versions are not available in fast mode.

In fast mode, the following JScript behaviors are triggered:

- All variables must be declared.
- Functions become constants.
- Intrinsic objects cannot have expando properties.
- Intrinsic objects cannot have properties listed or changed.
- The `arguments` object is not available.
- Cannot assign to a read-only variable, field, or method.
- `eval` method cannot define identifiers in the enclosing scope.
- `eval` method executes scripts in a restricted security context.

Most of these changes should not affect scripts used in Confirmit. For example, for security reasons the `eval` method is not even allowed in Confirmit surveys. Following are more details on the requirements that may require you to change scripts that have been working on the previous survey engine.

1.1.1 All Variables Must Be Declared.

Previous versions of JScript did not require explicit declaration of variables. Although this feature saves keystrokes for programmers, it also makes it difficult to trace errors. For example, you could assign a value to a misspelled variable name, which would neither generate an error nor return the desired result. Furthermore, undeclared variables have global scope, which can cause additional confusion. So where you previously could write script code like

```
precodes = f("q1").categories();
for(i=0;i<precodes.length;i++)
 {
  <some statements…>
 }
```

you now have to declare the variables with the `var` keyword:

```
var precodes = f("q1").categories();
for(var i=0;i<precodes.length;i++)
 {
  <some statements…>
 }
```

This will still not explicitly declare the type of the variable: It may change type later as a result of an assignment. In addition to this "loose typing", JScript .NET can now be a strongly typed language. JScript .NET provides more flexibility than previous versions of JScript by allowing variables to be type annotated. This binds a variable to a particular data type, and the variable can store only data of that type. Although type annotation is not required, using it helps prevent errors associated with accidentally storing the wrong data in a variable and can increase program execution speed. More information on this in Chapter 4 Types, Variables and Constants.

In addition to this explicit type declaration, a technology called *"implicit type inferencing"* is introduced. Type inferencing analyzes your use of variables in the script code and infers the type of the variable for you. This means that you can get considerable improvements in speed using scripts also when not specifying the type of your variables, as long as your variables are not changing type.

1.1.2 Functions Become Constants

In previous versions of JScript, functions declared with the function statement were treated the same as variables that held a Function object. In particular, any function identifier could be used as a variable to store any type of data.

In fast mode, functions become constants. Consequently, functions cannot have new values assigned to them or be redefined. This prevents accidentally changing the meaning of a function (either a user-defined or a Confirmit function). You will now not be allowed to define several functions with the same name in the same project, or reuse the name of a function as a variable name. This should not cause particular problems for scripts used in Confirmit surveys, as it is just a bad programming habit to reuse function names anyhow. Disallowing this will avoid errors when it is not clear what function definition to refer to. It will also make it impossible to make one of your own or the standard functions in Confirmit inaccessible because it has accidentally been redefined due to e.g. being used as a variable name.

1.1.3 The arguments Object is not Available

Previous versions of JScript provided an arguments object inside function definitions, which allowed functions to accept an arbitrary number of arguments. The arguments object also provided a reference to the current function as well as the calling function.

In fast mode, the arguments object is not available. However, JScript .NET allows function declarations to specify a parameter array in the function parameter list. This allows the function to accept an arbitrary number of arguments, thus replacing part of the functionality of the arguments object. For more information, see Chapter 10.2.4 Functions with a Variable Number of Arguments.

There is no way to directly access and reference the current function or calling function in fast mode.

1.1.4 Using a Sorting Function in the sort Method (on Array)

Arrays can be sorted with the `sort` method. This method takes an optional function name as parameter, a function that defines how to sort elements if they are not to be sorted conventionally.

From Confirmit 8.5 a script node is wrapped inside a class, so when you refer to the sort function you have to use the keyword `this` to refer to the current instance.

```
array.sort ({this.sortFunction})
```

1.2 New in This Version of the Scripting Manual

1.2.1 New Functions in Confirmit 9.0

The following new, system provided Confirmit functions are introduced in Confirmit 9.0:

- `IsInRdgMode()`
- `GetRespondentValue(fieldname)`
- `SetRespondentValue(fieldname,value)`
- `isEmailTaken(email)`

See Chapters 10.1.3.9 IsInRdgMode, 10.1.6.2 isEmailTaken and 10.1.3.5 GetRespondentValue and SetRespondentValue .

In addition to this, `CurrentForm ()` used in 3D grids has changed behavior in Confirmit 9.0: It will now return the question id of the question it is used in (e.g. the grid, single or multi inside the 3D grid) instead of the question id of the 3D grid itself. See Chapter 10.1.3.1 CurrentForm.

1.2.2 New Chapter: Useful ASP.NET Objects

You may also use some .NET objects in your script code. Chapter 16 Useful ASP.NET Intrinsic Objects has examples of some ASP.NET objects that may be useful when scripting Confirmit surveys.

1.2.3 New Chapter: Testing of Scripts

Confirmit 9.0 introduces a couple of features that simplify testing and validation of script code:

- Check script syntax without having to compile the survey.
- Random data generator, giving the ability to simulate real interviews with random responses.

In chapter 17 Testing Survey Scripts we describe this as well as giving some tips on debugging script code.

2 Where is Scripting Used in Confirmit?

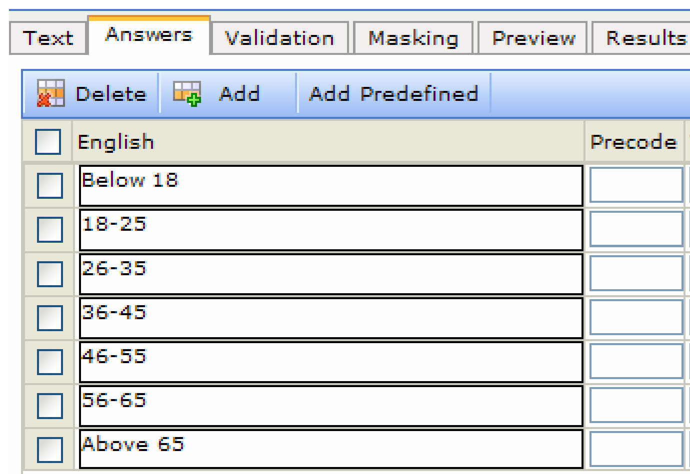
Let us start by looking at the different areas where scripting is used in Confirmit.

2.1 Conditions

In conditions you place some logical expression that is evaluated to true or false. If it is true the questions in the THEN-branch will be presented to the respondent. If it is false, the interview skips the THEN-branch. If the condition has an ELSE-branch the questions in the ELSE branch will be presented if the condition is false, otherwise they will be skipped.

Example 1: Screening Based on a Single Question

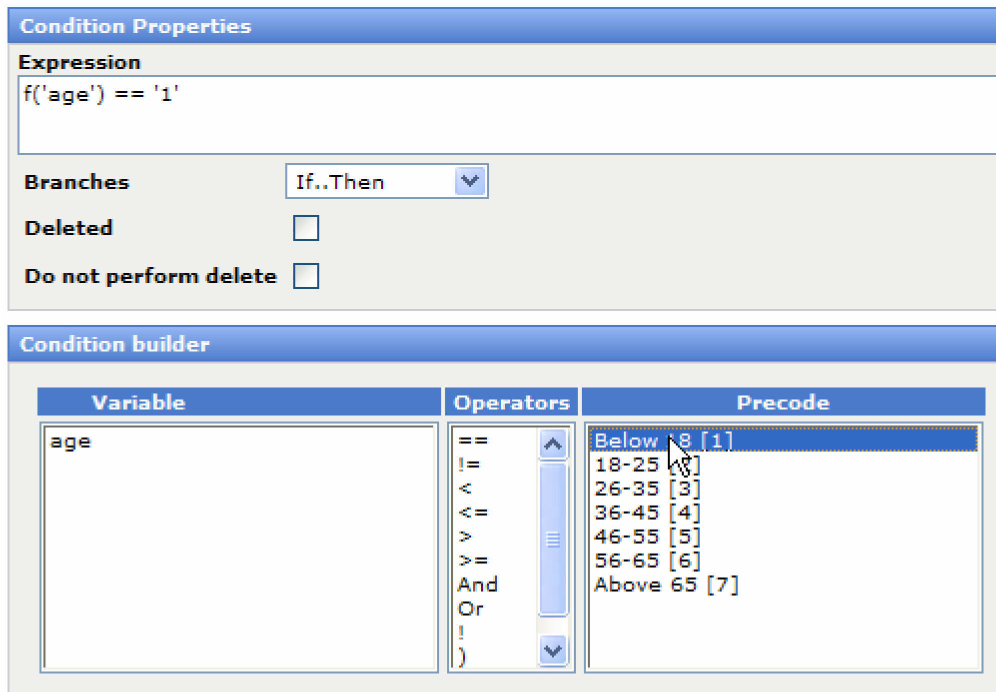
The questionnaire has an age question (single). You want to screen respondents below the age of 18 from doing the rest of the interview. The age question has the following answer list:



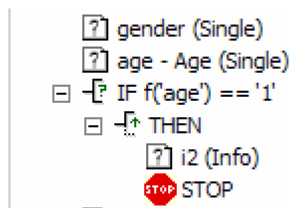
Since "Below 18" is the first item in the answer list and no precodes are specified, it will get the precode "1". To screen respondents that answer "Below 18", insert a condition after the single question with the following syntax:

```
f('age') == '1'
```

(The word "age" here is the question ID of the age question.) Building the conditional expression can be done by using the condition builder:



The routing will look like this:



Status is set to "screened" in the properties of the stop node. The interview will end there for all respondents answering "Below 18", but will continue for all the other respondents.

2.2 Filtering Answer Lists, Scales and Loops

It is very common to filter answer lists, scales and loops based on answers to previous questions. There are two types of filters:

Masks based on sets of precodes, which are used in "precode masks" of single, multi, grid, 3D grid questions and loops and "scale masks" of grids.

"Column masks", which are used in 3D grids to filter the columns and are based on true/false expressions, just like conditions.

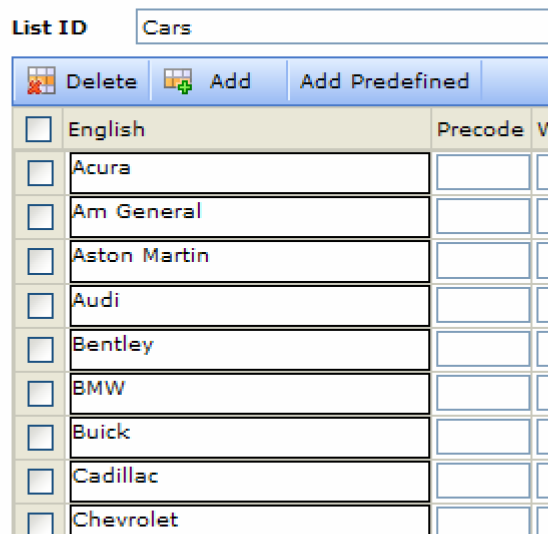
2.2.1 Precode Masks and Scale Masks

Precode masks are used to filter answer lists of 3D grids, grids, single, and multi questions and to define what iterations to run in a loop. In the precode mask field in properties of a question or a loop you may use a JScript .NET expression that evaluates to a set of precodes. The answer list (or loop member list) will be filtered based on the set of precodes in the precode mask field.

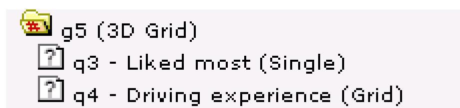
Scale masks are used to filter scale lists of grid questions. Use the scale mask field in the properties of a grid question to enter a JScript .NET expression that evaluates to a set of precodes. The scale list will be filtered based on this set of precodes.

Example 2: Filtering a Single Question Based on Answers to a Multi

You have a questionnaire that uses a list called "Cars".



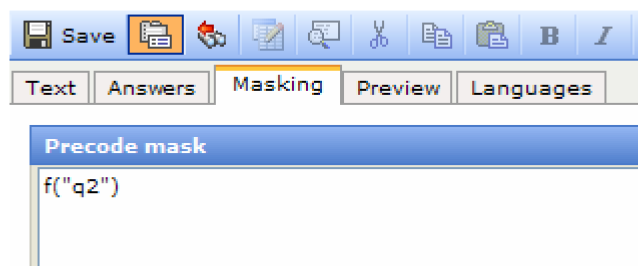
The list is used in two questions: First you have a multi question q2 which asks what cars you have tested, and then a 3D grid question g5 with a single question q3 which asks what car you did like most of the ones you tested, and a grid question q4 which asks you to rate the driving experience of the car.



Now, in the 3D grid question `g5` you want just the cars answered in `q2` to show. This can be achieved with a precode mask with the following code:

```
f("q2")
```

that will return a set with the precodes of the answers to q2. The answer list of the 3D grid g5 will be filtered so that only the answers with these precodes will be displayed.



When the question q3 is displayed to the respondent, it will only show the cars answered in q2.

2.2.2 Column and Question Masks

Column Masks are used for filtering columns in a 3D grid. If you want to dynamically exclude a column (a question element in a 3D grid), use this field to create a JScript .NET-expression that evaluates to true or false, just like when you make a condition. The column is displayed if the result is true, and not displayed if the result is false. If you leave the field empty, the column is always displayed.

Question Masks are similar to Column Masks, but used to filter an entire question. It is useful when you want a group of questions to appear on the same page, but one of more of them should not always be displayed. Before a condition a page break is automatically inserted, but for a question mask there will be no extra page break.

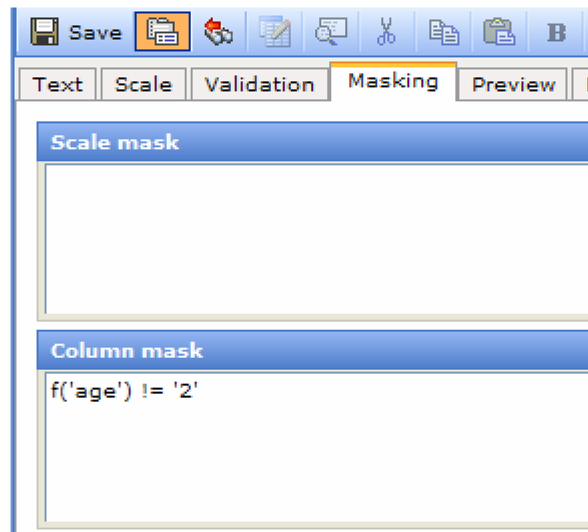
Example 3: Excluding a Column (Question) in a 3D grid

Let us say that in the 3D grid question from the previous example, we do not want inexperienced drivers (18-25 years) to answer question q4 – driving experience.

In the column mask field of question q4 we use the following code:

```
f('q2') != '2'
```

to exclude the question from the 3D grid whenever the respondent is in the age group 18-25. (The symbol "!=" means "not equal to".)



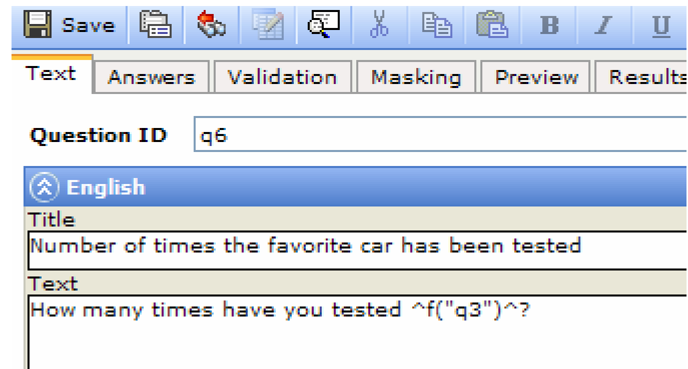
2.3 Text Substitution/Response Piping

In order to retrieve text or values from a question and insert it in the question wording of another question, you may use caret (^ - also known as "hat") in front of and after a JScript .NET expression. This **text substitution** or **response piping** can be used in all text fields, i.e. Title, Text, Instruction and Answer list/scale, but not in script nodes, conditions, precode/column masks or validation code fields.

Example 4: Piping in the Response to a Single Question

After the respondent has picked a favorite car in q3 we want to ask how many times he or she has tested that car. We want to pipe in the name of the car, and use the "hat" notation like this:

```
^f("q3")^
```



2.4 Validation Code

Confirmit provides several ways of validating survey responses. Some validation is based on the properties you define for your question, for example the field width on an Open Text Question.

Sometimes you need other types of validation on questions, or you want to specify your own error messages different from the error messages provided by the system. Enter your own validation code in the validation code field of the question's properties.

A validation code follows a pattern similar to this:

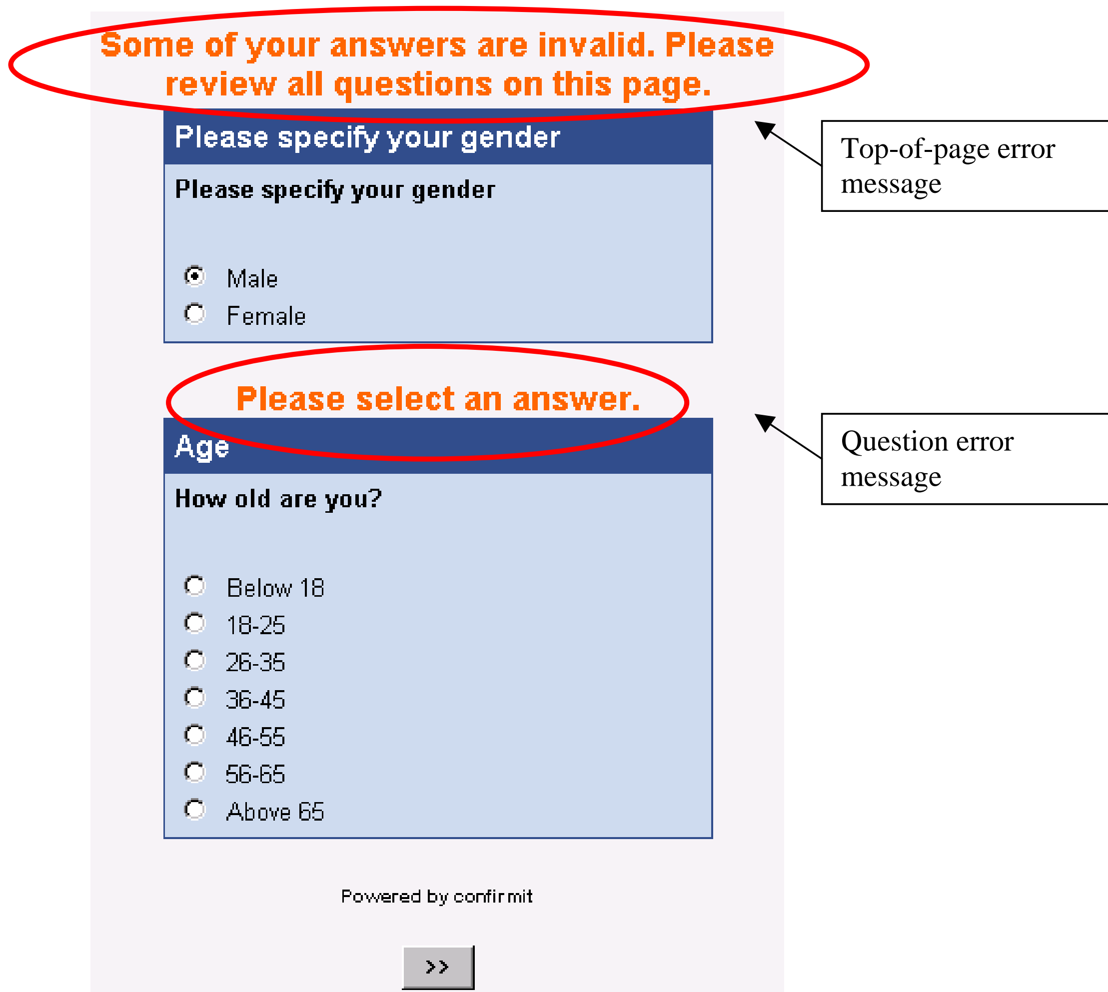
```
if(expression)
 {
   RaiseError();
   <some function(s) setting the text of the error message(s)>
 }
```

We will get back to this later; so don't worry if you don't understand all of it now. The terms expression and function will be explained later (in Chapters 5 Operators and Expressions and 7 Arrays).

Expressions similar to those in conditions and column masks can also be placed in the validation code field, i.e. expressions that are either true or false. If the expression is true, the function

```
RaiseError()
```

is called. This function tells the interview engine that there is an error situation. This means that the interview page should be re-displayed, this time with one or more error messages. The respondent is thus prohibited from moving to the next page until the expression in the validation returns false.



There are several functions available to set the text of error messages:

`ClearErrorMessage()`	takes away the error message at the top of the page (will remove the default error message).
`SetErrorMessage(langID, message)`	defines the text of the error message at the top of the page (will replace the default error message).
`AppendErrorMessage(langID, message)`	adds text to the current page's error message.
`ClearQuestionErrorMessage()`	takes away the error message for the current question (will remove the default question error message).
`SetQuestionErrorMessage(langID, message)`	defines the text of the error message for the current question.
`AppendQuestionErrorMessage(langID, message)`	adds message to the current question's error message.

For langID you insert a code to specify which language the error message is for. For example, the code

```
langIDs.en
```

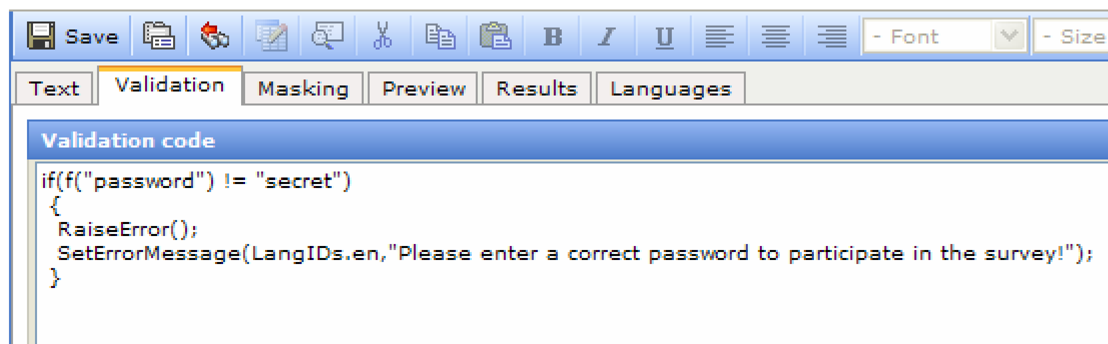
instructs the interview engine to use this error message when the language is English. These language codes, known as combidents, are listed in Chapter Appendix B: Further Reading.

Example 5: Password Check

Let us say you need to password protect an open survey. Then you can start the survey with a question asking for a password that will be the same for all respondents. This can be set up as an open text question with the "Password" property, so that *s are displayed instead of the text the respondent writes.

To check that the password is correct you can insert code similar to this one in the validation code field of the open question (In this example the question ID is password):

```
if(f("password") != "secret")
 {
  RaiseError();
  SetErrorMessage(LangIDs.en,"Incorrect password. Please enter the correct password to
participate in the survey.");
 }
```



2.5 Script Nodes

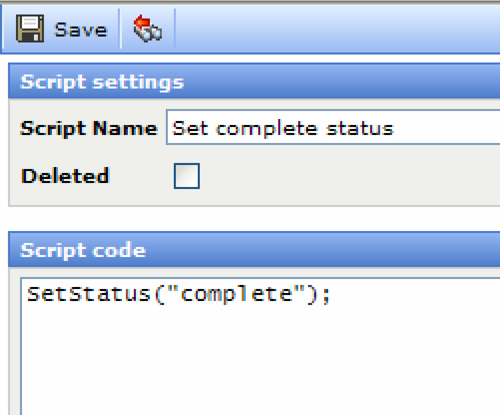
Script nodes typically contain code for

- Internal programming purposes.
- Defining functions used in precode masks, conditions, text substitution, other script nodes or validation code.
- Assigning values to different variables.
- Performing actions like sending an email, redirecting the respondent to a different URL, etc.

Example 6: Setting complete status before the end of the survey

Sometimes you want the "complete" status to be set before the last question (e.g. an open text "Other comments"-question), so that a respondent will be treated as a complete even though the last question(s) has not been answered. This can be done with the SetStatus function, which we will get back to in Chapter 10.1.3.7 GetStatus and SetStatus, in a script node:

```
SetStatus("complete");
```


Save
Script settings
Script Name
Set complete status
Deleted
Script code
SetStatus("complete");

3 Comments

Comments are text added in your scripts that are ignored when the script is run, but may be used to explain aspects of the code.

In JScript .NET you can add comments in two ways:

// is used to mark the rest of the line as a comment:

```
//This is a comment on one line.
```

/* are placed in front and */ after a comment that runs over several lines.

```
/* This is a comment on
   two lines */
```

Multi-line comments cannot be nested, because everything after `/*` will be interpreted as comments, and when the first `*/` appears, it will be interpreted as the end of the comment. So any text following the first `*/` will be treated as JScript code:

```
/* This is an attempt of a
   nested comment.
     /* Here is the second comment, inside the other.
        Both of these comments will terminate here->*/
   This line will be treated as JScript code and result in errors. */
```

It is recommended to add a lot of comments in your scripts, to explain for yourself and others what your script is supposed to do and what it can be used for.

But since it may be that you later want to comment out large parts of your scripts, including comments, and it is not possible to nest comments, it is recommended that you always use the single line comments, like this:

```
// This is a comment on
// two lines
```

This will make it easy to use `/*` and `*/` to comment out large sections of the script later without the nesting problems.

4 Types, Variables and Constants

If you need to calculate a value and save it for online reporting, for data exports or for use in the survey logic, you can store it in a hidden Confirmit question. A **hidden question** is an ordinary Confirmit question with the hidden property set. This will not be displayed to the respondent, but can be referenced like any other Confirmit question.

If you need to store temporary values e.g. as part of a calculation in a script, use a JScript **variable** or **constant**. The scope of a JScript variable or constant used in Confirmit is limited. It can only be accessed within the script node or validation code where it is defined. So if you need to access it later in the questionnaire, use a hidden question. A hidden question can be accessed from anywhere in the questionnaire.

4.1 Naming

Variable and constant names in JScript .NET and **question IDs** in Confirmit

begin with an upper- or lowercase letter (a-z, A-Z) or an underscore (_)

continue with letters (a-z, A-Z), digits (0-9) or underscore (_)

Examples of variable names:

```
counter
makeMoreMoney
car123
_temp
iThinkThisIsReallyBoring
```

Variable names are **case sensitive**, so makeMoreMoney and MakeMoreMoney are not the same variable**. This is a very common mistake.**

Even though variable and constant names can start with uppercase letters, it is recommended to follow the convention of **always starting variable and constant names with a lowercase letter**. This to easily distinguish it from e.g. functions, where the convention is to start with an uppercase letter. Variable and constant names should be made as descriptive as possible, e.g. instead of using names like `x` and `y`, you should try to describe what they refer to, e.g. `sumOfAllElements` or `precode`. When a variable or constant name consists of several words, each new word is usually started with an uppercase letter.

There are some reserved words that cannot be used as question IDs in Confirmit and some that cannot be used as variable or constant names in JScript .NET. See appendix C in the Confirmit User Manual and a JScript .NET reference manual. In addition to that, you cannot use names of functions (either the Confirmit provided functions described in this manual, or functions you define yourself in script nodes. See Chapter 10 Functions.

4.2 Data Declaration

A JScript program must specify the name of each **variable** that the program will use. In addition, the program may specify what **data type** each variable will store. Both of these tasks are accomplished with the `var` statement.

```
var counter : int;
```

This will declare a variable `counter` to be of type integer (see Chapter 4.4 Null for a description of types). Here it is not given an initial value, and will assume the default value for integers which is 0. You can also assign an initial value to it like this:

```
var counter : int = 1;
```

Constants are declared in the same way, with the keyword `const`, but <u>must</u> be initialized. A constant's value cannot be changed, whereas the value of a variable can.

```
const maxSelected : int = 100;
```

When you declare a variable or constant of a specific type, the value you assign to it must be valid for that type. You cannot declare an integer variable and try to assign a string value like `"This is a string"` to it.

You can make several declarations in the same row by listing them separated by commas:

```
var counter : int = 1,sumOfAllAnswer : int = 0;
```

This will give code that is harder to read so it is recommended to separate them on several lines instead:

```
var counter : int = 1;
var sumOfAllAnswer : int = 0;
```

Another reason for doing this is because it prevents you from doing errors that are hard to spot. Type annotation applies only to the variable that immediately precedes it. In the following code, x is an Object because that is the default type and x does not specify a type, while y is an int.

```
var x, y : int;
```

You do not need to use typed variables, but scripts that use untyped variables are slower and more prone to errors.

```
var counter;
```

Without a specified data type, the default type for a variable or constant is `Object`. Without an assigned value, the default value of the variable is `undefined`.

You can give a variable an initial value without declaring its type:

```
var counter = 0;
```

Untyped constants are defined in the same way:

```
const maxSelected = 100;
```

4.3 Undefined Values

A variable that is declared without assigning a value to it, will, if the data type is declared assume the default value for that type. For example, the default value for a numeric type is zero, and the default for the `String` data type is the empty string. However, a variable without a specified data type has an initial value of `undefined` and a data type of `undefined`.

To determine if a variable or object property exists, you can compare it to the keyword undefined (which will work only for a declared variable or property):

```
var x;
if(x == undefined)
 {
   <some code>
 }
```

You can also check if its type is `"undefined"` (which will work even for an undeclared variable or property):

```
if(typeof(x) == "undefined")
 {
   <some code>
 }
```

4.4 Null

`null` is used as "no value" or "no object". In other words, it holds no valid number, string, Boolean, array, or object (array and objects are complex data types we will get back to later). You can erase the contents of a variable (without deleting the variable) by assigning it the `null` value. Note that the value `undefined` and `null` compare as equal using the equality (==) operator.

In JScript, `null` does not compare as equal to 0 using the equality operator. This behavior is different from other languages, such as C and C++.

Example 7: *Removing an Answer in a Single or Grid Question*

You can use `null` to "remove" an answer to a question with a radio button (i.e. single or grid questions). Say you have a page with two single questions, present1 and present2, at the end of the survey. The respondent should answer just one of the questions, where the respondent can choose between two lists with incentives (for example wine or CD):

Wine

Please select you gift - either from this list of wines....

- ○ Gran Lurton Cabernet Sauvignon Reserva 1997
- ◉ Medalla Real Cabernet Sauvignon 1999
- ○ Santa Carolina Cabernet Sauvignon Reservado 2000
- ○ Ch. Panet 1997

CDs

...or from this list of CDs

- ○ Come Away with Me, Norah Jones
- ◉ The Divine Secrets of the Ya-Ya Sisterhood, Various Artists
- ○ The Eminem Show, Eminem
- ○ O Brother, Where Art Thou?, Various Artists - Soundtrack
- ○ Down the Road, Van Morrison

You have to set the "Not required" property on both questions. It is quite easy to write validation code to give an error message when both questions are answered (like in the screenshot above), but the respondent cannot de-select the answers, since this is not possible with radio-buttons. Then you have to remove the answers to the questions in the validation code, i.e. setting them to the `null` value:

```
if(f("present1").toBoolean() && f("present2").toBoolean()) //both questions answered
 {
   //Remove answers on both questions:
   f("present1").set(null);
   f("present2").set(null);
   //Provide error message
   RaiseError();
   SetErrorMessage(LangIDs.en,"Please select either a bottle of wine or a CD.");
 }
```

`toBoolean` is used to check if there is an answer on a question. It will be explained more thoroughly in Chapter 4.6.2.2 toBoolean. `set` is used to set the value of a question. It will be explained in Chapter 8.1 get and set.

This validation code will give this result when trying to move to the next page after selecting an answer on both questions:

Please select either a bottle of wine or a CD.

Wine

Please select you gift - either from this list of wines....

- Gran Lurton Cabernet Sauvignon Reserva 1997
- Medalla Real Cabernet Sauvignon 1999
- Santa Carolina Cabernet Sauvignon Reservado 2000
- Ch. Panet 1997

CDs

...or from this list of CDs

- Come Away with Me, Norah Jones
- The Divine Secrets of the Ya-Ya Sisterhood, Various Artists
- The Eminem Show, Eminem
- O Brother, Where Art Thou?, Various Artists - Soundtrack
- Down the Road, Van Morrison

4.5 Types

A **data type** specifies the type of value that a variable, constant, or function can accept. Type annotation of variables, constants, and functions helps reduce programming errors by making sure data that is assigned has the right types. Furthermore, type annotation also produces faster, more efficient code.

There are several primitive types of values in JScript .NET. In the following chapters we will present these types. We have grouped them as **numeric**, **Boolean and string** values.

Primitive types are types that can be assigned a single literal value. We will be looking at more complex types later.

4.5.1 Numeric

There are two main types of numeric data in JScript: Integers and floating-point data.

4.5.1.1 Integers

Positive whole numbers, negative whole numbers, and the number zero are integers. They can be represented in base 10 (decimal), base 8 (octal), and base 16 (hexadecimal). Most numbers in JScript are written in decimal. Octal and hexadecimal rarely have any practical purpose in Confirmit scripting; however, you should be aware of their denotation – particularly for octals, since it may cause unexpected results when a number is interpreted as an octal when it was supposed to be decimal.

You denote octal integers by prefixing them with a leading 0 (zero). They can only contain digits 0 through 7. Any number with a leading 0 will be interpreted as an octal, as long as it is not containing the digits 8 and/or 9, in which case it is interpreted as a decimal number.

You denote hexadecimal (hex) integers by prefixing them with a leading "0x" (zero and x or X). They can contain digits 0 through 9, and letters A through F (either uppercase or lowercase) only.

Both octal and hexadecimal numbers can be negative, but they cannot have a decimal portion and cannot be written in scientific (exponential) notation.

JScript .NET supports the following integral data types: `byte, ushort, uint, ulong, sbyte, short, int, long`. Variables of any integral data type can represent only a finite range of numbers. If you attempt to assign a numeric literal that is too large or too small to an integral data type, a type-mismatch error will be generated at compile time.

JScript value type	Range
`byte` (unsigned)	0 to 255
`ushort` (unsigned short integer)	0 to 65,535
`uint` (unsigned integer)	0 to 4,294,967,295
`ulong` (unsigned extended integer)	0 to approximately 10^{20}
`sbyte` (signed)	-128 to 127
`short` (signed short integer)	-32,768 to 32,767
`int` (signed integer)	2,147,483,648 to 2,147,483,647
`long` (signed extended integer)	Approximately -10^{19} to 10^{19}

4.5.1.2 Floating-point Data

Floating-point values are whole numbers with a decimal portion. They can either be represented with digits followed by a decimal point. (“period”/”dot”) and more digits (e.g. `1.29384`), or they can be expressed in scientific notation; that is, an uppercase or lowercase letter e is used to represent "times ten to the power of" (e.g. `7.64e3`). A number that begins with a single 0 and contains a decimal point is interpreted as a decimal floating-point literal and not an octal literal (see Chapter 4.5.1.1 Integers).

Additionally, floating-point numbers in JScript can represent special numerical values that integral data types cannot. These are:

- `NaN` (not a number). This is used when a mathematical operation is performed on inappropriate data, such as strings or the `undefined` value.
- `Infinity`. This is used when a positive number is too large to represent in JScript.
- `-Infinity` (negative Infinity). This is used when the magnitude of a negative number is too large to represent in JScript.
- `Positive and Negative 0`. JScript differentiates between positive and negative zero.

JScript supports the following floating-point data types:

JScript value type	Range
`float` (single-precision floating-point)	The `float` type can represent numbers as large as 10^{38} (positive or negative) with an accuracy of about seven digits, and as small as 10^{-44}. The `float` type can also represent `NaN` (Not a Number), positive and negative infinity, and positive and negative zero.

	This type is useful for large numbers where you do not need precise accuracy. If you require very accurate numbers, consider using the `Decimal` data type.
`Number, double` (double-precision floating-point)	The `Number` or `double` type can represent numbers as large as 10^{308} (positive or negative) with an accuracy of about 15 digits, and as small as 10^{-323}. The Number type can also represent `NaN` (Not a Number), positive and negative infinity, and positive and negative zero. This type is useful when you need large numbers but do not need precise accuracy. If you require very accurate numbers, consider using the `Decimal` data type.
`decimal`	The `decimal` type can accurately represent very large or very precise decimal numbers. Numbers as large as 10^{28} (positive or negative) and with as many as 28 significant digits can be stored as a decimal type without loss of precision. This type is useful when rounding errors must be avoided. The decimal data type cannot represent `NaN`, positive Infinity, or negative Infinity.

4.5.2 Boolean

The Boolean data type can only have two values. They are the literals

```
true
```

and

```
false
```

representing logical values. They are used in conditions. The THEN branch is executed when the expression evaluates to the Boolean value true, the ELSE branch when the expression evaluates to the Boolean value false.

JScript .NET automatically converts true and false into 1 and 0 when they are used in numerical expressions. When numbers are used in Boolean expressions, 0 is interpreted as false and any other number as true.

4.5.3 Characters and Strings

The `char` data type can store a single character.

A `string` value is a chain of zero or more characters (letters, digits, and punctuation marks) strung together. You use the string data type to represent text in JScript.

String and char values are enclosed in single (') or double (") quotation marks. You may use any of these marks, but always close with the same type of quotation mark as you opened it with. This is very convenient for example in strings like:

```
"I can't stand this anymore"
'"This is too much for me", he said.'
```

When you write JScript .NET code, you can have as many line breaks and blanks as you like in your code, but never have a line break inside a string. If you have a line break inside a string, you will get an error message.

However, there are codes you can insert for special formatting characters. E.g. for a line break use \n:

```
"I need a line break here\n and want to continue on the next line"
```

For apostrophe, use \':

```
'I can\'t stand this anymore'
```

Here is a table describing the special formatting characters:

Character	Meaning
`\'`	Single quote
`\"`	Double quote
`\\`	Backslash
`\n`	New line
`\r`	Carriage return
`\t`	Horizontal tab
`\b`	Backspace

A string that contains zero characters ("") is an empty (zero-length) string.

There is a significant difference between the string "123" and the number 123. The number 2 will be a smaller value than the number 123. But the string "2" is larger than the string "123", because the first character in the string is compared first, and the character 2 is larger than the character 1. (Like you are used to from alphabetical listings like dictionaries or phone directories). All values are stored as strings in Confirmit questions. So you have to convert them to numbers if you want them to be evaluated as numbers. We will get back to this.

4.6 Conversion between Types/Conversion Functions

Type conversion is the process of changing a value from one type to another. For example, you can convert the string, `"1234"` to the number `1234`. Data of any type can be converted to the String type. Some type conversions will never succeed because the types are too different.

Some types of conversions, such as from a string to a number, are time-consuming. The fewer conversions your program uses, the more efficient it will be.

In JScript .NET you can either have *implicit* or *explicit* conversion.

Explicit conversion is done by using the data type identifier (see Chapter 4.5 Types for lists of the data type identifiers). To explicitly convert an expression to a particular data type, use the data type identifier followed by the expression to convert in parentheses. Explicit conversions require more typing than implicit conversions, but you can be more certain of the result.

Here is a small example showing first how the number 1234 (integer) can be converted to the string “1234” and then to a double.

```
var i : int = 1234;
var d : double;
var s : String;
s = String(i);
```

Now the variable `s` holds the string “1234”. This type of conversion is called *widening*, since all possible integer values can be converted to string and string can also hold other values. Let us continue the example:

```
d = double(s);
```

Now the variable d holds the double 1234. This type of conversion is called *narrowing* since there are a lot of possible string values that cannot be converted to double (e.g. the string “Confirmit”). Explicit narrowing

conversions will usually work, but with loss of information. The string “Confirmit” converted to a double will give `NaN` (not a number). But some types are incompatible and will throw an error, and for some values there is not sensible value to convert to.

Implicit conversion occurs automatically when values are assigned to variable of a certain type. The data type of the variable determines the target data type of the expression conversion.

Here is a similar example to the one above showing first how the number 1234 (integer) can be converted to the string “1234” and then to a double.

```
var i : int = 1234;
var d : double;
var s : String;
s = i;
```

Now the variable `s` holds the string “1234”. It was converted to string since the receiving variable where of type string. This is an example of widening implicit conversion. Let us continue the example:

```
d = s;
```

Now the variable d holds the double 1234. This is an example of a narrowing conversion.

When this code is compiled, compile-time warnings may state that the narrowing conversions may fail or are slow. Implicit narrowing conversions may not work if the conversion requires a loss of information.

4.6.1 Conversion Methods in JScript .NET

4.6.1.1 parseInt

parseInt is used to convert a string value into an integer. It returns the first integer contained in the string or `NaN` (Not a Number) if the string does not begin with an integer. It is a method of the Global Object. The Global Object has no syntax, so its methods can be called directly. The expression

```
parseInt(string{,radix})
```

parses the string as an integer of base *radix*. *radix* is optional and is a value between 2 and 36 indicating the base of the number contained in *string*. If not supplied, strings with a prefix of '0x' are considered hexadecimal and strings with a prefix of '0' are considered octal. All other strings are considered decimal... Usually you want numbers to be treated as decimals, and since a leading zero would indicate that the number is an octal number (see Chapter 4.5.1 Numeric), it is a good idea to always include the base 10 when using `parseInt`.

The function will read numbers from the beginning of the string and will stop when the first non-digit is reached:

```
parseInt("123xyz",10)
```

returns 123

```
parseInt("xyz123",10)
```

returns `NaN` – not a number, since the first character is not a number.

4.6.1.2 parseFloat

parseFloat returns a floating-point number converted from a string. It is a method of the Global Object. The Global Object has no syntax, so its methods can be called directly:

```
parseFloat(numString)
```

The required *numString* argument is a string that contains a floating-point number. The function will read numbers from the beginning of the string and will stop at the first character that cannot be interpreted as part of a floating-point number. If the string does not begin with a floating-point number, `NaN` will be returned.

```
parseFloat("2.1e4xyz")
```

returns 21000 ($2.1 * 10^4$).

4.6.1.3 isNaN

The `isNaN` method returns `true` if the value is `NaN`, and `false` otherwise. It is a method of the Global Object. The Global Object has no syntax, so its methods can be called directly: You typically use this function to test return values from the `parseInt` and `parseFloat` methods.

```
isNaN(num)
```

num is a numeric value to test.

4.6.1.4 isFinite

The `isFinite` method returns `true` if the value is any other value than `NaN`, negative infinity or positive infinity. If it is any of those three, it returns `false`. It is a method of the Global Object. The Global Object has no syntax, so its methods can be called directly:

```
isFinite(num)
```

num is a numeric value to test.

4.6.1.5 toString

`toString` is a method of the Object object, which means it is available in any other JScript .NET object. It is used to convert a variable into a string. Sometimes this is needed on a variable before inserting it in a Confirmit question with the `set` method (see Chapter 8.1 get and set).

4.6.1.6 valueOf

`valueOf` is a method of the Object object, which means it is available in any other JScript .NET object. It returns the primitive value of the specified object, i.e. the numeric value of a number, the string value of a string etc.

4.6.2 Conversions Methods in Confirmit

There are also some methods available in Confirmit to convert values.

4.6.2.1 toNumber

`toNumber` is a method you can use to convert the value of a question from a string into a number:

```
f(qID).toNumber()
```

qID is the question ID.

<u>Example 8:</u> Screening on a Numeric Question

Let us say you have a survey with an initial open text question age with numeric property. In this question you ask for the respondent's age. For this survey you have defined a target: Persons between 18 and 35. To make a condition for your screening, you have to use toNumber to convert the answer from a string into a number. You may then make an expression like this in the condition:

```
f("age").toNumber() < 18 || f("age").toNumber() > 35
```

```
age - Age (OpenText)
IF f("age").toNumber() < 18 || f("age").toNumber() > 35
  THEN
    i4 (Info)
    STOP
```

Always use `toNumber` when you use expressions involving the operators <, <=, >= or >, or when doing arithmetic operations on the answers.

4.6.2.2 toBoolean

`toBoolean` is a method that converts the variable into a Boolean (`true`/`false`). It can be used to check if a question has been answered.

```
f(qID).toBoolean()
```

If the respondent has answered the question with question ID `qID`, the expression will yield `true`, if not, false.

You have already seen `toBoolean` used in Example 7:. Here is another example as well:

Example 9: Checking that a Multi Question has been Answered

A multi question is by default not required in Confirmit unless there is an exclusive element (at least one element in the answer list has the "single punch" property set). This is because when there is no "None of the above" answer alternative, it usually should be allowed to go on without selecting any of the items. (Not selecting any items is equivalent to answering "None of the above").

But sometimes you want to force the respondent to select one or more items. Then you need to provide validation code to check that the question has an answer. If the multi question has question ID *cars* you can use the following validation code:

```
if(!f("cars").toBoolean()) //no answer on cars question
 {
  RaiseError();
  SetErrorMessage(LangIDs.en,"Please select at least one car. ");
}
```

Please select at least one car.

Cars

Please pick the cars that most appeal to you

- ☐ Acura
- ☐ Am General
- ☐ Aston Martin
- ☐ Audi
- ☐ Bentley
- ☐ BMW
- ☐ Buick
- ☐ Cadillac
- ☐ Chevrolet
- ☐ Chrysler
- ☐ Daewoo
- ☐ DeTomaso
- ☐ Dodge
- ☐ Ferrari
- ☐ Ford
- ☐ GMC
- ☐ Honda
- ☐ Hyundai
- ☐ Infiniti
- ☐ Isuzu
- ☐ Jaguar
- ☐ Jeep
- ☐ Kia
- ☐ Lamborghini
- ☐ Land Rover
- ☐ Lexus
- ☐ Lincoln
- ☐ Lotus
- ☐ Mazda
- ☐ Mercedes-Benz
- ☐ Mercury
- ☐ Mini
- ☐ Mitsubishi
- ☐ Nissan
- ☐ Oldsmobile
- ☐ Panoz
- ☐ Pontiac
- ☐ Porsche
- ☐ Qvale
- ☐ Rolls-Royce
- ☐ Saab
- ☐ Saturn
- ☐ Subaru
- ☐ Suzuki
- ☐ Toyota
- ☐ Volkswagen
- ☐ Volvo

5 Operators and Expressions

5.1 Terminology

An **operator** is used to transform one or more values into a single resultant value. The values to which the operator applies are referred to as **operands**. The combination of an operator and its operands is referred to as an **expression**.

For example, in the expression

```
2 + 3
```

the operator + is used to transform the operands 2 and 3 into the resultant value 5.

Some operators result in a value being assigned to a variable, e.g. the assignment operator = in

```
x = 0;
```

Others produce a value that may be used in other expressions, like

```
2 + 3
```

For some operators the order of the operands does not matter:

```
2*3
```

is 6, the same is

```
3*2
```

Other operators give different results for different orderings:

```
3-2
```

is 1.

```
2-3
```

is -1.

Some operators are used only with one operand. They are called **unary** as opposed to **binary** operators, which have two operands. Examples of unary operands: ! (logical not) and – (unary negation, as in –4 (negative numbers)).

You can combine several operators and operands to make complex expressions. To evaluate complex expressions, you have to use rules of order of **precedence** to know which expressions to evaluate first.

In the following we will go through the most common JScript .NET operators. We have left out some operators that seldom are used in Confirmit scripts. For a full overview, please refer to a JScript .NET language reference.

5.2 Arithmetic Operators

Operator	Description
`+`	Addition

`-`	Subtraction or unary negation
`*`	Multiplication
`/`	Division
`%`	Modulus (the remainder of dividing two integers).
`++`	Increment and then return value (or return value and then increment)
`--`	Decrement and then return value (or return value and then decrement)

Some of these may need some explanation:

Modulus gives the remainder of dividing two integers. Examples:

```
7%2
```

gives 1 because 7/2 = 3 with a remainder of 1.

```
6%2
```

gives 0 because 6/3=2 with no remainder.

++ and --:

x++ and ++x both result in 1 being added to the value of x. x-- and --x both results in 1 being subtracted from x. They differ in what value is returned - the value before or after the increment/decrement. ++x and --x return the value after the increment/decrement. x++ and x-- return the value before the increment/decrement.

The distinction between x++ and ++x is illustrated in these two coding examples:

```
x = 3
y = ++x
```

Result: x = 4 and y = 4

```
x = 3
y = x++
```

Result: x = 4 and y = 3

In the first example, x is increased with 1 before the value is returned and stored in y. In the second the value of x is returned and stored in y first, and then x is increased with 1. It is recommended to be very careful when using these operators.

5.3 Logical Operators

Operator	Description
`&&`	logical **and**
`||`	logical **or**
`!`	logical **not**

5.4 Comparison Operators

Operator	Description
`==`	Equal
`===`	Strictly equal (without type conversion)
`!=`	Not equal
`!==`	Strictly not equal (without type conversion)
`<`	Less than
`<=`	Less than or equal
`>`	Greater than
`>=`	Greater than or equal

Expressions with comparison operators evaluate to true or false.

The difference between equal and strictly equal (==/===):
When using == JScript .NET automatically does a type conversion according to the types' order of precedence. For example, in the expression x == y, where x is the string '1' and y is the number 1, the value of y will be converted to the string '1' and the expression will return true. However, in the expression x === y (x strictly equal to y) this conversion will not be done, hence the expression will return false, because the operands has different types.

The same applies to not equal/strictly not equal (!=/!==).

5.5 String Operators

Operator	Description
`+`	String concatenation

Strings can be joined with the string concatenation operator +:

```
"Confirm"+"it"
```

will return

```
"Confirmit"
```

5.6 Assignment Operators

Operator	Description
`=`	Sets the variable on the left of the = operator to the value of the expression on its right.

`+=`	Increments the variable on the left of the += operator by the value of the expression on its right. When used with strings, the value to the right of the += operator is appended to the value of the variable on the left of the += operator.
`-=`	Decrements the variable on the left of the -= operator by the value of the expression on its right.
`*=`	Multiplies the variable on the left of the *= operator by the value of the expression on its right.
`/=`	Divides the variable on the left of the /= operator by the value of the expression on its right.
`%=`	Takes the modulus of the variable on the left of the %= operator using the value of the expression on its right.

The following table explains the operators +=, -=, *=, /= and %=.

Operator	Equivalent to
`x += y`	`x = x+y`
`x -= y`	`x = x-y`
`x *= y`	`x = x*y`
`x /= y`	`x = x/y`
`x %= y`	`x = x%y`

+= can be used for text concatenation as well as arithmetic addition:

```
x = "Confirm";
x += "it";
```

will result in x being equal to "Confirmit".

5.7 new

`new` is used to create new objects. Its used is described in Chapters 7 Arrays, 11 Objects, 13 Working with Sets, and 14 Predefined JScript .NET Objects.

5.8 The Conditional Expression Ternary Operator

This operator takes three operands (that is why it is called ternary):

```
condition ? value1 : value2
```

The first item is a condition that evaluates to either true or false. If the condition evaluates to true, value1 is returned. If the condition evaluates to false, value2 is returned.

Example 10: Response Piping from a Single Question with Other Specify

If you have text substitution in Confirmit and want to pipe in the response to a single question with an item with the "Other" property set, you probably want to pipe in the response in the "Other" text box instead of the answer text (which might be something like "Other, please specify:").

Say you want to pipe in the response to the single question musical where there is one item in the answer list with precode 98 that has the "Other" property set.

English	Precode	Weight	KeepPos	Other
Song & Dance			☐	☐
The Sound of Music			☐	☐
South Pacific			☐	☐
Starlight Express			☐	☐
Sunset Boulevard			☐	☐
Sweeney Todd			☐	☐
Tommy			☐	☐
West Side Story			☐	☐
Other, please specify	98		☐	☑

In the text field of the question or info where you want to pipe in the response to the musical question, you can use this code:

```
^f("musical").get() == "98" ? f("musical_98_other") : f("musical")^
```

`f("musical").get()` will return the precode of the answer to musical (get is described in more detail in Chapter 8.1 get and set). If this is equal to 98, the answer from the "Other" text box should be piped in (`f("musical_98_other")`). If not, use normal text substitution with `f("musical")` so that the answer text will be piped in.

Question ID: i2

English

Title: The chosen musical

Text: Your favorite musical is ^f("musical").get() == "98" ? f("musical_98_other") : f("musical")^

Example 11: Replacing "NO RESPONSE" in Response Piping

When there is no answer to a question, text substitution will give the text "NO RESPONSE" (different texts in other languages than English). So if the respondent for example does not answer on a not required single question income, response piping with

```
Income: ^f("income")^
```

will give this result:

Income: *NO RESPONSE*

If you instead want an empty string (i.e. no text at all) to appear when the question is not answered, use this expression in the question where you want to pipe in the answer:

```
Income: ^f("income").toBoolean() ? f("income") : ""^
```

If there is an answer to income, f("income").toBoolean() will return true and f("income") will be used for text substitution. If there is no answer, an empty string will be returned.

Income:

NOTE: Expressions inside ^ (carets) in question text fields, title fields and answer/scale lists in Confirmit must always return strings. You cannot run JScript .NET statements inside ^'s, however you can call a function that returns a string.

5.9 Coercion

You can set up JScript expressions that involve values of different types without the compiler raising an exception. Instead, one of the types will automatically be changed (coerced) to that of the other before performing the operation. The compiler will allow all coercions unless it can prove that the coercion will always fail. Any coercion that may fail generates a warning at compile time, and many produce a runtime error if the coercion fails.

For example, if you add a string “Confirmit“ and the number 8.5, the number will be converted to a string so that the expression

```
"Confirmit"+8.5
```

will give the string `"Confirmit8.5"`, the concatenation of the strings `"Confirmit"` and `"8.5"`.

In JScript :NET you can also force what type a value should be coerced to by using the target type name as described in Chapter 4.5 Types.

The table below shows what the result will be when using the operator + between values of different types. It illustrates what the order of precedence within the types. If a string is involved, the other types are converted to string. For numbers it will give the number as a string, the Boolean value `false` will give the string `"false"` and `true` the string `"true"`. `null` will be converted to the string `"null"`. An int is converted to a float in an expression with a float value. Boolean true will be converted `1` and `false` to `0` when used in an expression with a numeric value. `null` will be converted to `0` when used in an expression with a numeric value.

row + column	string "12.34"	int 123	float .123	logical true	logical false	null
string "test"	`test12.34`	`test123`	`test0.123`	`testtrue`	`testfalse`	`testnull`
int 123	`12312.34`	`246`	`123.123`	`124`	`123`	`123`
float .123	`0.12312.34`	`123.123`	`0.246`	`1.123`	`0.123`	`0.123`

logical true	true12.34	124	1.123	2	1	1
logical false	false12.34	123	0.123	1	0	0
null	null12.34	123	0.123	1	0	0

Since all values returned from questions in Confirmit are strings, and strings have precedence over all other types, you may have to use some sort of conversion function to change the type before using the value from a question in your scripts.

Exercise 1:

Find the result of this expression.

(As will be shown in the next chapter, expressions within brackets will be calculated separately before the resultant value is concatenated with the other strings, so e.g. 192+15=207 is the resultant value from the first part that is brought into the string.)

```
(192+15)+" is not the same as "+(true+206)+", but this isn't really "+true
```

See answer in Appendix A: Answers to Exercises.

5.10 Operator Precedence

The **precedence** of the operators determines which operators are evaluated before others in complex expressions where several operators are involved.

For operators on the same level of precedence, JScript .NET reads and evaluates expressions from left to right. The table below lists the order of precedence for the operators in JScript .NET, from highest to lowest.

<table>
<tr><th>Precedence</th><th>Operator</th></tr>
<tr><td>1</td><td>()</td></tr>
<tr><td>2</td><td>!, -- , ++, -, new</td></tr>
<tr><td>3</td><td>*, /, %</td></tr>
<tr><td>4</td><td>+, -</td></tr>
<tr><td>5</td><td><, <=, >, >=</td></tr>
<tr><td>6</td><td>==, !=, ===, !==</td></tr>
<tr><td>7</td><td>&&</td></tr>
<tr><td>8</td><td>||</td></tr>
<tr><td>9</td><td>?:</td></tr>
<tr><td>10</td><td>=, +=, -=, *=, /=, %=</td></tr>
</table>

The minus (-) in 2 is the unary negation operator, not subtraction.

Note that parentheses are at the top of the table, so by using parentheses you can always control in what order the expressions are calculated. It is highly recommended to use parentheses, also because it makes the expressions easier to understand.

Here is an example showing how the order of precedence decides how an expression is calculated:

```
3 + 4 * 2
      |
3 +   8
  |
 11
```

Exercise 2:

This is a piece of code from a script. For each one of these assignments, find the value of x, y and z after the line has been executed:

```
x = 4; y=0; z=0;
y = 5*2+1-(x++ == 4 ? 2 : 8)                         //ex. a
z = ((x+4)%3)*900+8                                  //ex. b
y+= --x*8-(z>3 ? z : ++z)                            //ex. c
x = (z == 8 && (z % 3 == 0) || ++y < 6*5) ? --z : ++y   //ex. d
```

See answers in Appendix A: Answers to Exercises.

5.11 Short Circuit Evaluation

In expressions involving the logical operators and (`&&`) and or (`||`), code will execute faster if you use a feature called **short circuit evaluation**. When JScript evaluates a logical expression, it only evaluates as many sub-expressions as required to get a result.

The logical and (`&&`) operator evaluates the left expression passed to it first. If that expression converts to `false`, then the result of the logical and operator cannot be `true` regardless of the value of the right expression. Therefore, the right expression is not evaluated.

Similarly, the logical or operator (`||`) evaluates the left expression first and if it converts to `true`, the right expression is not evaluated.

So to make a script run most efficiently, place the conditions most likely to be `true` first for the logical or operator. For the logical and operator, place the conditions most likely to be `false` first.

This is particularly helpful in expressions involving function calls.

6 Simple Statements

A **statement** is an instruction that makes a program perform some action.

Statements are separated either with a line break or a semicolon (;)

More than one statement may occur on a single line of text, provided that they are separated by semicolons.

A semicolon is not needed between statements that occur on separate lines, however it is recommended to use it anyhow in case the line breaks later are removed.

A statement may be written over multiple lines of text. However, do not use line break inside string constants, since this will give a script error. If you need a line break inside a string you have to use `\n` (see Chapter 4.5.3 Characters and Strings). Also, postfix increment and decrement operators must appear on the same line as their argument (e.g. `i++` and `i--`, see Chapter 5.2 Arithmetic Operators), `continue` and `break` keywords must appear on the same line as their label (see Chapter 9.5 The break Statement and Chapter 9.6 The continue Statement), and `return` must appear on the same line as its expression (see Chapter 10.2.5 The return Statement).

A group of JScript statements surrounded by curly brackets (`{}`) is called a **block**. Statements within a block can generally be treated as a single statement. They are used to group a set of statements inside branches of a condition, or inside loops and functions.

6.1 Declarations

We have used this to declare JScript .NET variables and constants.

```
var variableName {: type} {= expression};
```

or

```
const constantName {: type} = expression;
```

as for example in

```
var x : int = 12;
```

which both identifies the variable x and causes it to be implicitly **declared** as a variable of type int and initialized to the value 12.

Variables and constants may or may not be bound to a specific data type.

```
var x = 12;
```

Variables may not be initialized to any value. Constants have to be initialized.

```
var x;
```

6.2 Assignment Statements

```
variableName assignmentOperator expression;
```

The **assignment** statement updates the value of a variable based upon an assignment operator and an expression (and for `+=`, `-=`, `*=`, `/=` and `%=` also the current value of the variable). For example

```
var x : int = 12;
var y : int;
y = x - 2;
```

assigns the value 10 to y (if x is 12 as in the previous example).

6.3 The if Statement

You can use an `if` statement to control the flow of your code based on a logical expression. If the expression gives the Boolean value true, the statements are executed, whereas if the expression gives the Boolean value false they are skipped. If you use an `else` branch, the statements inside the else branch are executed if the expression gives the value false.

6.3.1 if

```
if( condition )
 {
   statements
 }
```

If the expression inside the parenthesis is true, the statements inside the curly brackets, `{` and `}`, are executed. If the condition is false, then the statements are skipped and execution continues on the next program line.

Note that the parentheses around the condition are required.

If-then conditions are used a lot when making scripts in Confirmit, especially for validation code.

We have used if conditions several times already. See Example 5:, Example 7: and Example 9:.

6.3.2 if-else

```
if( condition )
 {
   statements
 }
else
 {
  statements
 }
```

If the condition inside the parenthesis is `true`, then the first set of statements is executed. If the condition is `false`, then the second set of statements is executed.

Exercise 3:

What values will x and y have after running these two code samples (separately):

```
//Code sample 1
if(x<y)
 {
   x++;
 }
else
 {
  y++;
 }
```

```
//Code sample 2
if(x<y)
 {
   x++;
 }
 y++;
```

1. …when x and y are declared as

```
var x : int = 4;
var y : int = 5;
```

1. …when x and y are declared as

```
var x : int = 5;
var y : int = 4;
```

See answers in Appendix A: Answers to Exercises.

6.3.3 Using Curly Brackets in if Statements

Curly brackets are used to group a set of statements. If you have just one statement that should be executed inside an `if` statement, you may omit the curly brackets `{}`. The following is equivalent to sample 1 above:

```
if(x<y)
 x++;
else
 y++;
```

However, it is easy to do mistakes when you do not use the brackets.

```
if(x<y)
 x++;
 y++;
```

is equivalent to sample 2 in the exercise above, but that might be hard to spot. So it is recommended to always use the curly brackets, both for clarity and also to simplify later modifications like adding another statement within a branch in the `if` statement.

6.4 The switch Statement

The `switch` statement is used when you want different statements to run for different values of a variable. Instead of writing a lot of if statements with conditions for each of the values you want to check, you can use switch. The syntax for the switch statement is like this:

```
switch (expression)
 {
  case value1:
   statements1
   break
   .
   .
   .
  case valuen:
   statementsn
   break
  default:
   statementsx
}
```

You may have more than one statement between each `case` and `break`, and do not need to have curly brackets in front and after them, since for each `case` all the following statements will be executed until the `break` statement.

`switch` evaluates the expression and looks at the values one by one until a match is found in one of the `case` statements. When a match is found, the accompanying statements are executed until a `break` statement is encountered or the `switch` statement ends. If you omit the `break` statement before the next `case`, the following statements will also be executed until a `break` is reached or you get to the end of the `switch` statement.

Use the `default` clause to provide a statement to be executed if none of the values matches the expression.

If no value matches the value of *expression*, and a `default` case is not supplied, no statements are executed.

The `switch` statement above will be equivalent to this code:

```
if(expression == value1)
 {
  statements1
 }
...
```

```
else if(expression == valuen)
 {
  statementsn
 }
else
 {
  statementsx
 }
```

Example 12: *Using switch to set Values for each of the Answer Alternatives on a Single*

Let us say we have a single question where the respondent picks from a list of different concepts. Each of these has a price, and we want to set the price in a hidden open text question based on which of the concepts the respondent has picked. The value in the hidden numeric question may e.g. be used in calculations later in the questionnaire. The single question has question ID "*concept*" and the numeric question has question ID "*price*".

Here are two different ways of scripting this, the left one using if and the right one using switch. As you see, switch gives more compact code that is easier to read:

```
//using if:
if(f("concept")=="1")
 {
   f("price").set(234);
 }
else if(f("concept") == "2")
 {
 f("price").set(249);
 }
else if(f("concept") == "3")
 {
  f("price").set(244);
 }
else if(f("concept") == "4")
 {
  f("price").set(229);
 }
else
 {
  f("price").set(271);
 }
```

```
//using switch:
switch(f("concept").get())
 {
  case "1":
   f("price").set(234);
  break
  case "2":
   f("price").set(249);
  break
  case "3":
   f("price").set(244);
  break
  case "4":
   f("price").set(229);
  break
  default:
   f("price").set(271);
  }
```

7 Arrays

Arrays are a special type of JavaScript object. An object is referred to as a complex data type because it is built from primitive types. The methods and properties of the Array object are covered in Chapter 14.5 The Array Object.

Arrays are objects that are capable of storing a sequence of values in one variable. A single index number (for a one-dimensional array) or several index numbers (for an array of arrays or a multidimensional array) references the data in the array. You can refer to an individual element of an array with the array identifier followed with the array index in square brackets ([]). The indexes are numbered from 0 to 1 less than the number of elements in the array. To refer to the array as a whole, just use the array identifier.

There are two types of arrays in JScript, **JScript arrays (the Array object)**, and **typed arrays**. While the two types of arrays are similar, there are a few differences. JScript arrays and typed arrays can interoperate with each other. Consequently, a JScript Array object can call the methods and properties of any typed array, and typed arrays can call many of the methods and properties of the Array object. Furthermore, functions that accept typed arrays accept Array objects, and vice versa.

7.1 Typed Arrays

In typed arrays (also called native arrays) you define a data type which all of the elements have to comply with. (You can define a typed array of type Object to store data of any type.)

Also, you have to define the number of elements. The only way to change the number of elements is by recreating the array. Trying to access elements outside the range of the indexes, will generate an error. Typed arrays, are **dense**, that is, every index in the allowed range refers to an element.

Use of typed arrays provides type safety and performance improvements compared to JScript Arrays.

7.1.1 Declaring Typed Arrays

Typed arrays can be declared in three different ways. We will look at these by declaring and initializing the same typed array consisting of string values that holds the name of the days of the week in English.

The data type of the elements in a typed array is defined with the name of the data type (see Chapter 4.5 Types) followed by square brackets (`[]`).

One way of declaring a typed array is to list the elements separated by commas within square brackets (this is called an **array literal**) :

```
var weekday : String[] =
["Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","Sunday"]
```

You can also declare an empty array, and then use the `new` operator to define the size of it: Use normal assignments

```
var weekday : String[];
weekday = new String[7];
```

This can also be done in one operation:

```
var weekday : String[7];
```

To assign values to the elements of the array after declaring it as in either of the last two examples, you can use normal assignments, referring to each of the elements of the array:

```
weekday[0] = "Monday";
weekday[1] = "Tuesday";
weekday[2] = "Wednesday";
```

```
weekday[3] = "Thursday";
weekday[4] = "Friday";
weekday[5] = "Saturday";
weekday[6] = "Sunday";
```

It is also possible to declare multidimensional typed arrays and arrays of typed arrays. Refer to a JScript .NET reference for more information.

7.2 JScript Arrays

A JScript Array object provides more flexibility than a typed array. It can store data of any type, which makes it easy to quickly write scripts that use arrays without considering type conflicts. Scripts can dynamically add elements to or remove elements from JScript arrays. To add an array element, assign a value to the element. The `delete` operator can remove elements.

A JScript array is **sparse**. That is, if an array has three elements that are numbered 0, 1, and 2, element 50 can exist without the presence of elements 3 through 49. Each JScript array has a length property that is automatically updated when an element is added. In the previous example, the addition of element 50 causes the value of the length variable to change to 51 rather than to 4.

7.2.1 .Declaring JScript Arrays

We will demonstrate the different ways of declaring and initializing JScript arrays using the same example as in the previous chapters: An array containing the names of the seven weekdays in English.

First, declaring it using an array literal, listing the elements within square brackets, separated by commas:

```
var weekday = =
["Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","Sunday"]
```

Another way is to use the new operator to create a new Array object by using the `new` operator, the object name (Array) and then listing the elements of the array:

```
var weekday = new
Array("Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","Sunday");
```

You can also define an empty array:

```
var weekday = new Array();
```

or an array with size 7:

```
var weekday = new Array(7);
```

and then assign values to the elements:

```
weekday[0] = "Tuesday";
weekday[1] = "Tuesday";
weekday[2] = "Wednesday";
weekday[3] = "Thursday";
weekday[4] = "Friday";
weekday[5] = "Saturday";
weekday[6] = "Sunday";
```

The difference between declaring the array as a JScript array instead of as a typed array as in Chapter 7.1.1 Declaring Typed Arrays is that when it is defined as a typed array, all elements in the array has to be of that type.

Also, in a JScript array you can insert elements of any type, including complex types like other arrays or objects.

Another difference is that in a JScript array you can at any time add new elements, e.g. another weekday. This can be done no matter how the JScript array is defined, even when defined with the length specified, as in the example with `new Array(7)`.

```
weekday[7] = "Lastday";
```

(this would be a good idea to be able to at least occasionally finish projects before deadlines). The array has now 8 elements (0-7). In typed arrays, you cannot add new elements to the array, just modify existing elements.

And finally, in a JScript array you can have "holes". You can e.g. add a 50th element (with index 49):

```
weekday[49] = "Extremelyoverdueday";
```

without defining all the elements in between. The size of the array will be 50, even though elements with index 8,…,48 have not been defined.

7.3 length

The **length** of an array is its size, i.e. the number of elements in it. length is the index of the last element+1.

When we declared the weekday array above, we declared an array of length 7:

```
var weekday = new Array(7);
```

The array could have been declared without specifying length:

```
var weekday = new Array();
```

Initially, the length would then have been 0, but the length would increase as elements were added. And as explained in the previous chapter, the length will always be 1 more than the last index of the array, since the indexes start with 0. This is true even if there are empty elements in it.

It is also possible to declare multidimensional JScript arrays and arrays of JScript arrays. Refer to a JScript .NET reference for more information.

7.4 The length Property

As previously mentioned, JScript .NET arrays are implemented as objects. Objects are a complex data type with collections of data that have properties and may be accessed via methods. A property returns a value that identifies an aspect of the state of an object. Methods are used to read or modify the data contained in an object.

We will get back to objects in general in Chapter 11 Objects and also the methods supported by the array object (Chapter 14.5 The Array Object).

The **length** of an array is a property of an array. Properties of JScript .NET objects are accessed by appending a period and the name of the property to the name of the object:

```
objectName.propertyName
```

The length of an array is determined as follows:

```
arrayName.length
```

So if we used this in the weekday example (before adding "Lastday"),

```
weekday.length
```

would return 7.

8 Methods of the Form Objects

The `f` function we use in Confirmit returns objects that have a lot of different useful methods depending on the question type. These methods can be used to reference the values, titles and answers to Confirmit questions. Some of these return strings and some return arrays of strings. Using these methods, you are able to write all sorts of scripts that access survey variables, modifies them etc. For each of the methods in this chapter, see examples in Chapter 8.10 Applying the Methods on Different Types of Questions.

We will be looking on the f function in Chapter 12 Confirmit's f Function as well.

8.1 get and set

```
f(qID).get()
f(qID).set(value)
```

We have seen these earlier. Used on a question *q1*,

```
f("q1").get()
```

will return the value stored in the database for that object. For an **open text** question this will be the answer the respondent has typed in the text box, for a **single** question this will be the precode of the answer the respondent has selected.

```
f("q1").set("1");
```

With this statement the question *q1* can be set to a specific answer. If it is a single question, it is the precode value that is set, in this case the answer with precode "1". An open text question will be set to the value "1" (string) with this statement.

8.2 label

```
f(qID).label()
```

This method is used to access the question title of a question. This method is available on **open text**, **single, multi** and **grid** questions.

```
f("q1").label()
```

will return the title of the question q1.

8.3 value

```
f(qID).value()
```

`value` can be used on **single** questions to access the precode of the answer to the single question (similar to get on a single question).

8.4 valueLabel

```
f(qID).valueLabel()
```

Use this method on a **single** question to access the label of the answer to the single question in the current language.

8.5 domainValues

```
f(qID).domainValues()
```

This will return an array with all precodes from the answer list of a **single**, **multi** or **grid** question. This is subject to masking, so if a precode mask is used to filter the answer list, this will only return the precodes of the answers that are displayed to the respondent.

8.6 domainLabels

```
f(qID).domainLabels()
```

This method will give an array with all the labels (answer texts) on the question. This is the corresponding answer texts to the precodes returned with domainValues. They will be in the current language. The method will give all possible answer texts from the answer list of a **single**, **multi** or **grid** question. This is also subject to masking.

8.7 categories

```
f(qID).categories()
```

`categories` will return an array with the precodes of the items that have been answered on a **multi** or **grid** question. For a multi question, this will be the precodes of the answers that have been selected, for a grid this will be the precodes of the answers that have an answer. Usually a grid is required, so categories will be equal to domainValues for a grid. But if the grid is not required, they may differ.

8.8 categoryLabels

```
f(qID).categoryLabels()
```

This method will return an array with the labels of the items that have been answered on a **multi** or **grid** question in the current language, i.e. the texts from the answer list corresponding to the precodes returned from categories.

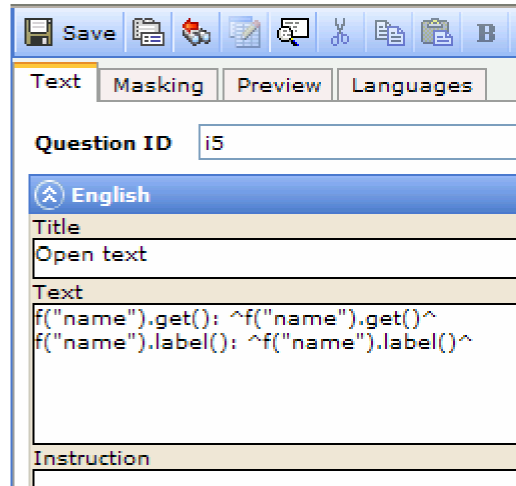
8.9 values

```
f(qID).values()
```

`values` will return an array with the answers stored in the database for a **grid** or **multi** question. For a **grid** question, this will be the precodes of the answers to the questions (from the scale). For a normal **multi** question this will be 0 (not selected) and 1 (selected), but for an ordered, open text or open text numeric multi question it will be the numbers or texts answered for each item in the answer list.

8.10 Applying the Methods on Different Types of Questions

The examples below show these methods applied on an open text, single, multi and grid question. Each question is followed by an info node that is set up with response piping using the different methods.



8.10.1 Open Text Question

Question ID: "name".

Name

Please specify your name

Ole

Powered by Confirmit

Open text

f("name").get(): Ole
f("name").label(): Name

Powered by Confirmit

8.10.2 Single Question

Question ID: "gender". Precodes: M (Male) and F (Female).

Gender

Please spesify your gender:

(•) Male
( ) Female

Powered by Confirmit

<< >>

Single:

f("gender").get(): M
f("gender").label(): Gender
f("gender").value(): M
f("gender").valueLabel(): Male
f("gender").domainValues(): M,F
f("gender").domainLabels(): Male,Female

Powered by Confirmit

<< >>

8.10.3 Multi Question

Question ID: "cars". Precodes: "1","2","3","4","5","6","7".

Cars Test Driven

Which of the following cars have you ever test driven?

- [x] Toyota
- [ ] Honda
- [x] BMW
- [x] Ford
- [ ] Mercedes
- [ ] Saab
- [ ] Volkswagen

Powered by Confirmit

Multi

f("cars").label(): Cars Test Driven
f("cars").values(): 1,0,1,1,0,0,0
f("cars").categories(): 1,3,4
f("cars").categoryLabels(): Toyota,BMW,Ford
f("cars").domainValues(): 1,2,3,4,5,6,7
f("cars").domainLabels():
Toyota,Honda,BMW,Ford,Mercedes,Saab,Volkswagen

Powered by Confirmit

8.10.4 Grid Question

Question ID: "importance". Precodes in answer list: "1","2","3","4","5","6","7". Precode in scale: "1","2","3","4".

Areas of Importance

Please indicate how important the following areas are to you when you are considering purchasing a new car.

	Not important	Somewhat Important	Important	Very Important
Comfort	○	○	○	◉
Design	○	○	◉	○
Color	○	◉	○	○
Safety	○	○	◉	○
Stereo	○	○	○	◉
Horsepower	○	○	○	◉
Options	○	○	◉	○

Powered by Confirmit

Grid

f("importance").label(): Areas of Importance
f("importance").values(): 4,3,2,3,4,4,3
f("importance").categories(): 1,2,3,4,5,6,7
f("importance").categoryLabels():
Comfort,Design,Color,Safety,Stereo,Horsepower,Options
f("importance").domainValues(): 1,2,3,4,5,6,7
f("importance").domainLabels():
Comfort,Design,Color,Safety,Stereo,Horsepower,Options

Powered by Confirmit

8.10.5 Referencing the Elements of a Multi or a Grid

Each item in the answer list of a multi or a grid can be referenced with syntax quite similar to the syntax for referencing the items in an array. But instead of a numeric index starting at zero, you refer to the elements of a multi or a grid by using the precode of the element. The precode is a string. For example

```
f("q1")["3"]
```

will be the item with precode "3" in the multi or grid *q1*.

When you use this syntax to access an element of a grid or a multi, you can use the following methods:

`get`, `set` and `label` for the elements of a **multi** question. `label` will then give the label of the answer in the current language.

`get`, `set`, `label`, `value`, `valueLabel`, `domainValues`, `domainLabels` for the elements of a **grid** question. `label` will then give the label of the answer (from the answer list) in the current language. `value` will return the precode of the answer (from the scale) to that element of the grid. `valueLabel` will return the corresponding answer text (from the scale) in the current language. `domainValues` and `domainLabels` will respectively return all possible precodes from the scale and the corresponding labels (answers) from the scale in the current language.

Using the same multi and grid from the previous example, these examples show methods that can be used on the single elements of a multi or a grid.

8.10.5.1 Element of a Multi Question

Multi:

f("cars")["3"].get(): 1
f("cars")["3"].label(): BMW

Powered by Confirmit

<< >>

8.10.5.2 Element of a Grid Question

Grid:

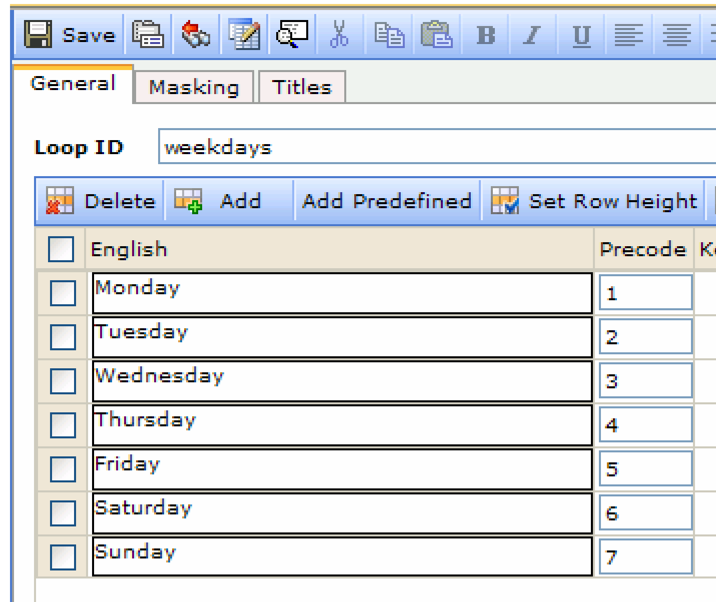
f("importance")["3"].get(): 2
f("importance")["3"].label(): Color
f("importance")["3"].value(): 2
f("importance")["3"].valueLabel(): Somewhat Important
f("importance")["3"].domainValues(): 1,2,3,4
f("importance")["3"].domainLabels(): Not important,Somewhat Important,Important,Very Important

Powered by Confirmit

<< >>

8.10.6 Loops

The same methods may be used on a loop node as on a single question. They return the same items as the single, except for `label`, which will return the loop's ID (not the loop's title, which is only used in reporting).



Here are the results from the first iteration of the loop:

Loop (from inside the loop)

f("weekdays").get(): 1
f("weekdays").label(): weekdays
f("weekdays").value(): 1
f("weekdays").valueLabel(): Monday
f("weekdays").domainValues(): 1,2,3,4,5,6,7
f("weekdays").domainLabels():
Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday

Powered by Confirmit

The first four methods (`get`, `label`, `value` and `valueLabel`) can only be used inside the loop and will give results for the current iteration. `domainValues` and `domainLabels` can be used outside of the loop as well.

8.10.7 3D Grid

None of the methods apply to 3D grids. Instead, refer to the questions contained in the 3D grid by their question IDs.

8.10.8 Implicit Conversion of Arrays to Strings

The arrays in the examples above are all presented in a string context (text substitution with ^) so they are converted into strings. When an array (the result from applying `categories`, `categoryLabels`, `values`, `domainValues` or `domainLabels`) is converted into a string, the elements are presented separated by commas.

Exercise 4:

In this grid, where the default precodes ("1","2","3",…) are used both in answers and in scale, what methods do you have to use on

```
f("importance")["2"]
```

to get the label

a. "Important"

1. "Design"

Areas of Importance

Please indicate how important the following areas are to you when you are considering purchasing a new car.

	Not important	Somewhat Important	Important (a)	Very Important
Comfort (b)	○	○	○	◉
Design	○	○	◉	○
Color	○	◉	○	○
Safety	○	○	◉	○
Stereo	○	○	○	◉
Horsepower	○	○	○	◉
Options	○	○	◉	○

Powered by confirmit

See answers in Appendix A: Answers to Exercises.

8.11 Overview – Methods of Basic Variable Objects in Confirmit

		Strings					Arrays				
		`.get()`	`.set()`	`.label()`	`.value()`	`.valueLabel()`	`.domainValues()`	`.domainLabels()`	`.categories()`	`.categoryLabels()`	`.values()`
Open text question	`f("name")`	✓	✓	✓							
Single question	`f("gender")`	✓	✓	✓	✓	✓	✓	✓			
Multi question	`f("cars")`			✓			✓	✓	✓	✓	✓
Grid question	`f("importance")`			✓			✓	✓	✓	✓	✓
Element of multi question	`f("cars")["3"]`	✓	✓	✓							
Element of grid question	`f("importance")["3"]`	✓	✓	✓	✓	✓	✓	✓			
Loop	`f("weekdays")`	✓	✓	✓	✓	✓	✓	✓			

9 Loop Statements

Loop statements are used to repeat the execution of a set of statements as long as a particular condition is true. There are three types of loop statements in JScript .NET: the `while` statement, the `do while` statement and the `for` statement. The loop statements in JScript .NET are similar to the loop construct in Confirmit, but there is a difference: In a loop in Confirmit, a set of questions is repeated for the items in the loop list (subject to masking). In a JScript .NET loop, a set of statements is repeated until a condition is evaluated to `false`.

9.1 The while Statement

```
while ( condition )
 {
  statements
 }
```

The `while` statement evaluates the condition, and if the condition evaluates to `true`, executes the statements enclosed within brackets. When the condition evaluates to `false`, it transfers control to the statement following the while statement.

Example 13: Validating Sums in a 3D Grid Using a while Loop

In the following example, the `while` statement is used in the validation code of a 3D grid to check that the sum for each row is 24:

Please make sure that the total number of hours for each day equals 24. Currently the sum for Wednesday is 25.

Time spent

Please specify approximately how many hours you spent working, sleeping and on leisure last week:

	Sleep	Work	Leisure
Monday	7	8	9
Tuesday	7	11	6
Wednesday	8	9	8
Thursday	5	12	7
Friday	7	7	10
Saturday	12	2	10
Sunday	10	0	14

Powered by confirmit

>>

The 3D grid contains 3 multi text questions with numeric property: *q2* (sleep), *q3* (work) and *q4* (leisure).

The following code is entered in the validation code field to evaluate the respondent's answers:

```
var precodes = f("q2").domainValues(); //array with all precodes
var i : int = 0;
var correctSum : Boolean = true; //Boolean variable. Will be set to false when a sum
is not correct
while(i<precodes.length && correctSum)
 {
  var code = precodes[i]; //current precode
  //calculate the sum for one row:
  var sum : int =
f("q2")[code].toNumber()+f("q3")[code].toNumber()+f("q4")[code].toNumber();
  if(sum != 24)
   {
    correctSum = false;
   }
  i++;
 }
if(!correctSum)
 {
   RaiseError();
   SetQuestionErrorMessage(LangIDs.en,"Please make sure that the total number of hours
for each day equals 24. Currently the sum for "+f("q2")[code].label()+" is "+sum+".");
 }
```

Let us see what actually happens here when the question is answered as in the picture on page 59.

The first statement,

```
var precodes = f("q2").domainValues();
```

declares an array called `precodes`, which contains all the precodes from the answer list of *q2*. (Since *q2*, *q3* and *q4* are all inside the same 3D grid, they share the same answer list, so we could have used any of these questions to populate the array.) We have used the default precodes ("1","2","3","4","5","6","7") in this answer list, so the first statement declares an array of length 7 where the items have the following values:

```
precodes[0] = "1"
precodes[1] = "2"
precodes[2] = "3"
precodes[3] = "4"
precodes[4] = "5"
precodes[5] = "6"
precodes[6] = "7"
```

The main advantage with using `domainValues` here is that we can do any changes we like to this answer list (including adding/removing items and changing the precodes) without having to change the script.

Then in the next statement we declare a new variable `i` as an integer with the initial value `0` (zero). This will be used to index the array. After that the Boolean `correctSum` is declared with the initial value `true`.

```
var i : int = 0;
var correctSum : Boolean =  true;
```

So, when we enter the `while` loop the first time, the condition will evaluate to `true` because i is `0`, which is less than the length of the array precodes (which is 7), and `correctSum` is `true`.

```
while(i<precodes.length && correctSum)
```

With `i` equal to 🗀, the first statement

```
var code = precodes[i];
```

will set `code` to the first precode, `"1"`.

A variable `sum` is then set in the statement

```
var sum : int =
f("q2")[code].toNumber()+f("q3")[code].toNumber()+f("q4")[code].toNumber();
```

to `7+8+9`, which is `24`. (This is the numbers entered for Monday in the 3D grid, see screenshot above.)

The condition in the following `if` statement will evaluate to `false`, so the statement inside the curly brackets will not be executed. Consequently, `correctSum` will remain `true`:

```
  if(sum != 24)
   {
    correctSum = false;
   }
```

At the end of the loop `i` is increased by 1:

```
  i++;
```

That completes the first iteration of the `while` statement. The next time the condition in the `while` statement is evaluated, `i` is 1. 1 is less than 7, and `correctSum` is still `true`, so the condition still evaluates to `true`, and the statements are to be run a second time.

```
while(i<precodes.length && correctSum)
```

With `i` equal to 1, the first statement

```
var code = precodes[i];
```

will set code to the second precode, "2".

The `sum` will now be `7+11+6` (Tuesday), which is 24.

```
var sum : int =
f("q2")[code].toNumber()+f("q3")[code].toNumber()+f("q4")[code].toNumber();
```

The condition in the `if` statement will again evaluate to `false` (so `correctSum` will not be changed), and `i` is increased by 1 at the end of the loop:

```
  if(sum != 24)
   {
    correctSum = false;
   }
  i++;
```

The third time the condition in the `while` statement is evaluated, i is 2. 2 is less than 7, and `correctSum` is `true`. The condition evaluates to `true`, and the statements are to be run a third time.

```
while(i<precodes.length && correctSum)
```

`i` is now 2, and code will be set to the third precode, "3":

```
var code = precodes[i];
```

The sum will now be `8+9+8` (Wednesday), which is 25.

```
var sum : int =
f("q2")[code].toNumber()+f("q3")[code].toNumber()+f("q4")[code].toNumber();
```

Now the condition in the `if` statement will yield `true` because 25 is not equal to 24. So this time `correctSum` will be set to `false`.

```
  if(sum != 24)
   {
    correctSum = false;
   }
```

Then at the end `i` is increased to 3.

```
  i++;
```

The fourth time the condition in the `while` statement is evaluated, the first part will be `true` because 3 is less than 7, but `correctSum` is now `false`, so the entire expression will give `false`. So this time the execution of the loop will end.

```
while(i<precodes.length && correctSum)
```

Since `correctSum` is now false, the condition in the following `if` statement will evaluate to `true` and the two statements inside will be executed. The statements are calling functions to identify this as an error situation, and to display an error message. We will get back to functions later, also showing a function that would make it easier to calculate the sum.

```
if(!correctSum)
 {
   RaiseError();
   SetQuestionErrorMessage(LangIDs.en,"Please make sure that the total number of hours
for each day equals 24. Currently the sum for "+f("q2")[code].label()+" is "+sum+".");
 }
```

9.2 The do while Statement

The `do while` statement is similar to the `while` statement. The only difference is that the looping condition is checked at the end of the loop, instead of at the beginning. This means that the enclosed statements are executed at least once. The condition is not evaluated before after the statements are executed the first time.

```
do
 {
  statements
 }
while (condition);
```

This may be used for example when the condition uses variables that aren't introduced before inside the loop. In the example below the variable `correctSum` is not introduced before inside the loop.

<u>Example 14:</u> Validating Sums in a 3D Grid Using a do while Loop

Using the same example as in Example 13:, this code would give the same validation of the 3D grid:

```
var precodes = f("q2a").domainValues(); //array with all precodes
var i : int = 0;
do
 {
  var code = precodes[i]; //current precode
  //calculate the sum for one row:
  var sum : int =
f("q2a")[code].toNumber()+f("q3a")[code].toNumber()+f("q4a")[code].toNumber();
  var correctSum : Boolean = (sum == 24); //false if sum not 24, true if sum is 24
  i++;
 }
while(i<precodes.length && correctSum)

if(!correctSum)
 {
   RaiseError();
   SetQuestionErrorMessage(LangIDs.en,"Please make sure that the total number of hours
for each day equals 24. Currently the sum for "+f("q2a")[code].label()+" is
"+sum+".");
 }
```

9.3 The for Statement

The `for` statement is similar to the `while` statement in that it repeatedly executes a set of statements while a condition is `true`. The difference is that the syntax of the `for` loop includes both an initialization statement and an update statement in its syntax:

```
for (initializationStatement; condition; updateStatement)
 {
  statements
 }
```

The initialization statement is executed at the beginning of the loop execution. Then the condition is tested, and if it is `true`, the statements enclosed within brackets are executed. If the condition is `false`, the loop is terminated and the statement following the `for` statement is executed. If the statements enclosed within the brackets of the `for` statement are executed, the update statement is also executed, and then the condition is reevaluated. So the enclosed statements and the update statement are repeatedly executed until the condition becomes `false`.

Typically the initialization statement declares an integer with an initial value, and the update statement increases or decreases this integer. The condition then usually defines the limit.

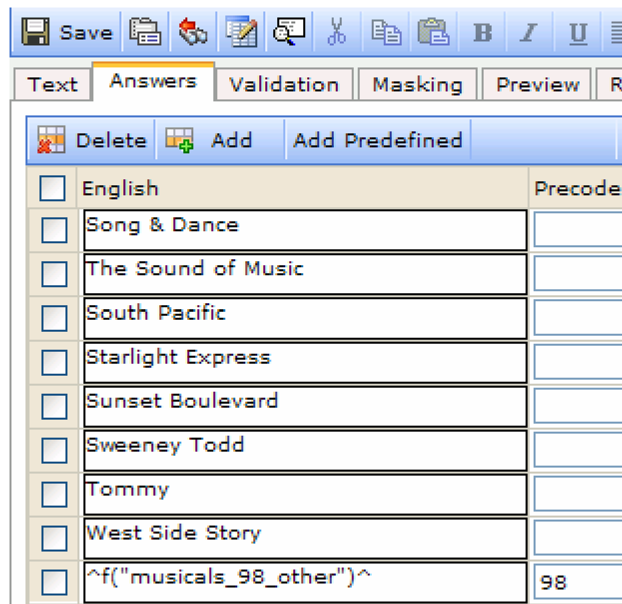
Example 15: *Copying a Multi to do Response Piping with "Other, specify"*

In Example 10: we saw how you could pipe in the answer in the text box of an "other, specify"-option on a single question. However, you cannot use the same approach on a multi question, because there "other, specify" can be answered in combination with any of the other alternatives.

The best way of solving this for a multi question, is to copy the answer on the multi question into another hidden multi question with the same answer list, except that for "other, specify" you pipe in the response from the text box into the answer list instead.

So if you have a multi question *musicals* with an "other, specify"-alternative with precode 98 and the "Other" property set to yes, you can set up a second multi question *musicals_hidden* with the "hidden" property where the answer from the "other, specify" text box is piped in with the syntax

```
^f("musicals_98_other")^
```



Then you need a script node to copy the response from *musicals* into *musicals_hidden*. Copying answers from one question to another hidden question can also be something you need to do for example to have different, shorter answer texts in response piping in a later question, or to have different texts for reporting.

Copying a single or an open text question can easily be done in one statement, for example like this to copy the response from *q1* to *q2*:

```
f("q2").set(f("q1").get());
```

For grid or multi questions it gets a bit more complicated, because they are compounds, i.e. they group more variables into the same form. As previously discussed, elements of a grid or a multi question can be referenced using the syntax

```
f("qID")["precode"]
```

To copy a multi or a grid into another multi or grid, we have to loop through all the elements and copy them one at the time:

```
var precodes = f("musicals").domainValues(); //all precodes
for(var i : int = 0;i<precodes.length;++i) //run through all precodes
 {
  var code = precodes[i]; //current precode
  /* Copy response on this item of musicals question into corresponding item of
musicals_hidden question: */
  f("musicals_hidden")[code].set(f("musicals")[code].get());
 }
```

For this script to work, *musicals* and *musicals_hidden* need to be of the same question type and have exactly the same answer lists (same number of items and same precodes).

Let us say *musicals* and *musicals_hidden* have an answer list with three items with the precodes "1", "2" and "98".

In the statement

```
var precodes = f("musicals").domainValues();
```

the variable *precodes* is declared and set to contain the precodes of musicals, i.e.

```
precodes[0] = "1";
precodes[1] = "2";
precodes[2] = "98";
```

In the loop statement `i` is declared as an integer and initially set to 0 (zero). 0 is less than 3 (the length of the array `precodes`, so the condition yields `true` and the statements inside the brackets are executed.

code is set to "1", the first precode (content of `precodes[0]`):

```
var code = precodes[i];
```

So if we replace `code` with the actual precode in the next statement, we get

```
f("musicals_hidden")["1"].set(f("musicals")["1"].get());
```

and the first element is copied from *musicals* into *musicals_hidden*. Then the update statement is run, and `i` is increased by 1, so the next time the condition is evaluated, i is 1. This is still less than the length of the array (3), so the statements are run once more.

The second time, `code` is set to the second precode, "2", and replacing this in the statement will yield

```
f("musicals_hidden")["2"].set(f("musicals")["2"].get());
```

which copies the second element from *musicals* into *musicals_hidden*. Then `i` is increased again, and the third time the condition is evaluated, `i` is 2. Still the condition yields `true`, and the statements are executed once more, now setting code to "98" and copying the third element from *musicals* into *musicals_hidden*:

```
f("musicals")["98"].set(f("musicals_hidden")["98"].get());
```

Then `i` is increased once more, but now, the value 3 for `i` is not less than the array's length (3), and consequently the loop terminates.

The script is to be placed after the *musicals* question:

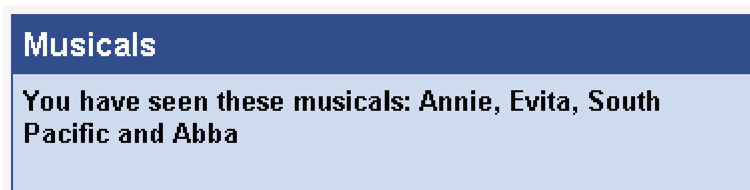
```
musicals - Musicals (Multi)
Copy answer from musicals into mu
musicals_hidden - Musicals (Multi)
i19 - Musicals (Info)
```

In the info node you may now refer to the *musicals_hidden* question in your response piping:

```
You have seen these musicals: ^f("musicals_hidden")^
```

If the respondent for example answers like this:

the result of the piping will be:

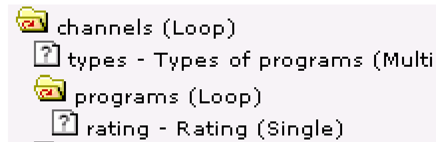


9.4 Loop Nodes in Confirmit

You can build loops in Confirmit if you want to ask the same question(s) for different elements. The loops can be nested. For example, we can have a loop that iterates through some TV channels asking a

question for each TV channel about what kind of programs the respondent watches on the different channels. Then, for those program types there is another loop inside the first one in which they are asked to rate those programs for each channel.

The inner loop is called *programs* and is placed inside the loop called *channels*. In the inner loop we have a single question called *rating* in which we want to ask the respondent about the program types he/she has specified in *types* (the multi question in the outer loop). So the *programs* loop uses the same list of program types as the *types* question, and the loop is filtered with a precode mask based on the types question, `f("types")`. So the loop will only iterate through the program types answered for a specific channel.



In scripts inside the loops you can refer to the questions as usual, e.g. `f("types")`. This will refer to that question in the current iteration of the loop. However, if you need to refer to questions in the loops from scripts outside of the loops, you have to specify which iteration you refer to.

For example, `f("types","1")` refers to the question called *types* in the iteration with precode 1 within this loop. With nested loops (like we have here), you specify the innermost iteration first, so `f("rating","3","1")` refers to the *rating* question from the iteration with precode "3" of the inner loop *programs*. For the outer loop *channels* it is the iteration with precode "1".

9.5 The break Statement

```
break;
```

Sometimes you want to terminate the execution of the loop before it is finished (before the condition in the loop statement evaluates to `false`). Then you may use the `break` statement. The `break` statement will terminate execution of the loop and transfer control to the statement following the loop.

Example 16: Validating Sums in a 3D Grid Using a for Loop and break

The example we used for the `while` loop can be programmed using a `for` loop and a `break` like this:

```
var precodes = f("q2b").domainValues(); //array with all precodes
for(var i : int = 0; i<precodes.length; i++)
 {
  var code = precodes[i]; //current precode
  //calculate the sum for one row:
  var sum : int =
f("q2b")[code].toNumber()+f("q3b")[code].toNumber()+f("q4b")[code].toNumber();
  if(sum != 24)
   {
   RaiseError();
   SetQuestionErrorMessage(LangIDs.en,"Please make sure that the total number of hours
for each day equals 24. Currently the sum for "+f("q2b")[code].label()+" is
"+sum+".");
   break; //terminate execution of the loop when an error is found
   }
}
```

When a row for which the sum is not equal to 24 is found, the loop will terminate without going through the last iterations.

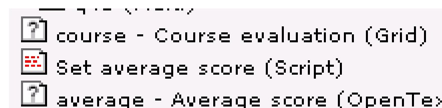
9.6 The continue Statement

```
continue;
```

The `continue` statement is similar to the `break` statement, but instead of transferring control to the statement after the loop it terminates the execution of the current iteration and skips to the next iteration of the loop (after checking the loop condition).

Example 17: *Calculating Averages in a Grid*

We have a grid question *course*, and for reporting we want to calculate an average of the answers to the grid and store it in a hidden open text numeric question called *average*. The grid uses a 5 points scale (and 6 as a value for "Don't know"). Obviously, we want the average to be calculated based on the questions with answers 1-5. The “Don’t know”s (6) should not be included in the calculation.




```
var precodes = f("course").domainValues(); //all precodes of the rows in the grid
var sum : int = 0; //this will hold the sum of the scores
var count : int = 0; //this will hold the number of items
for(var i : int = 0;i<precodes.length;i++)  //iterate through the rows in the grid
 {
  var code = precodes[i]; //current precode
  if(f("course")[code].get() == "6") //don't know
   {
    continue; //skip directly to next iteration
   }
  //here we know that the answer isn't don't know
  sum += f("course")[code].toNumber(); //add current score to sum
  count++; //increase counter with one
 }
if(count>0) //prevent division with zero
 {
  f("average").set(sum/count); //calculate average
 }
else
 {
  f("average").set(null); //set null value if answered "don't know" on all
 }
```

If the respondent has answered "don’t know" (precode 6) on one of the questions, the last two assignment statements are skipped because of the continue, so `sum` and `count` will not be updated for "Don't know" answers.

9.7 The label Statement/Nested Loops

Sometimes you need to specify exactly where execution is to continue after a `break` or a `continue` is used. Especially when you have nested loops (loops in loops). To specify which loop you want the `break` or `continue` to apply to, you can specify a label and refer to that in your `break` or `continue` statement. The names of labels follow the same rules as variable names (see Chapter 4.1 Naming).

Specifying a label:

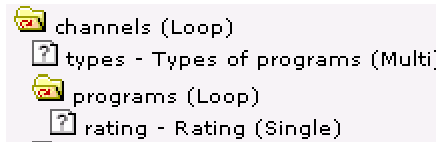
```
label:
```

Referring to a label:

```
break label;
continue label;
```

Example 18: *Calculating Averages on a Single Question in a Loop*

Going back to the example with two nested loops in Confirmit (See Chapter 9.4 Loop Nodes in Confirmit): One iterating through some TV channels, and the other iterating through some types of programs.



For each channel, the respondent answer what type of programs they watch on that channel, and then give a rating for each program type for that channel in the *rating* question. Now we want to write a script that calculates an average of these ratings for each channel. Here is the list of iterations in the channels question:

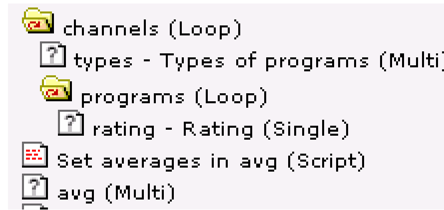
Loop ID: channels

English	Precode
ABC	1
CBS	2
CNN	3
FOX	4
NBC	5

We set up a hidden multi question *avg* with the same list, and set it to be an open text numeric question:

English	Precode
ABC	1
CBS	2
CNN	3
FOX	4
NBC	5

Now, after the loop we insert a script node to calculate these averages.



This script has to run through all the items in the `channels` loop, and for each of them run through all the ratings given in the `programs` loop:

```
var cprecodes = f("channels").domainValues(); //all precodes of the channels loop
Outer:
for(var i : int = 0;i<cprecodes.length;i++) //iterate through channels
 {
  var ccode = cprecodes[i]; //precode of current channel
  //initialize the counter and sum for this channel:
  var sum : int = 0, count : int = 0;

  //the precodes of the program types for this channel:
  var pprecodes = f("types",ccode).categories();

  Inner:
  for(var j : int = 0;j<pprecodes.length;j++) //run through the precodes in the
programs loop
   {
    var pcode = pprecodes[j]; //precode of current program type

    //check rating for this program type for this channel:
    if(f("rating",pcode,ccode).get() == "6")
      {
      continue Inner; //go to next program type
     }
    sum += f("rating",pcode,ccode).toNumber(); //add score to sum
    count++; //increase counter with one
   }

  //set average score for this channel:
  if(count>0)
   {
    f("avg")[ccode].set(sum/count);
   }
  else
   {
    f("avg")[ccode].set(null);
   }
 }
```

This example uses nested loops (loops in loops), i.e. one outer loop that iterates through the *channels* loop and one inner loop that iterates through the answers in the *programs* loop. We have used the labels `Inner` and `Outer` in the script to make sure that it is the iteration of the inner loop that is terminated with the `continue`, not the iteration of the outer loop. (We do not actually refer to the label `Outer` anywhere in the script, but we have included the label for clarity.)

Exercise 5:

Write a script that presets a grid question q2, which has a scale from 1 to 5, so that the first time the respondent comes to that question, all rows in the grid have been set to the middle value 3. Make sure that the values are not reset if the respondent reopens the questionnaire or uses the back button.

See answer in Appendix A: Answers to Exercises.

Example 19: Validation of Ranking when the Number of Elements is Optional

Lets us say we have a multi question with the ordered property checked off. The default validation of such a question is that it is required to rank all items. Here is a script that checks ranking only for

answers entered, allowing the respondent to only rank just the ones he/she wants to, not necessarily all. The items ranked must be ranked 1,2,3,etc., i.e. consecutive integers starting at 1.

In properties of question *ranking* (a multi) both "open text" and "numeric" are selected instead of "ordered", as well as selecting "not required". Then default validation will make sure that the respondent gets an error message if answering anything but numbers. You could also select the ordered property, but turn off the default ranking validation when generating WI.

To check the ranking, you can use this script in the question's validation code field:

```
var error : Boolean = false;
var answers = f("ranking").categories();
 //an array with precodes of the items which has been ranked

var ranks = new Array(answers.length+1);
 /* The ranks array will have true for an item if the rank index has been assigned
    to an element. Element with index 0 in this array will not be used */

for(var i : int = 0;i<answers.length;i++)
 {
  var x = answers[i]; //precode of the item
  var y : int = parseInt(f("ranking")[x].get(),10); //value (rank) assigned to that
item
  /* check that rank y has not already been given to another element and
     that the rank is greater than 0 and less than the total number of ranked elements
  */
  if(ranks[y] != true && y>0 && y <= answers.length)
   {
    ranks[y] = true; //valid rank, this rank is set to true in the ranks array
   }
  else
   {
   /* either the same rank has been given to two elements, or there is a rank outside
      valid range (1,...,number of ranked items) */
    error = true;
    break; //terminate the loop
   }
 }

if(error)
 {
  RaiseError();
  SetQuestionErrorMessage(LangIDs.en,"Please check you ranking. You have to use
consecutive numbers starting at 1 and can not give two elements the same rank.");
 }
```

10 Functions

Usually, when using scripts you reuse a lot of code. Instead of copying the entire code, you can define a function with that code and call it when you need it.

Functions combine several operations under one name. This lets you streamline your code. You can write out a set of statements and define the block of statements as a function and give it a name. Then the entire block of statements can be executed by calling the function and passing in any information the function needs. If a function is given a name that describes what it does, it will be easier to read and understand the code. It will hide details and make your scripts more modularized.

You pass information to a function by enclosing the information in parentheses after the name of the function. Pieces of information that are passed to a function are called **arguments** or **parameters**. Some functions do not take any arguments at all while others take one or more arguments. In some functions, the number of arguments depends on how you are using the function.

A function call is a statement used to invoke a function. Use the function name followed by parentheses containing the arguments, if any, to do a function call:

```
FunctionName(p1,p2,...,pn)
```

You always have to use the parentheses, even if a function contains no arguments:

```
FunctionName()
```

A function may or may not return a value.

10.1 Built-in Functions in Confirmit

Let us start by looking on the functions provided in Confirmit.

10.1.1 Arithmetic Functions

10.1.1.1 Sum

```
Sum( arguments )
```

`Sum` returns the sum of its arguments. Valid argument types are numbers, strings, arrays or any kind of object. Use comma to separate the arguments.

Example 20: Validating Sum on a Multi Numeric Question

Here is an example of a multi question with the numeric property where you ask respondents to apply percentages that should sum up to 100. You can use the "Auto sum" property to have the sum calculated and displayed directly below the respondent's answers.

Your numbers add up to 130. Please make sure that the numbers add up to 100.

Please indicate the percentage of soft drink purchases your household would do on each of these soft drinks. Answer with integers between 0 and 100 and make sure that the sum is 100.

7 Up	10
Cherry Coke	0
Coca-Cola	25
Coca-Cola Classic	10
Diet Coke	25
Dr Pepper	0
Fanta	0
Pepsi One	15
Pepsi-Cola	20
Schweppes	10
Sprite	15
TAB	0
TAB Clear	0
Wild Cherry Pepsi-Cola	0
=	130

The question id of the multi question is *percentage.* In properties precision is set to 3, scale to 0, and lower and upper limit to 0 and 100, so that the system provided validation makes sure that only integers between 0 and 100 is allowed. To validate that the answer is equal to 100, you can use the following code in the validation code field of the question:

```
var sum : int = Sum(f("percentage").values());

if(sum!=100)
{
RaiseError();
SetQuestionErrorMessage(LangIDs.en, "Your numbers add up to " + sum + ". Please make
sure that the numbers add up to 100.")
}
```

Example 21: *Validating Sums in a 3D Grid with the Sum Function*

Let us go back to the example with the validation script that checked that the respondent answered a total of 24 hours for each day (see Example 13:, Example 14: and Example 16:).

Please make sure that the total number of hours for each day equals 24. Currently the sum for Wednesday is 25.

Time spent

Please specify approximately how many hours you spent working, sleeping and on leisure last week:

	Sleep	Work	Leisure
Monday	7	8	9
Tuesday	7	11	6
Wednesday	8	9	8
Thursday	5	12	7
Friday	7	7	10
Saturday	12	2	10
Sunday	10	0	14

Powered by confirmit

>>

In the code of that example we used the expression

```
sum = f("q2")[code].toNumber()+f("q3")[code].toNumber()+f("q4")[code].toNumber();
```

to add up the number of hours. Instead we could use the *Sum* function, like this

```
sum = Sum(f("q2")[code],f("q3")[code],f("q4")[code]);
```

Notice that the Sum function automatically converts the values to numeric, so we do not need to use the `toNumber` method.

Then the entire validation code will look like this, if we use the solution with the `while` loop:

```
var precodes = f("q2").domainValues(); //array with all precodes
var i : int = 0;
var correctSum : Boolean = true; //Boolean variable. Will be set to false when a sum
is not correct
while(i<precodes.length && correctSum)
 {
  var code = precodes[i]; //current precode
  //calculate the sum for one row:
  var sum : int = Sum(f("q2")[code],f("q3")[code],f("q4")[code]);
  if(sum != 24)
   {
    correctSum = false;
   }
  i++;
 }
if(!correctSum)
 {
   RaiseError();
   SetQuestionErrorMessage(LangIDs.en,"Please make sure that the total number of hours
for each day equals 24. Currently the sum for "+f("q2")[code].label()+" is "+sum+".");
 }
```

10.1.1.2 Count

```
Count( arguments )
```

Count returns the number of arguments. Valid argument types are numbers, strings, arrays or any kind of object. Use comma to separate the arguments. An array will be split into its elements, so used on an array `Count` will return the number of elements in the array.

Example 22: Finding the Number of Answers on Three Multi Questions

We have three multi questions *officesa*, *officesb* and *officesc*. This could for example be a question in an employee survey for an international company with offices all around the world, where we want to split the list of offices and introduce sub-headers to display it like this:

Which of these offices have you been in contact with?

USA

- [ ] New York
- [ ] Los Angeles
- [ ] Chicago

Europe

- [ ] London
- [ ] Paris
- [ ] Rome

Far East

- [ ] Kuala Lumpur
- [ ] Singapore
- [ ] Sydney
- [ ] Hong Kong

If you want to find the number of items answered on all three questions combined, for example for use in a condition to ask some further question(s) only if the respondent has answered more than one office, you can use the `Count` function:

```
Count(f("officesa").categories(),f("officesb").categories(),f("officesc").categories()
) > 1
```

If the respondent has picked 2 offices on q1, 1 office on *q2* and 3 offices on *q3*, the result of this function call will be the value 6 (2+1+3).

10.1.1.3 Average

```
Average( arguments )
```

`Average` returns the average (mean) of its arguments. Valid argument types are numbers, strings, arrays or any kind of object. Use comma to separate the arguments.

Example 23: Calculating Averages on 3 Numeric Multi Questions in a 3D Grid

Again referring to the example with the hours and the weekdays, let us say we want to set three hidden numeric questions sleep, work and leisure with the daily average for each of them.

We use the `Average` function:

```
f("sleep").set(Average(f("q2").values()))
f("work").set(Average(f("q3").values()))
f("leisure").set(Average(f("q4").values()))
```

(The 3D grid consisted of tree multi questions with numeric property, *q2* for sleep, *q3* for work and *q4* for leisure). The hidden question sleep for instance, will here be set to 8 since 56/7 is 8 (see Example 20: above).

10.1.1.4 Max and Min

```
Max( arguments )
Min( arguments )
```

`Max` returns the maximum (the largest value) of its arguments. `Min` returns the minimum (the smallest value) of its arguments. Valid argument types are numbers, strings, arrays or any kind of object. Use comma to separate the arguments.

Example 24: Finding Maximum and Minimum Values on Numeric Multi Questions

Once again, referring to the "hours and weekdays" in Example 20:, let us say we want the maximum number of work hours in the week (stored in a hidden question *maxwork*) and the minimum number of hours sleep in the week (stored in a hidden question *minsleep*):

```
f("maxwork").set(Max(f("q3").values()))
f("minsleep").set(Min(f("q2").values()))
```

In our example (see Example 20:), *maxwork* will get the value 12 and *minsleep* the value 5.

10.1.2 Range

10.1.2.1 InRange and InRangeExcl

```
InRange(arg,min,max)
```

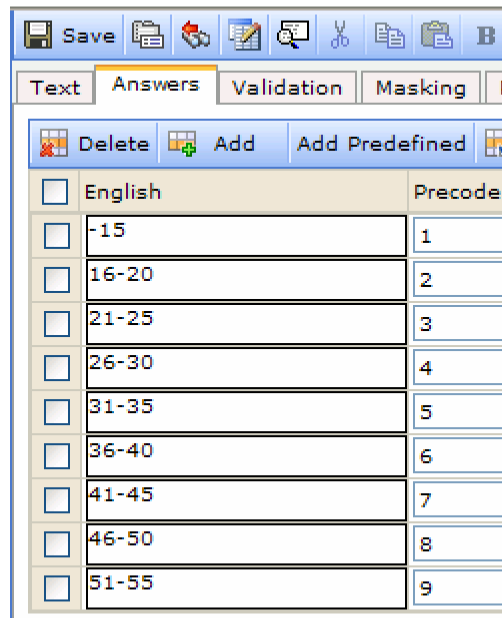
`InRange` returns true if *`arg`* is within the range [*`min`*, *`max`*] (inclusive).

```
InRange(arg,min,max)
```

`InRangeExcl` returns true if *`arg`* is within the range (*`min`*, *`max`*) (exclusive).

Example 25: Building a Condition on a Range of Precodes

Often you need your conditions to work for several precodes within a range. E.g. say you have an *age* question where this is a part of the answer list:



Let us say you want some questions only to go to people from 26 to 50 years. Then you could use a condition with an expression like this:

```
f("age").toNumber() >= 4 && f("age").toNumber() <= 8
```

but it is easier to use the `InRange` function:

```
InRange(f("age"),4,8)
```

10.1.3 Context Information

10.1.3.1 CurrentForm

```
CurrentForm()
```

`CurrentForm` is used in validation code (or in functions called from validation code) and returns the question ID (string) of the question currently being validated. From Confirmit 9.0 this is also true for questions inside 3D grids: Used inside the validation code field of one of the questions inside the 3D grid, it will give the question id of that question instead of the id of the 3D grid itself.

You can use this to write generic code that can be reused without having to change the question ID.

Example 26: Making a Multi Question Required (Generic Code)

As explained in Example 9:; a multi question is by default not required in Confirmit, unless there is an exclusive element (i.e. at least one element in the answer list has the "single punch" property set). This is because when there is no "None of the above" answer alternative, it should usually be allowed to go on without selecting any of the items.

Sometimes you want to force the respondent to select one or more items. Then you need to provide a validation code. This code is probably something you may want to reuse in a lot of questions, so by using `CurrentForm` you can reuse it without having to change the question ID:

```
if(!f(CurrentForm()).toBoolean()) //no answer on current question
 {
  RaiseError();
  SetErrorMessage(LangIDs.en,"Please select at least one.");
}
```

10.1.3.2 CurrentID and CurrentSID

In Confirmit a single respondent is identified with a combination of *respid* (respondent ID) and *sid* (security ID). The respondent ID is a number (starting on 1 for the first respondent in a new survey database). The security ID is a string consisting of 8 random upper case characters. This means that there are 26^8 (more than 200 billion) possible combinations for the security ID.

With the correct combination of *respid* and *sid* you can get access to a survey for one particular respondent. This is used in the links that are generated when you do email sending in limited surveys, e.g.:

http://survey.confirmit.com/wix/<project number>.aspx?r=1&s=JWUDIKLS

(If you are not using Confirmit on FIRM's ASP servers in US, but FIRM's Euro ASP nor have a server installation of Confirmit, you of course have to replace survey.confirmit.com. This goes for all the examples with survey links below.)

There are two functions provided to access *respid* and *sid*:

```
CurrentID()
```

`CurrentID` returns the respondent ID (*respid*).

```
CurrentSID()
```

`CurrentSID` returns the security ID (*sid*).

10.1.3.3 CurrentLang

```
CurrentLang()
```

`CurrentLang` returns the language code of the current language used (e.g. 9 for English). This can be used e.g. in conditions if you want different routing for different languages. You can also use it to set hidden questions to use in reporting if you want to report on the different languages.

10.1.3.4 CurrentPID

```
CurrentPID()
```

`CurrentPID` returns the Confirmit project number of the survey (a number prefixed by p). This can e.g. be set in a hidden variable to be used in data exports if you export from several surveys into a common format. It can also be used when constructing links as in the previous examples.

Example 27: Building a Cryptic URL to be displayed in an info node

Even the respondents that answer open surveys (e.g. pop-ups) get a *respid* and a *sid*. (This can be seen in the source of the interview page as hidden forms called *r* and *s*).

So if you want respondents to an open survey like for example a pop-up survey to be able to quit in the middle of a survey, close their browser and then continue later, they need to have the correct *respid* and *sid* to be able to enter their own survey with previous answers intact, or else they will just start on a new survey.

To achieve this one solution is to display the url including the respondent's *respid* and *sid* in the interview, instructing them to copy that exact url if they want to stop answering and instead continue later.

If you want to leave the survey and continue later, please copy this url and use it to reenter:
http://survey.confirmit.com/wi/p71470843/i.asp?r=3&s=GUYCSWCW&l=9

To make the code as generic as possible you can use `CurrentPID` to insert the project number. `CurrentLang` can be used to include language code as well, if it is a multi-lingual survey. If it is not, the *l* parameter is not necessary.

You can then build the entire url like this in the text area of an info node:

```
http://survey.confirmit.com/wi/^CurrentPID()^/i.asp?r=^CurrentID()^&s=^CurrentSID()^&l
=^CurrentLang()^
```

(If you are not using FIRM's ASP servers, replace survey.confirmit.com with the domain of your Confirmit interview servers.)

The same functions can be used to email a personal link to a respondent, as in Example 38:.

10.1.3.5 GetRespondentValue and SetRespondentValue

Sometimes you may want be able to modify values from the respondent list. Through the "background variable" property on Confirmit questions values can be fetched from the respondent list and set in the survey data, but you can not use these to update the respondent list; it is one way only.

One scenario where it would be beneficial to update the respondent list is if you have a survey which the respondents continuously update, and you want to keep sending new reminders to the respondents. If they change their email address in the survey, you would like the respondent list updated with the new email address so that new reminders are sent to the new email address.

```
GetRespondentValue(fieldname)
```

will return the value uploaded in the *`fieldname`* (string) column in the respondent list for the current respondent. If the *`fieldname`* does not exist, `null` will be returned.

```
SetRespondentValue(fieldname,value)
```

will set *`fieldname`* (string) in the respondent list to *`value`* (string) for the current respondent. If *`fieldname`* does not exist, nothing will happen. There are three system fields that can not be updated: `rid`, `sid` and `rowguid`. Attempting to update any of these will give a script error.

10.1.3.6 InterviewStart, InterviewEnd, SetInterviewStart and SetInterviewEnd

When a respondent enters a Confirmit interview, a SQL variable called *interview_start* is set with the exact time and date. When the respondent reaches a stop node or the end of the interview, a SQL variable called *interview_end* is set. These variables can be used in reporting and are included in data exports. Reporting on time series is for example based on *interview_start*. Note that if a respondent re-enters the interview, *interview_start* is re-set, and similarly, if she or he reaches a stop node or the end of the interview after re-entering a completed interview, the variable *interview_end* is also re-set.

Two functions can be used to get the values of these timestamps in scripts

```
InterviewStart()
```

`InterviewStart` returns the respondent's interview start time.

```
InterviewEnd()
```

`InterviewEnd` returns the respondent's interview end time.

We will get back to examples on how to use these functions in Chapter 14.1.3.1 InterviewStart and InterviewEnd where they are used together with the Date object.

There are also two functions available to set these timestamps. `SetInterviewStart` can be used e.g. when you want interview start to be set after the screening questions of a survey instead of at the beginning of the survey. `SetInterviewEnd` can be used to set the interview end timestamp when a

stop node is never reached because of a redirect at the end of the survey (see Chapter 10.1.8.2 Redirect).

```
SetInterviewStart()
```

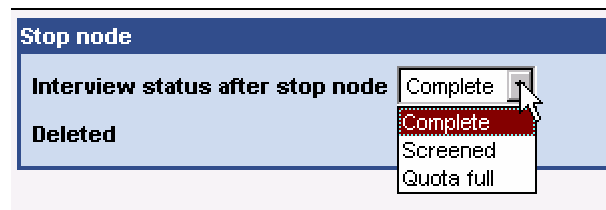
`SetInterviewStart` sets the respondent’s interview start time to current server time and date.

```
SetInterviewEnd()
```

`SetInterviewEnd` sets the respondent's interview end time to current server time and date.

10.1.3.7 GetStatus and SetStatus

When the respondent reaches a stop node, *status* is set according to the status defined for that stop node. When the respondent reaches the end of the interview, *status* is set to "complete".



Currently Confirmit recognizes the following status settings:

Status	Code
Complete	`complete`
Screened	`screened`
Quota Full	`quotafull`
Error	`error`

If status is `null`, it means that the interview is incomplete. The "error" status is set if the interview terminates because of an error in a script.

There are two functions that operate on status:

```
GetStatus()
```

`GetStatus` returns the current interview status.

```
SetStatus(status)
```

`SetStatus` can be used in a script to set the interview status, e.g. if you want respondents who have answered the questionnaire up to a certain point to count as complete interviews, even though they do not answer all the remaining questions. *`status`* is a string with the status value you want to set, i.e. "complete", "screened" or "quotafull". `SetStatus` is very often used in combination with the Redirect function (see Chapter 10.1.8.2 Redirect).

The `SetStatus` function will just set the interview status, not the *interview_end* time stamp. i*nterview_end* is just set when the end of the interview or a stop node is reached.

Please note that the usage of redirects, quotas, status-screened or other solutions with the principle purpose of avoiding the survey "complete" status being reached by a respondent having offered a reasonable amount of responses, is prohibited and will be regarded as an attempt to avoid transaction fee obligations to FIRM.

Example 28: Setting Interview Status Before End of Survey

If you want to set the status of the interview to "complete" before the last questions (which e.g. could be questions about personal details on where to send incentives etc.), you can include a script node with the following code where you want the status to be set:

```
SetStatus("complete");
```

If you do this, even respondents that do not answer those last questions with personal details will still count as completes for reporting etc.

10.1.3.8 Forward

```
Forward()
```

`Forward` returns `true` if the respondent is moving forward through the questionnaire (has clicked on the forward button), and `false` if the respondent is moving backward (has clicked on the back button). This can be used for code that you only want to execute when the respondent moves in a particular direction.

There is no point in using this function in validation code, because validation code is only run when the respondent moves forward in the questionnaire.

10.1.3.9 IsInRdgMode

```
IsInRdgMode()
```

`IsInRdgMode` returns `true` if the scripts are executed during a Random Data Generator run, `false` otherwise. You may use this function to prevent some script code from being run when testing the surveys with the RDG – for example to prevent a SendEmail script to run when doing automated testing.

10.1.4 Browser Information

10.1.4.1 BrowserType and BrowserVersion

```
BrowserType()
```

`BrowserType` returns the browser type (e.g. "IE" for Internet Explorer), as detected by the MSWC.BrowserType component.

```
BrowserVersion()
```

`BrowserVersion` returns the browser version (e.g. "6.0"), as detected by the MSWC.BrowserType component.

You can use these two functions to set hidden variables to report on what browser versions your respondents use:

Example 29: Recording the Respondent's Browser Type and Version

To set browser type and version in the hidden open text questions *browsertype* and *browserversion* you can use this code in a script node:

```
f("browsertype").set(BrowserType());
f("browserversion").set(BrowserVersion());
```

10.1.4.2 RequestIP

```
RequestIP()
```

Returns the IP address of the respondent as a string on quad IP form. This can for example be used to set the IP address of a respondent in an open text question.

NB! Recording the respondents' IP addresses may be in conflict with privacy of the respondents. Responsibility for ensuring respondents' privacy resides with the Confirmit clients. FIRM recommends following ESOMAR guidelines, see **http://www.esomar.org/esomar/show/id=104178.**

If you have an open survey, but want to limit the respondents from answering more than once, you might want to record the IP address and remove responses from equal IP addresses. However, this probably will not give the desired effect, because

- respondents inside a firewall will have the same IP address
- internet providers may use dynamic IP addresses

So to make sure that each respondent answers only once, the best option is to upload a respondent list and email links with r and s, or to upload username and password and use a login page in the survey.

One useful way of using `RequestIP` is to use it in combination with the `IsNet` function to screen respondents from a particular domain. See Chapter 10.1.7.4 IsNet and Example 36:.

10.1.5 Quota check

10.1.5.1 qf

The function `qf` is used to check if a quota is full.

```
qf(quotaName)
```

`qf` will check if the quota *`quotaName`* is full with the current respondent's answers on the questions the quota is based on and return `true` if it is full and `false` if it is not.

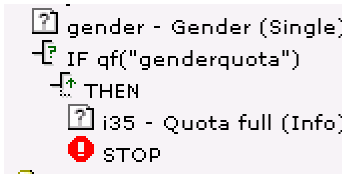
Please note that the usage of redirects, quotas, status-screened or other solutions with the principle purpose of avoiding the survey "complete" status being reached by a respondent having offered a reasonable amount of responses, is prohibited and will be regarded as an attempt to avoid transaction fee obligations to FIRM.

Example 30: Quota Check

If a quota *genderQuota* is based on a question *gender*, and the quota is full for males and not full for females,

```
qf("genderQuota")
```

will return `true` if the respondent has answered "Male" on the *gender* question, and `false` if the respondent has answered "Female" on the *gender* question.



Example 31: *Presetting a Quota Question to Check Several Quotas*

Sometimes you have quotas based on a brand list, where the respondent may qualify for several of the quotas, but you want to pick only one of the brands the respondent has chosen and ask a set of questions for that brand only. Out of the brands the respondent has chosen, there should be picked a brand where the quota is not full yet. To check this, we have to try to set the quota question and use `qf` to check the quota until we find a brand where the quota is not full.

Let us say there is a multi question *brands*, and from this question one of the brands answered should be picked, if the quota is not full for that brand. We set up a hidden single question *chosen_brand* that should hold this brand. The quotas will be set up based on this question in the quota *`brandsquota`*.

The script will try to set *chosen_brand* to a brand chosen in *brands*, check the quota and continue to next brand if the quota is full. If the quota is not full, the current brand will be used.

```
var answers = f("brands").categories(); //precodes of all brands selected
for(var i : int = 0;i<answers.length;i++) //iterate through the precodes
 {
  var code = answers[i]; //current precode
  f("chosen_brand").set(code); //try to preset this precode
  if(!qf("brandsquota")) //check if quota is not full for this code
   {
    break; //if quota is not full, keep this brand
   }
 }
```

After this script *chosen_brand* will either be set to a brand where the quota is not full, or if the quota is full for all the brands *chosen_brand* will be set to the last one. Typically the interview should terminate if the quotas are full for all the brands answered, so after the script there should be a normal quota check in a condition, and then an info and a stop node with *quotafull* status in the then-branch. The expression in the condition should be like this:

```
qf("brandsquota")
```

10.1.6 Panel

10.1.6.1 isUsernameTaken

In Panels, the unique identifier for a panelist is the panelist's username. The `IsUsernameTaken` function is used in panelist recruitment surveys to check if any previous respondents have selected the same username.

```
isUsernameTaken(userName)
```

`isUsernameTaken` returns `true` if there already is a panelist in the sample with the provided *`userName`* (string), `false` otherwise.

It is used in the validation code of the *username* question (mandatory) in a panel survey:

```
if(isUsernameTaken(f("username").get()))
{
   SetQuestionErrorMessage(LangIDs.en,"User name taken. Please choose another user
name.");
   f("username").set("");
   RaiseError();
}
```

The function can also be used in the same way in ordinary projects, to check that the value entered in a question with question id *username* is unique. If the function is used in ordinary projects, the *indexed* property should be set on the *username* question to improve performance.

10.1.6.2 isEmailTaken

The `isEmailTaken` function is used in panelist recruitment surveys in the same way as `isUsernameTaken`; to check if any previous respondents have selected the same email address.

```
isEmailTaken(email)
```

`isEmailTaken` returns `true` if there already is a panelist in the sample with the provided *`email`* (string), `false` otherwise.

The function can also be used in the same way in ordinary projects, to check that the value entered in a question with question id *email* is unique. If the function is used in ordinary projects, the *indexed* property should be set on the *email* question to improve performance.

10.1.7 Classification Functions

10.1.7.1 IsNumeric and IsInteger

```
IsNumeric( argument )
```

`IsNumeric` returns true if the argument (string) is numeric.

```
IsInteger( argument )
```

`IsInteger` returns true if the argument (string) is an integer.

Example 32: Checking that a Response in a Multi Open Text Question is an Integer

If you have a multi question details with open text property set, you can have text boxes for "name", "title", "address", "age" etc. If we want integers only to be allowed for "age", you have to provide validation code for that specific row in the multi question. So, if "age" has precode 4 in the multi question *`details`*, we can use this validation code:

```
if(!IsInteger(f("details")["4"])) //Age not integer
 {
  RaiseError();
  SetQuestionErrorMessage(LangIDs.en,"Please use integers only for \"age\".");
 }
```

(You could use `InRange` as well if you want to check that the value answered is in a reasonable range for age). See Chapter 10.1.2.1 InRange and InRangeExcl.

10.1.7.2 IsDateFmt and IsDate

```
IsDateFmt( argument,format )
```

determines whether the provided *`argument`* (string) is a valid date according to *`format`* (string).

Within a date format, the following character sequences have special significance:

`Y`	Year, four digits.
`YY`	Year, two digits.
`YYYY`	Year, four digits.
`M`	Month number, one or two digits.
`MM`	Month number, exactly two digits.
`D`	Day number, one or two digits.
`DD`	Day number, exactly two digits.

All other characters are treated as separators.

All parts are optional. If omitted, then the system date is used to determine month and year and the number 1 is used for the day.

```
IsDate( day,month,year )
```

`IsDate` determines whether the provided *`year`*, *`month`* and *`day`* combination (all strings) constitutes a valid Gregorian date.

The *`month`* and *`year`* arguments are optional. If they are not provided then the current month and/or year is used.

For both `IsDateFmt` and `IsDate` century interpolation for 2-digit years is handled with a break-point value of 10: values between 0-10 are interpreted as the years 2000-2010, while 11-99 is treated as 1911-1999.

`IsDateFmt` and `IsDate` return an object with the properties `day`, `month` and `year` if the date is valid and in the correct format, `null` otherwise. A general description of objects will follow in Chapter 11 Objects, and Chapter 14.1.3.2 IsDateFmt and IsDate has examples where the object returned from `IsDateFmt` and `IsDate` and the properties `day`, `month` and `year` are used.

Both `IsDateFmt` and `IsDate` can be used in conditions to check for valid dates. If the date is valid and in the correct format, an object is returned. Used in a condition this will be interpreted as `true`. If the date is invalid or in wrong format, `null` will be returned. Used in a condition this will be interpreted as `false`.

Example 33: Validating Date Format of Open Text Date Question

If you have an open text question `date1` and want the respondent to answer in a particular format, you can use `IsDateFmt`. `IsDateFmt` will both check that the format is correct, and that the date is a valid date.

Date 1

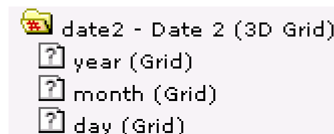
Please enter a date on the format YYYY-MM-DD:

Here is a script you can use in the validation code of such a question:

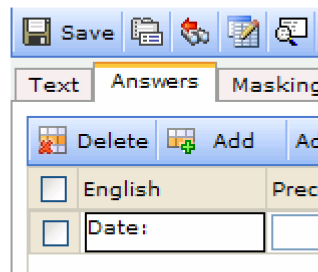
```
if(!IsDateFmt(f("date1").get(),"YYYY-MM-DD"))
 {
  RaiseError();
  SetQuestionErrorMessage(LangIDs.en,"Invalid date. Please correct. Use the format
YYYY-MM-DD");
 }
```

Example 34: Validating Date with Dropdowns for the Date Parts

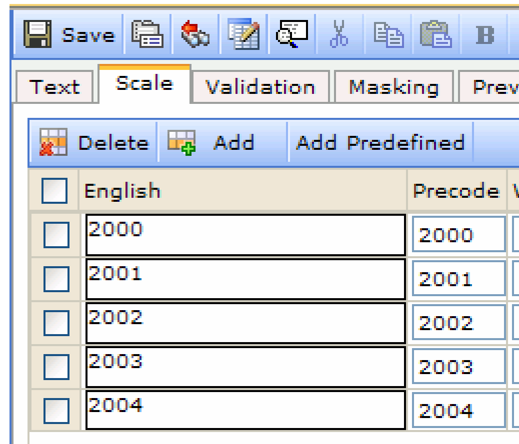
You can also set up date in three questions with dropdowns: One for year, one for month and one for day. To get these on the same line, you can place 3 grid questions in a 3D grid question:

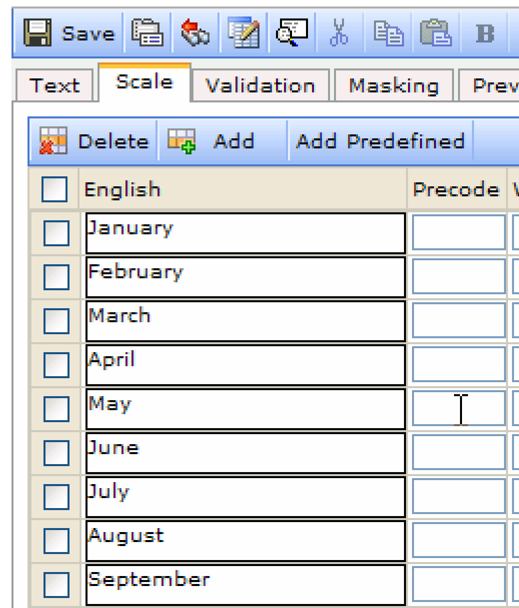


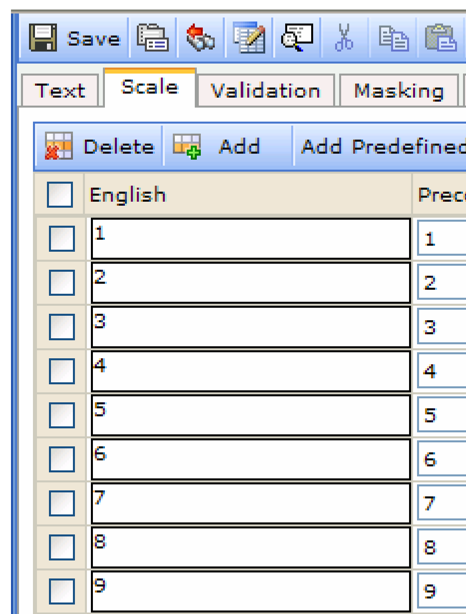
The answer list of the 3D grid will just have one item:



The answer list of the grids will be years, months and dates. It is important that you define precodes that are equal to the number of the years, months (1-12) and dates (1-31):







The question will look like this:

Here is a script you can use in the validation code of such a question, using the `IsDate` function with the different date parts (the grid questions) as arguments:

```
if(!IsDate(f("day")["1"].get(),f("month")["1"].get(),f("year")["1"].get()))
 {
  RaiseError();
  SetQuestionErrorMessage(LangIDs.en,"Invalid date. Please correct.");
 }
```

10.1.7.3 IsEmail

```
IsEmail( argument )
```

`IsEmail` checks whether argument has the format of a valid email address. Returns true if it has, false otherwise.

Note: the function does not attempt any name look-ups or address verification, it just checks that the format is valid (i.e. includes an @ etc.).

Example 35: Validation of Email Address Format

An open text question *email* is used to collect the respondent's email address. To check that the answer is a valid email address, use the following code in the validation code field:

```
if(!IsEmail(f(CurrentForm())))
 {
  RaiseError();
  SetQuestionErrorMessage(LangIDs.en,"Please provide a valid email address.");
 }
```

10.1.7.4 IsNet

```
IsNet(quadIP,quadNet,quadMask)
```

Determines whether an IP address belongs to an Internet network.

Argument:	Description
`quadIP`	The IP address, on quad form (X.X.X.X)
`quadNet`	The network number, on quad form.
`quadMask`	An optional mask, on quad form, if sub netting is used.

If no mask is provided then the number of bits that constitute the network part of the address is determined from the address class type. In this case the function returns true if the IP address' network number is identical to the supplied network number and it has a non-zero local address. Otherwise it returns false

If a mask is provided then the function returns `true` if the IP address' network and subnet parts are identical to the one supplied and the host number part is non-zero. Otherwise it returns `false`.

You can use the `RequestIP` function to get the respondent's IP address. See Chapter 10.1.4.2 RequestIP.

Example 36: Excluding Respondents from Specific Networks

Let us say we want to exclude respondents with network number "192.168.18.0", subnet mask "255.255.255.0" from taking a specific survey (e.g. because they belong to the company that is running the survey).

Then you may place a condition in the beginning of the questionnaire with the following expression to screen those respondents:

```
IsNet(RequestIP(), "192.168.18.0", "255.255.255.0")
```

10.1.8 General Utilities

10.1.8.1 SendMail

```
SendMail(from,to,subject,body,cc,bcc,mailformat,bodyformat,codepage)
```

`SendMail` can be used to send an email from inside a survey.

Arguments (all strings):

Argument	Description
from	Message sender
to	Recipient's address
subject	Message subject line
body	Message body
cc *	Carbon Copy
bcc *	Blind Carbon Copy

*mailformat**	0 – MIME 1 – Text
*bodyformat**	0 – HTML 1 – Plain text
*codepage**	The codepage the mail is to be sent with. 1252 is Western European (default value), 65001 is Unicode.

*) optional

The last 5 arguments are optional, but if you include any of them, you have to include all arguments that precede them. For example, if you do not want a carbon copy (*cc*), just use an empty string (`""`) for that argument.

If you set the *codepage* to something else than 1252, the default value, you must use 0 for *mailformat* (MIME). This also applies if you want to send the email as HTML. If you don't set the *mailformat* to MIME, the email will be sent as plain text

See Chapter Appendix D: Codepage for codepage values.

If your mail text includes Unicode characters and you send a plain text mail, you may use the `fromCharCode` method, see Chapter 14.3.5.2 Building a String from a Number of Unicode Characters.

Example 37: Send Confirmation Email at the End of a Survey

Here is an example of a script placed at the end of a pop-up survey to thank the respondent for participating in the survey. The respondent has provided her or his email address in an open text question called email:

```
//build body of email:
var body : String = "You have just completed a survey powered by Confirmit.\n\n";
body += "We would like to thank you very much for your contribution.\n\n";
body += "Best Regards\n\n";
body += "Future Information Research Management - FIRM\n";

//send mail:
SendMail("interviewer@confirmit.com",f("email"),"Thank
you!",body,"","interviewer@confirmit.com");
```

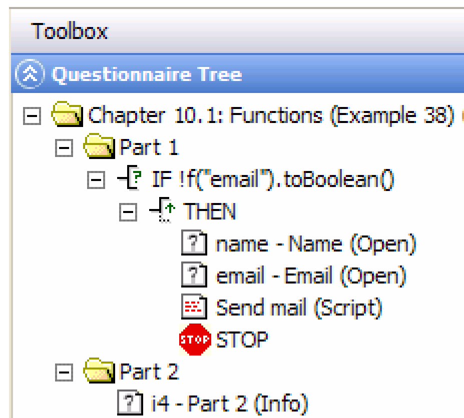
The email will be sent to the respondent with a blind copy also to "info@confirmit.com", but without the respondent seeing it since the info address is in the bcc field. You have to include an empty string ("") for the cc argument if you do not want to cc anybody, but want to bcc like in this example. If you need more than one email address in either to, cc or bcc, separate the addresses with semi colons (;) inside the string, e.g. "support@confirmit.com;consulting@confirmit.com".

Example 38: Invitation Email to a Different Part of the Same Survey

We want a survey to start as a pop-up survey where the respondent registers email (with validation of the email address) and name. Based on that, an email is to be sent to the email address with the url to the rest of the survey, and the pop-up survey is to terminate.

When the respondent opens the url in the mail, he or she should get a page with a text that includes his or her name.

Build a questionnaire like this:



The first condition checks if there is an answer to the *email* question. The *email* question is required, so if it has no answer the respondent should get the first part of the survey.

Then *name* and *email* questions follow. In the *email* question, use the following validation code:

```
if(!IsEmail(f(CurrentForm())))
 {
  RaiseError();
  SetQuestionErrorMessage(LangIDs.en,"Please provide a valid email address.");
 }
```

After the *email* question there is a script node, which does the emailing (change "survey.confirmit.com" if you are not using FIRM's ASP):

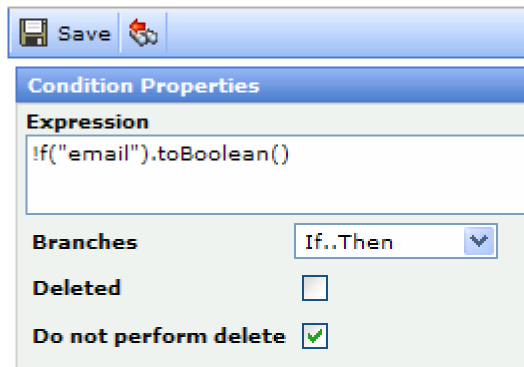
```
var body : String = "Thank you for registering.\n\n";
body += "Here is the url to your follow up survey:\n";
body +=
"http://survey.confirmit.com/wi/"+CurrentPID()+"/i.asp?r="+CurrentID()+"&s="+CurrentSI
D()+"&l="+CurrentLang()+"\n";

SendMail("interviewer@confirmit.com",f("email"),"Follow up",body);
```

Then there is a stop node to end the first part of the interview. When the respondent re-opens the interview with the url, he/she is brought into the same interview with the data from the first part. So since there now is an answer to the email question, the first part will be skipped and the respondent will be brought to the info node (*i4*).

However, by default, when a condition in Confirmit evaluates to `false`, the interview engine removes all answers to questions in the THEN-branch. This ensures that you get consistent data in your surveys and do not have to do data cleaning. But here, this would mean that the answers to `name` and *email* would be deleted, and that is not what we want – we are going to use `name` in part 2 of the interview.

Fortunately there is a property on conditions that can be used in situations like these: "Do not perform delete". When this property is checked, the interview engine will never remove answers in any of the branches (THEN/ELSE).



To pipe in the name in the first info node (*i4*), use this code:

```
^f("name")^
```

10.1.8.2 Redirect

```
Redirect(url,{noexit})
```

`Redirect` can be used to redirect the respondents to a different site (*`url`*).

If it is a redirect at the end of the survey, it is important that the interview status is also set, using the `SetStatus` function (see Chapter 10.1.3.7 GetStatus and SetStatus). Please note that the usage of redirects, quotas, status-screened or other solutions with the principle purpose of avoiding the survey "complete" status being reached by a respondent having offered a reasonable amount of responses, is prohibited and will be regarded as an attempt to avoid transaction fee obligations to FIRM.

`noexit` is a Boolean. It is optional, and is used when you want the respondents to be able to reenter the survey, for example if you redirect out of the survey to another *`url`*, and then at some point the respondents are redirected back to the Confirmit survey again (with r (respid) and s (sid)). This is especially important if the survey is set up without "Allow user to modify answers after the interview is complete". With *`noexit`* as `true`, the respondent will be allowed to reenter the survey. With *`noexit`* as `false` (default), the respondent will not be allowed reentry to the survey.

Example 39: Redirect to Another Site Before the End Page

If you want to send the respondent to another site (here: Google) before he or she reaches the default end-of-survey page in Confirmit, you can use a script like this:

```
SetStatus("complete");
SetInterviewEnd();
Redirect("http://www.google.com",false);
```

`Redirect` is used in combination with `SetStatus` and `SetInterviewEnd` so that the respondent gets the correct status and end time stamp before being redirected out of the survey.

10.2 Creating Your Own Functions

You can create your own functions and use them where needed. It is recommended to use functions

- whenever there are pieces of code that it is likely that you will reuse several times in the questionnaire
- when you need to refer to values (often computed expressions) that are not stored in survey variables

- to group statements that perform a well-defined task, to make the code easier to read, and to make your code more modularized

Functions can be called from anywhere within the questionnaire.

Type annotation in a function specifies a required type for function arguments, a required type for returned data, or a required type for both. If you do not type annotate the parameters of a function, the parameters will be of type Object. Likewise, if the return type for a function is not specified, the compiler will infer the appropriate return type.

Using type annotation for function parameters helps ensure that a function will only accept data that it can process. Declaring a return type explicitly for a function improves code readability since the type of data that the function will return is immediately clear.

10.2.1 Defining functions

Before you can use a function, you have to define it. In Confirmit, functions are usually defined in script nodes. The syntax of a function definition is as follows:

```
function FunctionName(p1{ : type},p2{ : type},...,pn{ : type}) : type
 {
   <statements>
 }
```

The function name is used to refer to the function in function calls. The same naming rules apply to function names as to variable names. The convention for function names is to start them with a capital letter, to distinguish them from variable names. The arguments (p_1,p_2,...) are the names of the variables that receive the values passed to the function. You may define functions without any arguments at all, and functions with a variable number of arguments.

Type annotation on parameters and the function itself (for the return type) is not required.

Example 40: Function to Make a Multi Question Required

In Example 26: we wrote a script that made a multi question without an exclusive item required. Instead of having to copy all that code into all multi questions where you want to use it, you can define a function like this:

```
function RequiredMulti()
 {
  if(!f(CurrentForm()).toBoolean())
  {
   RaiseError();
   SetQuestionErrorMessage(LangIDs.en,"Please select one or more items.");
  }
 }
```

The function will be called from the validation code of the multi questions where you want to use it. Since it is called from the validation code of a question, `CurrentForm` will return the correct question ID.

10.2.2 Function Call

As we have seen in numerous examples, to call a function simply use the function name followed by parentheses containing the arguments, if any:

```
FunctionName(p1,p2,...,pn)
```

A function defined in a script node in a Confirmit questionnaire, may be called in all script context, i.e. from other script nodes, conditions, response piping (within ^s) and in precode masks and column masks in the same questionnaire.

The function `RequiredMulti` defined above (Example 40:) can be called from the validation code of a multi question like this:

```
RequiredMulti();
```

When calling a function, make sure that you always include the parentheses and any required arguments. Calling a function without parentheses causes the text of the function to be returned instead of the results of the function.

10.2.3 Functions with a Fixed Number of Arguments

When you define a function with arguments, you define variables with the argument names as variable names. The arguments will hold the values that are passed into the function, and may be used as normal JScript .NET variables inside the function.

Example 41: *Function to Copy a Multi Question*

If you want to copy the answers to one multi question into another multi question you can define a function that takes the question IDs of the two questions as arguments. This function could for example be used to copy responses from a question musicals to a second multi question *musicals_hidden* to get the "other, specify" text in as in Example 15:.

```
function CopyMulti(from,to)
 {
  var precodes = f(from).domainValues();
  for(var i : int = 0;i<precodes.length;i++)
   {
    var code = precodes[i];
    f(to)[code].set(f(from)[code].get());
   }
 }
```

To call the function, simply place the following code in a script node:

```
CopyMulti("musicals","musicals_hidden");
```

Another example where this function could be used is the three office questions from Example 22:, which we used to get sub-headers in the answer list like this:

Which of these offices have you been in contact with?

USA

- [ ] New York
- [ ] Los Angeles
- [ ] Chicago

Europe

- [ ] London
- [ ] Paris
- [ ] Rome

Far East

- [ ] Kuala Lumpur
- [ ] Singapore
- [ ] Sydney
- [ ] Hong Kong

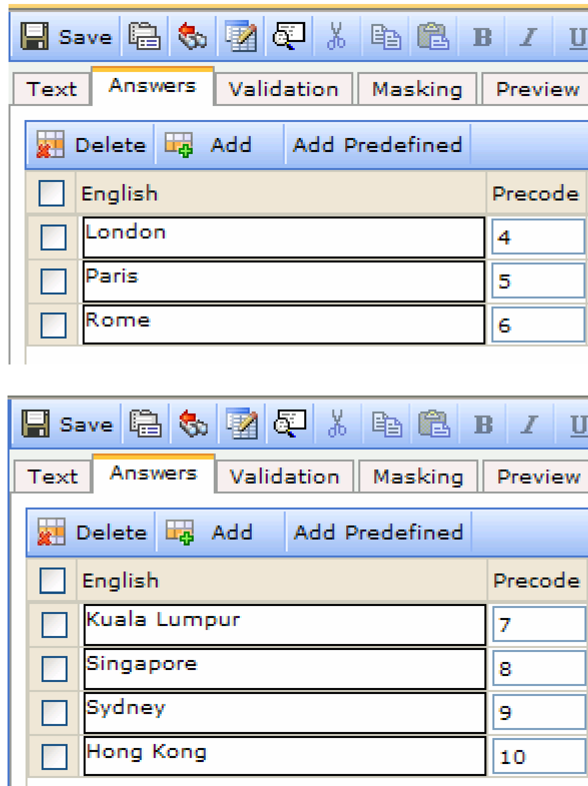
This was set up in three multi questions, *officesa*, *officesb* and *officesc*. However, for reporting you might want to join the answers in one hidden question, `offices`. You can use the `CopyMulti` function to copy the answers into *offices* if the precodes for the items in the answer lists are unique, for example like this:

Save B I U

Text | Answers | Validation | Masking | Preview

Delete | Add | Add Predefined

English	Precode
New York	1
Los Angeles	2
Chicago	3



Then the answer list of the *offices* question can be set up having all of the offices, and the answers can be copied with a script like this using the `CopyMulti` function:

```
CopyMulti("officesa","offices");
CopyMulti("officesb","offices");
CopyMulti("officesc","offices");
```

10.2.4 Functions with a Variable Number of Arguments

Previously in JScript you could define functions that took a variable number of arguments, using the `arguments` array. In JScript .NET fast mode this is not available. If you need a function to take an arbitrary number of arguments, you can use a *parameter array*. This is done by including, as the last element in the argument list, an array that is defined with three periods (…), then the name of the array and then a typed array annotation.

```
function FunctionName(... parArray : type[])
 {
  <statements>
 }
```

this is replaced with the ability to automatically create the arguments array for each function invocation.

```
parArray.length
```

returns the number of arguments in this array. To refer to each argument in this array, use indexing like in all other arrays: 0 for the first argument, 1 for the second, and so on:

```
parArray[index]
```

This makes it possible to write extremely flexible functions, as we have seen examples of in the built-in arithmetic functions in Confirmit; Sum, Count, Average, Max and Min (see Chapter 10.1.1 Arithmetic Functions).

10.2.5 The return Statement

Often you want a function to return a value, either a result from a calculation that has been done in the function, or just a Boolean to indicate whether the function did what it was supposed to. To return a value to the statement that invoked the function, use the return statement:

```
return expression
```

The return statement will terminate a possible loop and send the value to the statement that invoked the function.

Example 42: Returning a Calculated Value from a Function

The following function will return what percentage the number n is out of the base `b`:

```
function Percentage(n : int,b : int) : float
 {
  return (n/b)*100;
 }
```

10.2.6 Local Variables

Since you cannot always predict from where a function will be called, you may face problems if you use variable names that are also used outside of a function.

Let us say that we have a grid q1 inside a loop loop1. We want to check the grid and return true if the value "1" is answered for at least one item in the grid. (1 being the most negative answer). We want to store which iterations there was a "1" answer in the grid in a hidden multi q2. The multi q2 has the same answer list as the loop members list of loop1.

We use the following code to check all the loop iterations. The details of how the checking of the grid q1 for each of the iterations is hidden in a function CheckGridAnswer.

```
var loopPrecodes = f("loop1").domainValues();
for(var i : int = 0;i<loopPrecodes.length;i++)
 {
  var loopCode = loopPrecode[i];
  if(CheckGridAnswer("q1",loopCode,"1"));
   {
    f("q2")[loopCode].set("1");
   }
  else
   {
    f("q2")[loopcode].set("0");
   }
 }
```

Here is the code of the function `CheckGridAnswer`:

```
function CheckGridAnswer(qID,iter,val) : Boolean
 {
  var precodes = f(qID).domainValues();
  for(i=0;i<precodes.length;i++)
   {
    var code = precodes[i];
    if(f(qID,iter)[code].get() == val)
     {
      return true;
     }
   }
  return false;
 }
```

The problem here is that `i` is used both outside and inside the function. Hence, it is a **global variable**. This means that `i=0` before the function call, but the function changes it since the loop in the function

will increase `i` until either an answer with precode "1" is found in the grid, or the loop in the function has checked all the answers in the grid without finding a "1" answer. So after the first function call, `i` will have increased. This will cause some iterations in *loop1* to be skipped, and could also lead to script errors if the number of items in the grid is larger than the number of iterations in the loop. How can we then make sure that the variables used inside the function will not affect variables outside of the function, without having to check that we do not use identical variable names?

The answer is to declare them as **local variables** that exist only within the function and do not affect variables outside of the function.

To declare a local variable, prefix the variable declaration with the keyword `var`.

When used inside a function, this will declare the variable as a local variable – a local version of the variable so that a variable with the same name outside the loop will not be affected.

So, if we rewrite the function definition using local variables, the script will do what we intend:

```
function CheckGridAnswer(qID,iter,val) : Boolean
 {
  var precodes = f(qID).domainValues();
  var i : int;
  for(i=0;i<precodes.length;i++)
   {
    var code = precodes[i];
    if(f(qID,iter)[code].get() == val)
     {
      return true;
     }
   }
  return false;
 }
```

11 Objects

JScript .NET is an **object-oriented language**. We will not elaborate on the object-oriented features in this manual. We will just describe the JScript and Confirmit objects that are relevant for survey programmers, and advice the reader to consult a JScript .NET reference for more in-depth knowledge. All script code in Confirmit is wrapped as functions inside a class. This means that you can not define your own classes in Confirmit scripts.

JScript objects are collections of **properties** and **methods**. A method is a function that is a member of an object. A property is a value or set of values that is a member of an object.

11.1.1 Properties

Properties are used to access the data values contained in an object.

An object's properties are accessed by combining the object's name and its property name:

```
objectName.propertyName
```

We have already seen an example of a property on the `Array` object, a property called `length` for the size of the array (for example, the `weekday` array from previous examples):

```
weekday.length
```

11.1.2 Methods

Methods are functions that are used to perform operations on an object.

You access a method of an object by combining the object's name and the name of a method:

```
objectName.methodName(arguments)
```

Just like in functions, arguments in method calls are optional.

We have already used the basic variable objects of Confirmit returned from the `f` function. We have seen some methods that can be applied on these objects to retrieve information about the questions in a Confirmit survey. For example will

```
f(qID).label()
```

return the title of the question with ID *qID*, and

```
f(qID).get()
```

the precode or answer stored in the database for that question.

We have also seen a method that can be used to set the answer of a question:

```
f(qID).set(code)
```

11.1.3 Constructors: Creating Instances of Objects

Instances of objects of a particular object type are created using an operator called `new`, which you already have seen used when creating an array, for example:

```
var a = new Array();
```

An array is a special object type. The general syntax for creating an instance of an object is like this:

```
variableName = new objectType(arguments);
```

`objectType(arguments)` is called the **constructor** of the object. Some object types have more than one constructor. Constructors differ in the number of arguments they allow.

12 Confirmit's f Function

The f function is probably the most important function available in Confirmit. It is used to access survey variables/questions, and we have already seen numerous examples of how to use it (see Chapter 8 Methods of the Form Objects).

12.1 Calling the f Function

```
f(qID,{loopiter1,loopiter2,...,loopitern})
```

returns a form object of some kind. A **form** is a question or loop node in a Confirmit questionnaire. The argument *qID* is the question ID as defined in the properties of the question.

The rest of the argument(s) ($loopiter_1, loopiter_2, ..., loopiter_n$) are only used for questions inside loops. These arguments are precodes of specific loop iterations, starting with the innermost loop if there are nested loops (loops within loops). The curly brackets {} are used here to indicate that the loop iterations are optional.

Example 43: Referencing a Question in a Loop

Let us say you have a question q1 in a nested loop structure with two loops. The outer loop has iterations with precodes "1" and "2", and the inner loop has iterations with precodes "a" and "b". Then the question q1 is asked four times, and these four instances of q1 can be referenced like this (in the order they were asked) from outside the loop:

```
f("q1","a","1")
f("q1","b","1")
f("q1","a","2")
f("q1","b","2")
```

12.2 Storing the Form Object in a Variable

The form objects can be large data structures with long answer lists and texts. Every call of the f function will result in a new instance being retrieved. If the question has a precode or scale mask, this mask will be evaluated for every call on the `f` function for this question. It is recommended to try to reduce the number of calls on the `f` function, for example by not always applying methods and properties directly on the function as we have been doing in all our examples so far. If you will be calling the `f` function on the same question ID several times in a script (especially if you call the `f` function inside a `for` or `while` loop), the best approach is to call it once and store the object instance in a variable instead:

```
variableName = f(qID,{loopiter1,loopiter2,...,loopitern});
```

Usually the difference is not significant, but it is recommended because it will reduce the execution time of your scripts. The interview pages will then load faster reducing the risk of irritating respondents and reducing server performance.

12.3 Compounds

The `f` function returns in fact different objects for different question types. These objects have some properties and methods in common, but some are specific to questions of a specific type.

Grid and **multi** questions are questions with one or more variables that are defined in the same form. They are called **compounds**. They can be accessed using the `f` function in the normal manner, but the object returned is more complex than objects returned from other question types. You can refer directly to each of their elements using syntax that is similar to that of an array:

```
x[precode]
```

x represents a variable holding an instance of an object returned from calling the `f` function on a compound. *precode* is a string representing the precode from the answer list of the compound. An example is referring to the element with precode `"1"` in a multi question with question ID *q2*:

```
f("q2")["1"]
```

Example 44: Using a Variable instead of Repeated Calls on the f Function

Once more, back to the example with hours and weekdays where we validate the number of hours per day.

Please make sure that the total number of hours for each day equals 24. Currently the sum for Wednesday is 25.

Time spent

Please specify approximately how many hours you spent working, sleeping and on leisure last week:

	Sleep	Work	Leisure
Monday	7	8	9
Tuesday	7	11	6
Wednesday	8	9	8
Thursday	5	12	7
Friday	7	7	10
Saturday	12	2	10
Sunday	10	0	14

Powered by confirmit

>>

In our previous solution for validation of this question (see Example 13:) we looped through the weekdays. For each day, we called the f function for three questions q2, q3 and q4:

```
  sum = Sum(f("q2")[code],f("q3")[code],f("q4")[code]);
```

Now, instead we can define three variables sleep, work and leisure to hold the form objects:

```
var sleep = f("q2");
var work = f("q3");
var leisure = f("q4");
var precodes = sleep.domainValues(); // array with all precodes
var i : int = 0;
var correctSum : Boolean = true;
while(i<precodes.length && correctSum) //iterate through precodes (rows)
 {
  var code = precodes[i]; //current code
  //calculate sum:
  var sum : int = Sum(sleep[code],work[code],leisure[code]);
  if(sum != 24)
   {
    correctSum = false;
   }
```

```
  i++;
 }
if(!correctSum)
 {
   RaiseError();
   SetQuestionErrorMessage(LangIDs.en,"Please make sure that the total number of hours
for each day equals 24. Currently the sum for "+sleep[code].label()+" is "+sum+".");
 }
```

The way the script was initially set up you could have as much as 21 calls (3*7=21) on the f function inside the loop. We have now reduced this to 3 function calls.

12.4 Properties

The form objects can have the following properties, depending on the type of the question (x represents a form object returned from the f function):

```
x.CODED
```

CODED is true if the question referenced is a **coded variable.** If not, CODED is undefined. Coded variables are single questions and loops, and answer list items of a grid or an ordinary multi question.

```
x.OPEN
```

OPEN is true if the question referenced is an **open variable.** If not, OPEN is undefined. Open variables are open text questions, other-specify elements of answer lists and answer list elements of a multi question with the open text and/or ordered property.

```
x.NUMERIC
```

NUMERIC is true if the question referenced is a **numeric variable**. If not, NUMERIC is undefined. Numeric variables are open text questions with the numeric property and answer list elements of a multi question with the numeric property.

```
x.DICHOTOMY
```

DICHOTOMY is true if the question referenced is a **dichotomy.** If not, DICHOTOMY is undefined. The elements of ordinary multi questions (referenced with f(qID)[precode]) are dichotomies.

```
x.COMPOUND
```

COMPOUND is true if the question referenced is a **compound.** If not, COMPOUND is undefined. Multi questions and grids are compounds.

These properties are extremely powerful, because by using them it is possible to write generic scripts that will work no matter what the question type is.

Example 45: Deleting the Content of any Question

This function will remove the answers on any question, no matter what kind of question it is. This could for example be used to automatically remove respondent information at the end of the survey if you ask the respondents if they want to be anonymous or not. An answer is removed by overwriting the current value with null. It will work on open text, single, multi and grid questions, and you use it like this:

```
ClearForm(f("q1"));
```

for a question with question ID q1.

If the question is a compound, the function will remove answers in all variables in the compound. (It will however not automatically remove a value in an "Other, specify" field on a question. To remove the answer on "Other, specify" you have to call the function referring to the "Other, specify" field, e.g. like this:

```
ClearForm(f("q1_98_other"));
```

if the "Other " property is set on the item in the answer list with precode "98".)

Here is the definition of the function:

```
function ClearForm(form)
 {
  if(form.COMPOUND) //form with multiple items
   {
    var fcodes = form.domainValues(); //all precodes in form
    for(var i : int = 0;i<fcodes.length;i++) //iterate through precode
     {
       form[fcodes[i]].set(null); //clear item
     }
   }
  else //form with one item
   {
    form.set(null);
   }
 }
```

Example 46: *Copying the Contents of any Form into Another*

We have previously seen examples of code used to copy a single question and code used to copy a multi question. Here is a function that will copy any type of question. It will return true if the copying succeeds, and false without copying anything if it does not. (If the from and to questions were not the same type of questions).

```
function CopyForm(from,to) : Boolean
 {
  if((from.CODED && to.CODED) || (from.OPEN && to.OPEN) || (from.DICHOTOMY &&
to.DICHOTOMY))
   {
    to.set(from.get());
    return true;
   }
  else if(from.COMPOUND && to.COMPOUND)
   {
    var fcodes = from.domainValues();
    var tcodes = to.domainValues();
    if(fcodes.length == tcodes.length)
     {

      for(var i : int = 0;i<fcodes.length;i++)
       {
        if(fcodes[i].toString()==tcodes[i].toString())
         {
          to[fcodes[i]].set(from[fcodes[i]].get());
         }
        else
         {
          return false;
         }
       }
      return true;
     }
   }
  return false;
 }
```

12.5 Methods

The methods of the form objects are listed below. The methods are described in detail in Chapters 4.6.2 Conversions Methods in Confirmit, 8 Methods of the Form Objects, and 13.3 Methods of the Set Object.

Methods	CODED	OPEN	NUMERIC	DICHOTOMY	COMPOUND
toBoolean	✓	✓	✓	✓	✓
toNumeric	✓	✓	✓	✓	
get	✓	✓	✓	✓	
set	✓	✓	✓	✓	
label	✓	✓	✓	✓	✓
value	✓				
valueLabel	✓				
domainValues	✓				✓
domainLabels	✓				✓
categories					✓
categoryLabels					✓
values					✓
inc	✓				✓
size	✓				✓
union	✓				✓
isect	✓				✓
diff	✓				✓
members	✓				✓

13 Working with Sets

Sets are typically used in precode masks of single, multi, grid questions and loops, and in scale masks of grids. A set consists of a collection of strings that represents precodes. When used in precode and scale masks, the precodes contained in the set are used to determine which answer alternatives to display to the respondent. Only answers corresponding to precodes from the set returned from the expression in the precode or scale mask will be shown.

13.1 Constructor

An instance of the Set object can be created like this:

```
s = new Set();
```

You will also get instances of the Set object returned from the functions `a`, `set`, `nset`, `nnset`, and `Filter`.

13.2 Functions Returning Sets

13.2.1 The a Function

The function `a` returns a set consisting of all possible precodes on a question (i.e. the precodes of all items in the answer list).

```
a(qID)
```

13.2.2 set, nset and nnset

There are three functions you can use to define sets with specific precodes – `set`, `nset` and `nnset`.

```
set(precode1,precode2,...,precoden)
```

will return a set consisting of the precodes listed within the parenthesis. For example

```
set("1","3","4")
```

returns a set consisting of the precodes "1","3" and "4".

```
nset(n)
```

will return a set with precodes "1","2",...,n. n must be an integer greater than 0. For example

```
nset(5)
```

returns a set consisting of the precodes "1","2","3","4" and "5".

```
nnset(m,n)
```

will return a set consisting of precodes from `m` to `n` inclusive, i.e. `m,m+1,...,n-1,n`. For example

```
nnset(6,10)
```

returns a set consisting of the precodes "6","7","8","9" and "10".

13.2.3 The Filter Function

The function `Filter` is used to filter a list based on the matches a prefix has in the answer list.

```
Filter(qID,prefix)
```

It returns a set with the precodes of items in the answer list of question *qID* that starts with the string prefix, regardless of case. If prefix is "c", Filter will return the precodes of the answers "Cadillac", "Chevrolet" and "Chrysler" if they are in the answer list of *qID*.

Example 47: Filtering an Answer List by the First Characters in the Answer

There may be situations where you have a single question with a really long answer list, which you want to filter based on textual input from the respondents. Then you can present an open text question to the respondents asking them to provide the first one or two letters in their answer, and then filter the answer list based on that answer. Say you have a single question *car*, and in front of that an open text question *prefix*. In the prefix question the respondent is asked to type in the first letter in the name of the car. The precode mask of the car question can have this precode mask:

```
Filter("car",f("prefix").get())
```

13.2.4 The f Function

The `f` function can also be used in set context (precode and scale masks). The `f` function does not return a `Set` object, but a form object. However, since it can be used in set context (in precode and scale masks) and has most of the methods of the `Set` object, we choose to discuss it together with the `Set` object. The two set methods not supported by the form object, as indicated in the next chapter, is `add` and `remove`.

```
f(qID {,loopiter1,loopiter2,...,loopitern})
```

The curly brackets {} are used here to indicate that the loop iterations are optional arguments. (See Chapter 12 Confirmit's f Function).

13.3 Methods of the Set Object

In the syntax below, s_1 and s_2 represents instances of set objects or form objects (returned from the `f` function). For `add` and `remove` s_1 represents only an instance of the set object, because these methods are not defined for form objects.

13.3.1 inc

```
s1.inc(precode1,precode2,...,precode3)
```

`inc` returns `true` if all the precodes listed are contained in the set s_1.

For example, for a multi question `q1`, the expression

```
f("q1").inc("1","2")
```

is `true` if **both** the answer with precode "1" **and** the answer with precode "2" has been selected on *q1*. If you need an expression that is `true` if the answer with precode "1" **or** the answer with precode "2" has been selected, you can use this code:

```
f("q1").inc("1") || f("q1").inc("2")
```

13.3.2 size

```
s1.size()
```

`size` returns the number of elements in the set s_1.

Example 48: Checking the Number of Answers on a Multi Question

Often you want to restrict your respondents from answering more than a given number of answers on a multi question. The following function will do that check, and return true if the number of answers given is within the limit, and false if not:

```
function CheckNumberOfAnswers(limit) : Boolean
 {
  return (f(CurrentForm()).size()<=limit);
 }
```

Call the function like this in the validation code of the multi question (a question *q1* with maximum 3 answers):

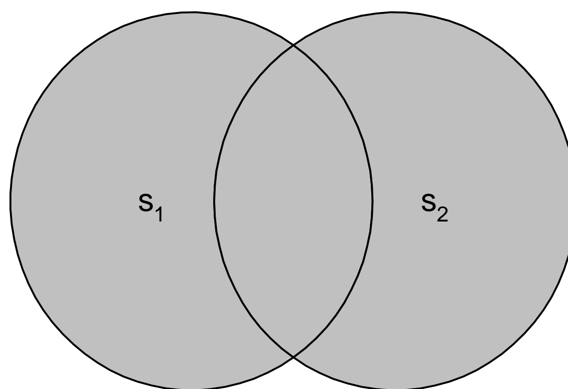
```
if(!CheckNumberOfAnswers(3))
 {
  RaiseError();
  SetQuestionErrorMessage(LangIDs.en,"Please do not select more than 3 items.");
 }
```

13.3.3 union, isect and diff

There are three methods available to do set operations: `union`, `isect` and `diff`. These methods are useful when you need complex precode masks, for example when you want to filter the answer list based on answers to two previous questions. Examples of this are: Showing the items answered on both questions, on any of the questions, on one but not the other etc.

```
s1.union(s2)
```

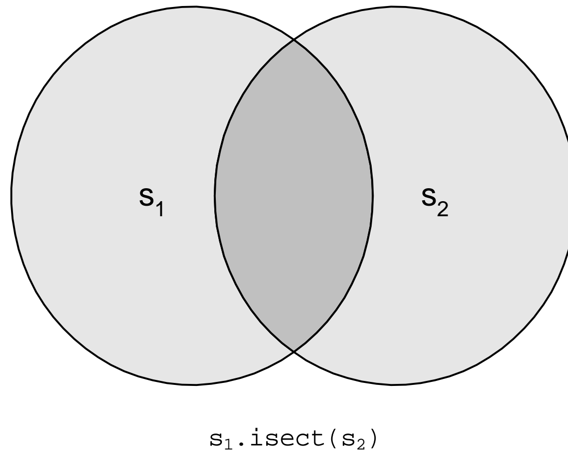
The **union** of two sets s_1 and s_2 is the set obtained by combining the members of both sets. So union will return a set consisting of all elements in s_1 **or** s_2.



s1.union(s2)

```
s1.isect(s2)
```

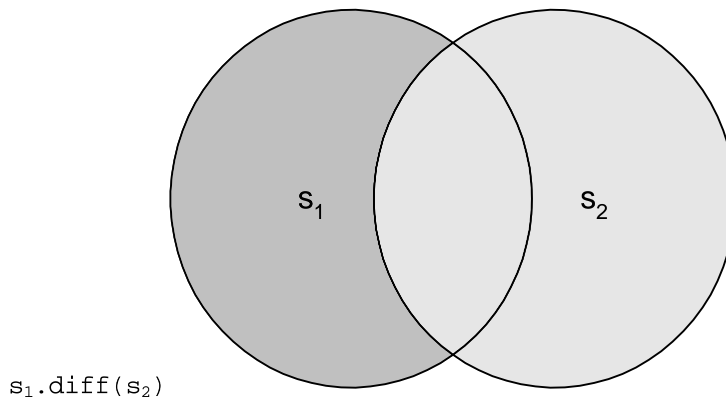
The **intersection** of two sets s_1 and s_2 is the set of elements common to s_1 and s_2. So `isect` will return a set consisting of the elements that are both in s_1 **and** s_2.



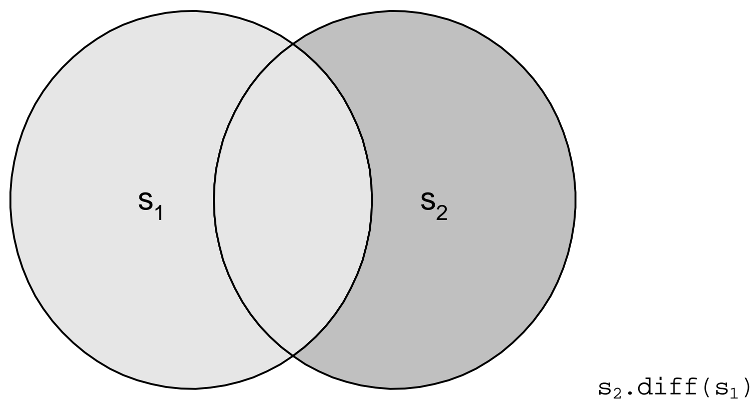
s_1.isect(s_2)

```
s1.diff(s2)
```

The **difference** between two sets s_1 and s_2 will yield a set consisting of the elements that are in the first set, but not in the other. For difference the order of the sets in the expression is significant. s_1.diff(s_2) will return a set consisting of the elements in s_1 that are not in s_2, but s_2.diff(s_1) will return a set consisting of the elements in s_2 that are not in s_1. As the illustrations show, these are two completely different sets.



s_1.diff(s_2)



s_2.diff(s_1)

Example 49: Filtering an Answer List on Items Selected in Two Previous Questions

You have two multi questions *q1* and *q2*, and then a loop where you loop through the same answer list that is used in *q1* and *q2*.

If you want the loop to be filtered so that only the items the respondent answered in both *q1* and *q2*, you can use a precode mask like this on the loop:

```
f("q1").isect(f("q2"))
```

If you want the loop to be filtered so that all the items the respondent answered in either of *q1* or *q2*, you can use a precode mask like this on the loop:

```
f("q1").union(f("q2"))
```

If you want the loop to be filtered so that only items answered in *q1*, but not in *q2*, you can use a precode mask like this:

```
f("q1").diff(f("q2"))
```

Similar, for items answered in *q2* but not in *q1*:

```
f("q2").diff(f("q1"))
```

Example 50: *Filtering Answers Not Selected in a Previous Question*

If you have a multi question *q1* followed by a grid *q2* where you only want the answers not selected in *q1* to be displayed, you can use the a function to get a set with all the precodes in the answer list, and use the diff method to remove the precodes of the answers selected on *q1*. The code for such a precode mask will be like this:

```
a("q2").diff(f("q1"))
```

You have to use the same precodes for corresponding items in the answer lists of the two questions. This can easily be achieved for example by using a predefined list.

Example 51: *Always Including a "Don't know" Answer Alternative*

Often you want to filter the answer list, but you want a "Don't know" alternative always to be included at the bottom of the answer list. Say for example a multi *q1* is followed by a single question *q3* where the answers given to *q1* and "Don't know" should be displayed. It is a good idea to assign a precode to "Don't know" that is different from the other precodes, for example by using a large number like "99" or letters like "DK". We will use "DK" in this example. In the precode mask of *q3* we can use this code:

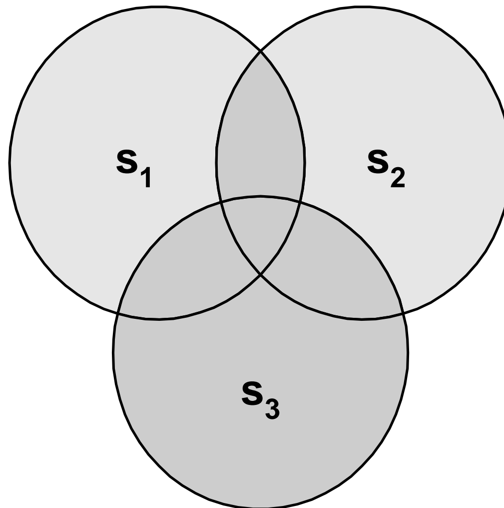
```
f("q1").union(set("DK"))
```

13.3.4 Combining Set Operators

You may build expressions where several set operators are combined. When several set operators are included in the same expression, the expression is evaluated from left to right. So if you for example have three sets s_1, s_2 and s_3, the expression

```
s1.isect(s2).union(s3)
```

will give a set consisting of all precodes either in the sets s1 and s2, or in set s_3:

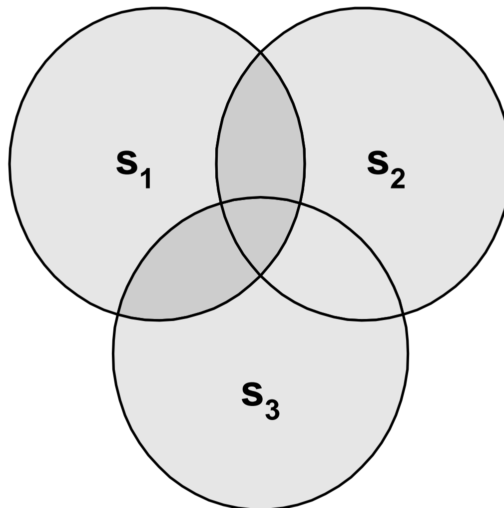


s_1.isect(s_2).union(s_3)

To force a sub-expression to be evaluated before other expressions, you can insert entire expressions inside the parentheses of the set methods. In the expression

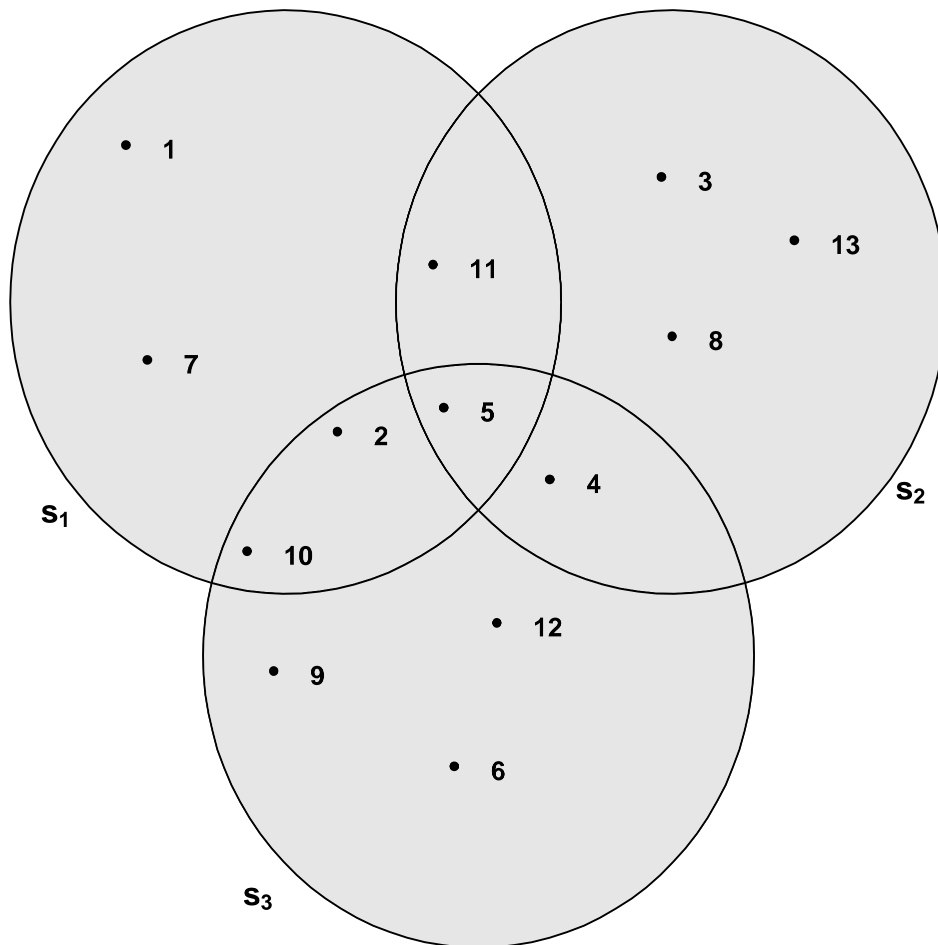
```
s₁.isect(s₂.union(s₃))
```

the sub-expression s_2.union(s_3) will be evaluated first, and then the intersection between the result of this sub-expression and the set s_1 is computed. The result is a set consisting of all precodes both in s_1 and in either of s_2 or s_3:



s_1.isect(s_2.union(s_3))

Exercise 6:



You have three sets: s_1, s_2 and s_3. s_1 consists of the codes `1,2,5,7,10,11`, s_2 consists of the codes `3,4,5,8,11,13` and s_3 consists of the codes `2,4,5,6,9,10,12`, as indicated in the illustration above.

What elements will the sets returned from the following expressions contain?

a. s_1.union(s_3.diff(s_2))

1. s_1.union(s_3).diff(s_2)

1. s_1.isect(s_2.diff(s_3))

1. s_1.isect(s_2).union(s_2.isect(s_3)).union(s_3.isect(s_1))

See answers in Appendix A: Answers to Exercises.

13.3.5 members

```
s1.members()
```

`members` is used to convert a set to an array.

Example 52: *Joining Answers in one Multi Question*

In Example 41: we saw code for copying the answers to the three office questions into one for reporting. The questions where set up to get sub-headers in the answer list like this:

Which of these offices have you been in contact with?

USA

- [ ] New York
- [ ] Los Angeles
- [ ] Chicago

Europe

- [ ] London
- [ ] Paris
- [ ] Rome

Far East

- [ ] Kuala Lumpur
- [ ] Singapore
- [ ] Sydney
- [ ] Hong Kong

This was set up in three multi questions, *officesa*, *officesb* and *officesc*. However, for reporting you might want to join the answers in one hidden question, *offices.*

Here is a solution using the set operations instead of the `CopyMulti` function we defined in that example. It assumes that the survey is set up without back button and not the "Allow user to modify answers after initial submission" setting. Then the respondents cannot change their answers and we do not need to "clean" up any previous answers, and can set just the responses that the respondents have given, with no need to set 0, "not answered", on any items. This script will reduce the execution time slightly because we will now only run through and set the answers that are selected.

As in Example 41:, it is a requirement that there is no overlap in precodes. All items need to have unique precodes.

```
//build a set with all precodes selected in either of the questions
var s = f("officesa").union(f("officesb")).union(f("officesc"));

var precodes = s.members(); //convert the set of precodes to an array

for(var i : int = 0;i<precodes.length;i++) //iterate through those precodes
 {
  var code = precodes[i]; //current precode
  f("offices")[code].set("1"); //set this item to "selected"
 }
```

13.3.5.1 add and remove

To add and remove items from a set, there are two methods available: `add` and `remove`.

```
s1.add(precode);
s1.remove(precode);
```

`add` will add `precode` to the set s_1. If precode is already in the set s_1, the set will not be changed. `remove` will remove `precode` from the set s_1. If precode is not in the set s_1, the set will not be changed.

These methods are not defined for the form objects (returned from the f function), only for the set object.

Example 53: Using a Function to Filter an Answer List Based on the Answers on a Grid

If you have a grid where you give some elements a rating from 1-5, you may for example want the next question to use only the elements that get a score of 4 or 5. This can be achieved with a function like this defined in a script node:

```
function ScoreFilter(qID)
 {
  var form = f(qID);
  var precodes = form.domainValues();
  var s = new Set();
  var i : int;
  for(i=0;i<precodes.length;i++)
   {
    var code = precodes[i];
    if(form[code].get() == "4" || form[code].get() == "5")
     {
      s.add(code);
     }
   }
  return s;
 }
```

This function returns a set with the precodes of the items that has received a score of "4" or "5". In the precode mask where you want to use it you can just call this function with the question ID of the grid (for example *q3*) as argument:

```
ScoreFilter("q3")
```

13.4 User-Defined Functions in Precode or Scale Masks

The previous example used a user-defined function in the precode mask field. This is a convenient way of doing it, but there are a few pitfalls with this approach.

The precode mask is called several times when you go through the questionnaire, not only when the question is displayed. Every time you use the f function on a question with a precode mask, the precode mask is evaluated. So the function in the precode mask of a question will be called every time there is a reference to the question in scripts elsewhere in the survey.

This can lead to two problems:

The time it takes for the respondent to load the survey pages will increase because there are more scripts to execute.

It is easy to make mistakes with global variables. If you forget to use the `var` keyword in the function, the function called from the precode mask may change the values of variables in other scripts if you use the same variable names. Errors like that are hard to identify.

For these reasons, it is recommended to be cautious with the use of functions in precode masks. An alternative approach is to set a hidden multi question and filter on that instead. This hidden multi question can also be useful for reporting purposes.

Example 54: Using a Hidden Multi to Filter an Answer List Based on a Grid

Just as in Example 53:, we have a grid where the elements are given a 1-5 rating, and want the next question to use only the elements that get a score of "4" or "5". Now let us use a script to set a hidden multi question, *scorefilter*, instead of using a function. The grid has question ID *carrating*.

```
var form = f("carrating");
var precodes = form.domainValues();
var i : int;
for(i=0;i<precodes.length;i++)
 {
  var code = precodes[i];
  if(form[code].get() == "4" || form[code].get() == "5")
   {
    f("scorefilter")[code].set("1");
   }
  else
   {
    f("scorefilter")[code].set("0");
   }
 }
```

Then the precode mask uses *scorefilter* instead of the function call:

```
f("scorefilter")
```

14 Predefined JScript .NET Objects

In this chapter we will describe some of the objects provided in JScript .NET. We will present the objects that are most useful for scripting in Confirmit, and the properties and methods that are most frequently used. For descriptions of other JScript .NET Objects, please consult a JScript .NET reference.

14.1 The Date Object

The `Date` object enables basic storage and retrieval of dates and times, and can be used to set timestamps, do calculations of time differences and to validate dates in the respondent's answers.

The `Date` object type of JScript .NET provides a common set of methods for working with dates and times, either the Local Time or Universal Coordinated Time (UTC). Since we are working with scripts that run on the Confirmit servers (server-side), not on the respondent's PC (client-side), **Local Time** means the time on the Confirmit servers you are working on. **Universal Coordinated Time (UTC)** (sometimes also called "Zulu Time") was formerly called Greenwich Mean Time (GMT) and is the mean solar time at the prime meridian (0° longitude).

Date values are stored internally as number of milliseconds since January 1, 1970 UTC. For dates before this date, this will be a negative number. The range of dates that can be represented in an instance of the Date object is approximately 285,616 years on either side of January 1, 1970. This should be sufficient for most of us.

14.1.1 Constructors

You can create a new instance of the `Date` object in three ways:

```
dateObj = new Date();
dateObj = new Date(dateVal);
dateObj = new Date(year, month, date{, hours{, minutes{, seconds{,ms}}}});
```

If you just use the constructor `Date()`, the instance of the `Date` object will be set to current date and time.

If you use `Date(`*`dateVal`*`)` and *`dateVal`* is an integer, *`dateVal`* represents the number of milliseconds since midnight January 1, 1970 Universal Coordinated Time. Negative numbers indicate dates prior to 1970. If *`dateVal`* is a string, *`dateVal`* is parsed according to the rules in the parse method (see description in Chapter 14.1.2.1.1 parse).

Here are two ways of getting an instance of the `Date` Object that is set to midnight 2000 (UTC):

```
d = new Date(946684800000);
d = new Date("Sat, 1 Jan 2000 00:00:00 UTC");
```

When you use `Date(`*`year,month,date`*`{,`*`hours`*`{,`*`minutes`*`{,`*`seconds`*`{,`*`ms`*`}}}})`, the first three arguments are required. *`year`* has to be the full year, for example 1984 (and not just 84). *`month`* is an integer between 0 and 11 (This means that January is 0, December is 11. This is similar to how we have seen that arrays are indexed, where 0 is the first element). *`date`* is an integer between 1 and 31.

The next arguments are optional (that's what the curly brackets indicate), but when one of them is included, all preceding arguments must be supplied. This means that if you specify *`seconds`*, it has to be preceded by *`hours`* and *`minutes`*. *`hour`* is an integer from 0 to 23 (midnight to 11pm). *`minutes`* and *`seconds`* are integers from 0 to 59 and *`milliseconds`* is an integer from 0 to 999. Here is an example:

```
d = new Date(2000,0,1,0,0,0,0);
```

This may or may not be equal to the two examples above using the other constructors. It depends on the server settings on the Confirmit server. The two examples above both use UTC. With this last constructor, the time zone cannot be specified. The time zone of the Confirmit server will be used.

If an argument is greater than its range or is a negative number, other stored values are modified accordingly. So 150 seconds will be redefined as 2 minutes and 30 seconds. The 2 minutes will be added to the *minutes* value. If this exceeds 60 minutes, hours will be modified and so on.

14.1.2 Methods

The `Date` object has two **static** methods. They are called static because they are called without creating an instance of the `Date` object. They are `parse` and `UTC`. The other methods are invoked as methods of a created instance of the `Date` object. In code below *dateObj* is an instance of a `Date` object.

14.1.2.1 Static Methods

14.1.2.1.1 parse

Parses a string containing a date, and returns the number of milliseconds between that date and midnight, January 1, 1970.

```
Date.parse(dateVal)
```

dateVal is a string containing a date.

The `parse` method returns an integer value representing the number of milliseconds between midnight, January 1, 1970 and the date supplied in *dateVal*.

The `parse` method is a static method of the `Date` object. Because it is a static method, it is invoked as shown in the following example, rather than invoked as a method of a created `Date` object.

```
Date.parse("January 1, 2000 00:00 AM")
```

The following rules govern what the `parse` method can successfully parse:

- Short dates can use either a "`/`" or "`-`" date separator, but must follow the month/day/year format, for example "`7/20/96`".
- Long dates of the form "July 10 1995" can be given with the year, month, and day in any order, and the year in 2-digit or 4-digit form. If you use the 2-digit form, the year must be greater than or equal to 70.
- Any text inside parentheses is treated as a comment. These parentheses may be nested.
- Both commas and spaces are treated as delimiters. Multiple delimiters are permitted.
- Month and day names must have two or more characters. Two character names that are not unique are resolved as the last match. For example, "Ju" is resolved as July, not June.
- The stated day of the week is ignored if it is incorrect given the remainder of the supplied date. For example, "Tuesday November 9 1996" is accepted and parsed even though that date actually falls on a Friday. The resulting Date object contains "Friday November 9 1996".
- JScript .NET handles all standard time zones, as well as Universal Coordinated Time (UTC) and Greenwich Mean Time (GMT).
- Hours, minutes, and seconds are separated by colons, although all need not be specified. "10:", "10:11", and "10:11:12" are all valid.

- If the 24-hour clock is used, it is an error to specify "PM" for times later than 12 noon. For example, "23:15 PM" is an error.

- A string containing an invalid date is an error. For example, a string containing two years or two months is an error.

14.1.2.1.2 UTC

`UTC` returns the number of milliseconds between midnight, January 1, 1970 Universal Coordinated Time (UTC) (or GMT) and the supplied date.

```
Date.UTC(year, month, day{, hours{, minutes{, seconds{,ms}}}})
```

The arguments are equal to those of `the Date(year,month,day{,hours{,minutes{,seconds{,ms}}}})` constructor (see Chapter 14.1.1 Constructors).

If the value of an argument is greater than its range, or is a negative number, other stored values are modified accordingly. For example, if you specify 150 seconds, JScript .NET redefines that number as two minutes and 30 seconds.

The difference between the `UTC` method and the corresponding `Date` object constructor is that the `UTC` method assumes UTC, and the `Date` object constructor assumes local time. (Local time when doing server side scripting will be the time zone of the Confirmit servers, not the time zone of the client (the PC of the respondent)).

The `UTC` method is a static method. Therefore, a `Date` object does not have to be created before it can be used.

```
Date.UTC(2000,0,1,0,0,0,0)
```

14.1.2.2 Methods for Setting or Retrieving Values of Parts of Dates

The methods with UTC in the method name use Universal Coordinated Time. The other methods use local time, and since we are dealing with server-side scripts that will be the time of the Confirmit server. The following table show these methods grouped based on the date part they operate on. Curly brackets, { and }, are used to indicate optional arguments.

getFullYear()	getUTCFullYear()	setFullYear(*numYear*{, *numMonth*{, *numDate*}})	setUTCFullYear(*numYear*{, *numMonth*{, *numDate*}})
getMonth()	getUTCMonth()	setMonth(*numMonth*{, *numDate*})	setUTCMonth(*numMonth*{, *numDate*})
getDay()	getUTCDay()		
getDate()	getUTCDate()	setDate(*numDate*)	setUTCDate(*numDate*)
getHours()	getUTCHours()	setHours(*numHours*{, *numMinutes*{, *numSeconds*{, *numMilliseconds*}}})	setUTCHours(*numHours*{, *numMinutes*{, *numSeconds*{, *numMilliseconds*}}})
getMinutes()	getUTCMinutes()	setMinutes(*numMinutes*{, *numSeconds*{, *numMilliseconds*}})	setUTCMinutes(*numMinutes*{, *numSeconds*{, *numMilliseconds*}})
getSeconds()	getUTCSeconds()	setSeconds(*numSeconds*{, *numMilliseconds*}}	setUTCSeconds(*numSeconds*{, *numMilliseconds*})
getMilliseconds()	getUTCMilliseconds()	setMilliseconds(*numMilliseconds*)	setUTCMilliseconds(*numMilliseconds*)
getTime()		setTime(*milliseconds*)	
getTimezoneOffset()			

For the methods that set date parts, the other parts of the date is modified accordingly if the value of an argument is greater than its range or is a negative number. So, if you for example use the `setMinutes` method with 62 as argument, minutes will be set to 2 and hours will be increased with 1. If this makes hours exceed its limit, date is changed to the next day, and so on. This gives a very efficient way of working with dates in scripts.

For all methods with *numMonth* as argument, remember that JScript .NET months use the numbers 0 to 11. 0 is January and 11 is December.

14.1.2.2.1 getFullYear, getUTCFullYear, setFullYear and setUTCFullYear

```
dateObj.getFullYear()
dateObj.getUTCFullYear()
```

`getFullYear` and `getUTCFullYear` return the year value in the `Date` object as an absolute number, thus avoiding the Y2K problem.

```
dateObj.setFullYear(numYear{, numMonth{, numDate}})
dateObj.setUTCFullYear(numYear{, numMonth{, numDate}})
```

setFullYear and setUTCFullYear set the year value in the `Date` object.

numYear is required and is a numeric value equal to the year.

numMonth and *numDate* are both optional. If *numDate* is supplied, *numMonth* must also be supplied. *numMonth* is a numeric value equal to the month (0-11). *numDate* is a numeric value equal to the date (1-31).

If you do not specify the optional arguments, the value will be retrieved from the corresponding get method. For example, if *numMonth* is not specified, JScript .NET uses the value returned from the `getMonth` or `getUTCMonth` method.

If the value of an argument is greater than its range or is a negative number, other stored values are modified accordingly.

There are two methods called `getYear` and `setYear`. They are obsolete and kept for backwards compability only. It is **not** recommended to use them, because of Y2K problems. Use the FullYear methods instead.

14.1.2.2.2 getMonth, getUTCMonth, setMonth and setUTCMonth

```
dateObj.getMonth()
dateObj.getUTCMonth()
```

`getMonth` and `getUTCMonth` return the month value in the `Date` object. The value returned is an integer between 0 and 11 indicating the month value in the `Date` object. Note that this is different from the conventional way of numbering months (1-12). The numeric value for January is 0 and for December the value is 11, one less than the usual way of numbering the months. If "Jan 1, 2000 00:00:00" is stored in a `Date` object, `getMonth` returns 0.

```
dateObj.setMonth(numMonth{, dateVal})
dateObj.setUTCMonth(numMonth{, dateVal})
```

`setMonth` and `setUTCMonth` set the month value in the `Date` object. *numMonth* is required, and is a numeric value equal to the month (0-11).

dateVal is optional. It is a numeric value representing the date. If not supplied, the value from a call to the `getDate` or `getUTCDate` method is used.

If the value of *numMonth* is greater than 11 or is a negative number, the stored year is modified accordingly. For example, if the stored date is "Jan 1, 2000" and setMonth(14) is called, the date is changed to "Mar 1, 2001."

14.1.2.2.3 getDay and getUTCDay

```
dateObj.getDay()
dateObj.getUTCDay()
```

`getDay` and `getUTCDay` return the day of the week of the `Date` object.

The value returned from the `getDay` method is an integer between 0 and 6 representing the day of the week and corresponds to a day of the week as follows:

Value	Day of the Week
`0`	Sunday
`1`	Monday
`2`	Tuesday
`3`	Wednesday
`4`	Thursday
`5`	Friday
`6`	Saturday

14.1.2.2.4 getDate, getUTCDate, setDate and setUTCDate

```
dateObj.getDate()
dateObj.getUTCDate()
```

`getDate` and `getUTCDate` return the day of the month value in a `Date` object. The return value is an integer between 1 and 31 that represents the date value in the `Date` object.

```
dateObj.setDate(numDate)
dateObj.setUTCDate(numDate)
```

`setDate` and `setUTCDate` set the day of month value of the `Date` object.

`numDate` is a numeric value equal to the day of month value. If the value is outside the number of days in the month stored in the `Date` object or a negative number, the other values are modified accordingly, e.g. so that if the date of the object is January 1st 2000 and you use `setUTCDate(32)` on that object, you get February 1st 2000.

14.1.2.2.5 getHours, getUTCHours, setHours and setUTCHours

```
dateObj.getHours()
dateObj.getUTCHours()
```

`getHours` and `getUTCHours` return the hours value in a `Date` object.

The value returned is an integer between 0 and 23, indicating the number of hours since midnight.

```
dateObj.setHours(numHours{, numMin{, numSec{, numMilli}}})
dateObj.setUTCHours(numHours{, numMin{, numSec{, numMilli}}})
```

`setHours` and `setUTCHours` set the hours value in a `Date` object.

The only required argument is *numHours*, which is a numeric value equal to the hours value.

The rest of the arguments are optional. If they are not included, the results from using the corresponding get functions are used. For example, if *numMin* is not provided, the result from `getMinutes` or `getUTCMinutes` is used. All preceding arguments must be included if one of the optional arguments is included. For example, if *numSec* is used, *numMin* must also be included in the method call.

numMin is a numeric value equal to the minutes value, *numSec* is a numeric value equal to the seconds value and *numMilli* is a numeric value equal to the milliseconds value.

If the value of an argument is greater than its range or is a negative number, other stored values are modified accordingly. For example, if the stored date is "Jan 1, 2000 00:00:00", and `setHours(30)` is called, the date is changed to "Jan 2, 2000 06:00:00." Negative numbers have a similar behavior.

14.1.2.2.6 getMinutes, getUTCMinutes, setMinutes and setUTCMinutes

```
dateObj.getMinutes()
dateObj.getUTCMinutes()
```

`getMinutes` and `getUTCMinutes` return the minutes value in a `Date` object.

The value returned is an integer between 0 and 59 equal to the minutes value stored in the `Date` object.

```
dateObj.setMinutes(numMin{, numSec{, numMilli}})
dateObj.setUTCMinutes(numMin{, numSec{, numMilli}})
```

setMinutes and setUTCMinutes set the minutes value in a `Date` object.

The only required argument is *numMinutes*, which is a numeric value equal to the minutes value.

The rest of the arguments are optional. If they are not included, the results from using the corresponding get functions are used. For example, if *numSec* is not provided, the result from `getSeconds` or `getUTCSeconds` is used. All the preceding arguments must be included if one of the optional arguments is included. If *numMilli* is used, *numSec* must also be included in the method call.

numSec is a numeric value equal to the seconds value and *numMilli* is a numeric value equal to the milliseconds value.

If the value of an argument is greater than its range or is a negative number, other stored values are modified accordingly. For example, if the stored date is "Jan 1, 2000 00:00:00", and `setMinutes(62)` is called, the date is changed to "Jan 1, 2000 01:02:00." Negative numbers have a similar behavior.

14.1.2.2.7 getSeconds, getUTCSeconds, setSeconds and setUTCSeconds

```
dateObj.getSeconds()
dateObj.getUTCSeconds()
```

getSeconds and getUTCSeconds return the seconds value in a `Date` object.

The value returned is an integer between 0 and 59 equal to the seconds value stored in the `Date` object.

```
dateObj.setSeconds(numSeconds{, numMilli})
dateObj.setUTCSeconds(numSeconds{, numMilli})
```

`setSeconds` and `setUTCSeconds` set the seconds value in a `Date` object.

The only required argument is *numSeconds*, which is a numeric value equal to the seconds value.

numMilli is optional. If it is not included, the results from using the corresponding get functions are used (`getMillieconds` or `getUTCMilliseconds`).

numMilli is a numeric value equal to the milliseconds value.

If the value of an argument is greater than its range or is a negative number, other stored values are modified accordingly. For example, if the stored date is "Jan 1, 2000 00:00:00", and `setSeconds(124)` is called, the date is changed to "Jan 1, 2000 00:02:04." Negative numbers have a similar behavior.

14.1.2.2.8 getMilliseconds, getUTCMilliseconds, setMilliseconds and setUTCMilliseconds

```
dateObj.getMilliseconds()
dateObj.getUTCMilliseconds()
```

`getMilliseconds` and `getUTCMilliseconds` return the milliseconds value in a `Date` object.

The millisecond value is an integer from 0 to 999.

```
dateObj.setMilliseconds(numMilli)
dateObj.setUTCMilliseconds(numMilli)
```

`setMilliseconds` and `setUTCMilliseconds` set the milliseconds value in the `Date` object.

numMilli is a numeric value equal to the milliseconds value.

If the value of *numMilli* is greater than 999 or is a negative number, the stored number of seconds (and minutes, hours, and so forth if necessary) is modified accordingly.

14.1.2.2.9 getTime and setTime

```
dateObj.getTime()
```

`getTime` returns an integer value representing the number of milliseconds between midnight, January 1, 1970 UTC and the time value in the `Date` object. Negative numbers indicate dates prior to 1970.

```
dateObj.setTime(milliseconds)
```

`setTime` sets the date and time of the instance of the `Date` object. *milliseconds* is an integer value representing the number of elapsed seconds since midnight, January 1, 1970 UTC. If *milliseconds* is negative, it indicates a date before 1970.

14.1.2.2.10 getTimezoneOffset

```
dateObj.getTimezoneOffset()
```

Returns the difference in minutes between the time on the host computer (the Confirmit server when doing server-side scripting) and Universal Coordinated Time (UTC). This number will be positive if you are behind UTC (e.g., Pacific Daylight Time), and negative if you are ahead of UTC (e.g., Japan).

14.1.2.3 Conversion Methods

14.1.2.3.1 valueOf

```
dateObj.valueOf()
```

`valueOf` returns the stored time value in milliseconds since midnight, January 1, 1970 UTC.

14.1.2.3.2 toLocaleString, toString and toUTCString

```
dateObj.toLocaleString()
```

toLocaleString returns a date converted to a string using the current locale. This mean that it will give the Confirmit server time, and the date is formatted according to the Regional Settings on the server.

```
dateObj.toUTCString()
```

toUTCString returns a string that contains the date formatted using Universal Coordinated Time (UTC) convention.

```
dateObj.toString()
```

toString returns the textual representation of the date, in the time of the Confirmit server, but not reflecting the locale settings.

Here is an example of the output from the three different methods from a server with English (United States) as locale and where the time zone is EST (like FIRM's Confirmit ASP servers):

Method	Output
toUTCString	Sat, 1 Jan 2000 00:00:00 UTC
toLocaleString	Friday, December 31, 1999 7:00:00 PM
toString	Fri Dec 31 19:00:00 EST 1999

There also exists a method called toGMTString, which returns a date converted to a string using Greenwich Mean Time(GMT). This method is obsolete, and it is recommended that you use the toUTCString method instead.

14.1.3 Confirmit Date Functions

Confirmit provides six functions for working with dates. InterviewStart and InterviewEnd return the interview_start and interview_end time stamps automatically recorded when the respondent starts the interview and reach the end of the interview or a stop node (the end page). SetInterviewStart and SetInterviewEnd set these timestamps. IsDate and IsDateFmt are used for validation of dates.

14.1.3.1 InterviewStart and InterviewEnd

Both of these functions return dates, however they will return the dates as strings. This is the syntax you can use to get instances of the JScript .NET Date object from InterviewStart and InterviewEnd:

```
dateObj = new Date(InterviewStart());
dateObj = new Date(InterviewEnd());
```

Example 55: Calculating Time Spent

You can calculate the response time of the respondents on specific pages or sections of the survey by using the Date object. Let us say you want to calculate the number of seconds the respondent has spent from the beginning to a specific point in the interview and store that in an open text question with hidden and numeric property (*time_part1*).

```
if(!f("time_part1").toBoolean())
 {
  var d1 = new Date(InterviewStart())
  var d2 = new Date()
```

```
 var diff = (d2.getTime() - d1.getTime())/1000

 f("time_part1").set(diff)
}
```

The if-condition is there to make sure that the time is just calculated the first time the respondent goes through the questionnaire (if he or she is allowed to modify previous responses), when no value has been set in the hidden *time_part1* question.

You have to make sure you use the correct settings on the hidden numeric question. Scale should be set to 3 (3 digits after the decimal point) since `getTime` returns milliseconds. We divide the number of milliseconds with 1000 to get the answer stored as seconds. Precision has to be set high enough to be able to store both the 3 digits after the decimal point and the highest number of seconds a respondent will use.

If you want to calculate the response time for the next part of the questionnaire as well, you can for example use the following script to set that in another hidden open text question with numeric property (*time_part2*):

```
if(!f("time_part2").toBoolean())
{
 var d1 = new Date(InterviewStart());
 var d2 = new Date();

 var diff = ((d2.getTime() - d1.getTime())/1000)-f("time_part1").toNumber();

 f("time_part2").set(diff);
}
```

14.1.3.2 IsDateFmt and IsDate

These functions are also described in Chapter 10.1.7.2 IsDateFmt and IsDate. Here we will look closer at the properties of the objects returned from them and provide some examples on how to use the functions together with the JScript `Date` object.

If you have a string with day, month and year in some format (for example from an open text question) and you want to check if it is a valid date, you can use the `IsDateFmt` function.

```
dObj = IsDateFmt( argument,format );
```

If you have three values, one for day, one for month and one for year, (for example from three questions), you can use `IsDate` to check if it is a valid date.

```
dObj = IsDate( day,month,year );
```

Both `IsDateFmt` and `IsDate` return an object with the properties `day`, `month` and `year` if the date is valid, `null` otherwise. The object returned is not a `Date` object – it has none of the methods of the `Date` object, just these three properties.

```
dObj.day
```

returns the day part of the date as an integer in the range 1-31.

```
dObj.month
```

returns the month part of the date as an integer in the range 1-12. (**Important**: Note that this is different from the `Date` object, where `getMonth` and `getUTCMonth` return an integer in the range 0-11).

```
dObj.year
```

returns the year part of the date as an integer (4 digits).

Example 56: *Validating Date Format and that it is a Valid Date after Current Date*

In this example we have an open text question and want the respondent to enter a date in the format YYYY-MM-DD. We want to validate that the format is correct and that the date is a valid date, and also that the date is not after the current date.

This can be done with the following script in the validation code field of the open text question (*date1*).

```
var d = IsDateFmt(f("date1").get(),"YYYY-MM-DD");

if(!d)
 {
  RaiseError();
  SetQuestionErrorMessage(LangIDs.en,"Invalid date. Please correct using the format
YYYY-MM-DD");
 }
else
 {
  var dt = new Date();
  dt.setFullYear(d.year,d.month-1,d.day);
  var current = new Date();
  if(dt.valueOf() > current.valueOf())
   {
    RaiseError();
    SetQuestionErrorMessage(LangIDs.en,"Please do not enter a date after the current
date.");
   }
 }
```

If the date provided is not valid, `IsDateFmt(f("date1").get(),"YYYY-MM-DD")` will return `null`. Used in a condition `null` will yield the Boolean value `false`. If the date is valid, `IsDateFmt` will return an object. Used in a condition this will yield `true`. So the first `if` condition, with `!d` , makes sure there will come an error message only when the date is invalid or wrong format is used.

It is important that we include the rest of the script in an else branch because the properties `year`, `month` and `day` are not defined if `null` is returned. So we need to make sure that that part of the script only is run when we have an object in the variable `d`, to prevent script errors.

Then we introduce two instances of the `Date` object, and set one of them (`dt`) to the date answered, and the other (`current`) to the current date. We use the `setFullYear` method to set the date. Observe that we have to subtract 1 from the month, since the `Date` object uses the integers 0-11 for month, whereas the `day` property in the object returned from `IsDateFmt` uses the more conventional integers 1-12.

Example 57: *Validating that a Date with Dropdowns is within the next two Weeks*

You can also set up the date with three questions, one for *year*, one for *month* and one for *day*. This can e.g. be set up in a 3D grid with 3 grid questions as dropdowns like this:

date2 - Course date (3DGrid)
year (Grid)
month (Grid)
day (Grid)

Course date

When do you wish to participate?

	Year	Month	Day
Date:	2002	April	12

12
13
14
15
16
17
18
19
20
21
22

Powered by confirmit

>>

NB: The precodes of the questions must be numeric and the same as the legal `year/month/day` values. For *year* the precodes must be the full value (e.g. 2002), for *month* they have to be numbers between 1 and 12 and for *day* they have to be numbers between 1 and 31. See also Example 34:.

Let us say that we want to check that the answer is a valid date (so that e.g. February 30th is not allowed), and that it is a date in the next two weeks from the current date.

```
var d = IsDate(f("day")["1"].get(),f("month")["1"].get(),f("year")["1"].get());

if(!d)
 {
  RaiseError();
  SetQuestionErrorMessage(LangIDs.en,"Invalid date. Please correct.");
 }
else
 {
  var dt = new Date();
  dt.setFullYear
(d.year,d.month-1,d.day);
  var current = new Date();
  var limit = new Date();
  limit.setDate(current.getDate()+14)
  if(dt.valueOf() < current.valueOf() || dt.valueOf() > limit.valueOf())
   {
    RaiseError();
    SetQuestionErrorMessage(LangIDs.en,"Please select a date within the next two
weeks.");
   }
 }
```

This script takes advantage of the fact that when using `setDate`, the month and possibly also year value will automatically be updated if the value for date is outside the limit for the month. So if the result of `current.getDate()+14` is a number larger than the number of days in a month, `limit` will be set to a date in the next month. For example, if current date is April 20th 2002, the limit will be 4th of May 2002 (April 20th + 14 days).

Example 58: *Finding the Weekday*

You can use the `Date` object to make a small application in Confirmit to find the weekday of a specific date. Start with an open text question (*date2*) asking for a date. The validation can be done like this:

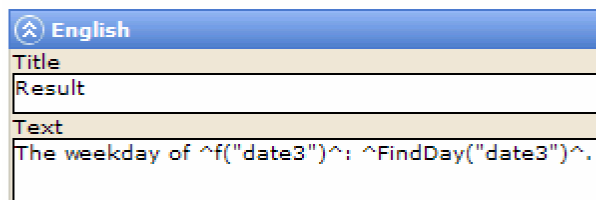
```
var d = IsDateFmt(f("date3").get(),"YYYY-MM-DD");

if(!d)
 {
  RaiseError();
  SetQuestionErrorMessage(LangIDs.en,"Invalid date. Please correct. Use the format
YYYY-MM-DD");
 }
```

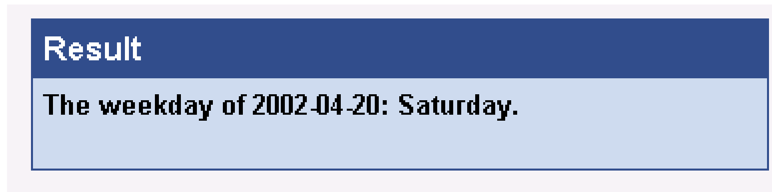
Then you can have a script node with the following function:

```
function FindDay(qID) : String
 {
  var d = IsDateFmt(f(qID).get(),"YYYY-MM-DD");
  var dt = new Date();
  dt.setFullYear(d.year,d.month-1,d.day);
  var days = new Array
("Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday")
  return days[dt.getDay()]
 }
```

After the open text question you can have an info node like this:



This will for example give this output if the answer on the date question was 2002-04-20:



Exercise 7: *Course Registration*

Let us say you want to build a questionnaire where people can register for weekly courses that are held on Mondays. You want the respondents to provide the date they want to register for in an Open Text Question, but have to validate the date format (which should be "MM/DD YYYY") and check that the date provided is a Monday and it is after the current date.

Write a validation script for this question. Question ID is *date4*.

See answer in Appendix A: Answers to Exercises.

Example 59: *Converting Data of Birth into Age (in number of years)*

If you have an open text question *dob* where respondents insert their date of birth, you can calculate the respondent's age and set it in a hidden question *age* with the following script, which is placed in the validation code of the question:

```
var d = IsDateFmt(f("dob").get(),"YYYY-MM-DD");
if(!d)
 {
  RaiseError();
  SetQuestionErrorMessage(LangIDs.en,"Invalid date. Please use the format YYYY-MM-
DD");
 }
else
 {
  var birthday = new Date();
  birthday.setFullYear(d.year,d.month-1,d.day);
  var today = new Date();
  var years = today.getFullYear() - birthday.getFullYear();
  birthday.setYear(today.getFullYear());
  // If your birthday hasn't occurred yet this year, subtract 1.
  if(today < birthday)
  {
   years-- ;
  }
 f("age").set(years);
}
```

14.2 The Math Object

The `Math` object provides a standard library of mathematical constants and functions. The `Math` object cannot be created using the `new` operator, and gives an error if you attempt to do so. This is because it is a built-in object and not an object type. Most of the properties and methods are different mathematical constants and functions that you will seldom need to use in your questionnaires. The most important methods are the rounding methods in Chapter 14.2.2.2 Rounding and the random method in Chapter 14.2.2.3 Random .

14.2.1 Properties

The properties of the `Math` object are different mathematical constants.

```
Math.E
```

Euler's constant, the base of natural logarithms. The `E` property is approximately equal to 2.718.

```
Math.LN2
```

The natural logarithm of 2. The `LN2` property is approximately equal to 0.693.

```
Math.LN10
```

The natural logarithm of 10. The `LN10` property is approximately equal to 2.302.

```
Math.LOG2E
```

The base-2 logarithm of e, Euler's constant. The `LOG2E` property is approximately equal to 1.442.

```
Math.LOG10E
```

The base-10 logarithm of *e*, Euler's constant. The `LOG10E` property is approximately equal to 0.434.

```
Math.PI
```

π, the ratio of the circumference of a circle to its diameter, approximately 3.141592653589793.

```
Math.SQRT1_2
```

The square root of ½, or one divided by the square root of 2. The `SQRT1_2` property is approximately equal to 0.707.

```
Math.SQRT2
```

The square root of 2. The `SQRT2` property is approximately equal to 1.414.

14.2.2 Methods

14.2.2.1 Trigonometric Functions

```
Math.cos(x)
```

returns the cosine of a numeric expression x.

```
Math.sin(x)
```

returns the sine of a numeric expression x.

```
Math.tan(x)
```

returns the tangent of a numeric expression x.

```
Math.acos(x)
```

returns the arc cosine of a numeric expression x.

```
Math.asin(x)
```

returns the arc sine of a numeric expression x.

```
Math.atan(x)
```

returns the arctangent of a numeric expression x.

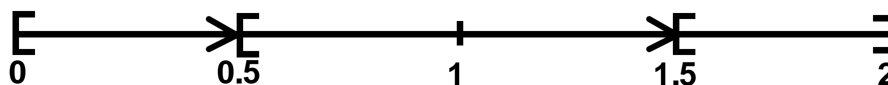
```
Math.atan2(x,y)
```

Returns the angle of the polar coordinate corresponding to (x,y)

14.2.2.2 Rounding

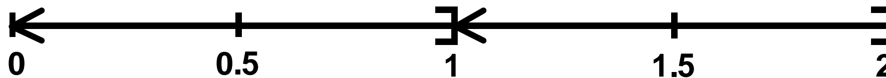
```
Math.round(x)
```

returns a supplied numeric expression x rounded to the nearest integer. If the decimal portion of number is 0.5 or greater, the return value is equal to the smallest integer greater than number. Otherwise, round returns the largest integer less than or equal to number. This is exactly as conventional rounding. If we have a floating point number between 0 and 2, values from 0 and up to, but not including 0.5 will be rounded to 0, values from and including 0.5 and up to, but not including 1.5 will be rounded to 1 and values from and including 1.5 to and including 2 will be rounded to 2.



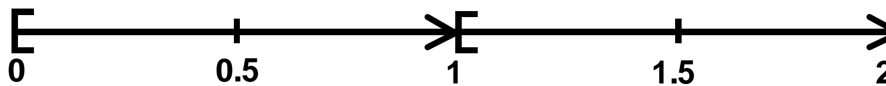

```
Math.ceil(x)
```

returns the smallest integer greater than or equal to its numeric argument x, i.e. rounding **up** to the nearest integer. If we have a floating-point number between 0 and 2, the value 0 will be rounded to 0, and all values greater than 0 and up to and including 1 will be rounded to 1. All values greater than 1 and including 2 will be rounded to 2.




```
Math.floor(x)
```

returns the greatest integer less than or equal to its numeric argument x, i.e. rounding down to the nearest integer. If we have a floating point number between 0 and 2, all values from 0 and up to, but not including 1, will be rounded to 0, and the value 1 and all values from 1 and up to, but not including 2 will be rounded to 2, and the value 2 will be rounded to 2.



Example 60: *Rounding to a Number with Two Digits*

If we ask people to provide their yearly salary, and we want to compute the monthly salary it can be done by dividing the yearly salary with 12. However, this might be a number with many decimals. If we want the answer with an accuracy of two digits and need to round the result, we can do that by multiplying the result with 100, then do the rounding, and then divide the result with 100. Let us say the yearly salary is in the question *yearly* and the monthly salary should be stored in the question *monthly*. Then we can use a script like this to set monthly:

```
var yearly : int = f("yearly").toNumber();
var monthly : float = yearly/12;
f("monthly").set(Math.round(monthly*100)/100);
```

14.2.2.3 Random

```
Math.random()
```

returns a random number between 0 and 1 (float), e.g. 0.523. The number generated is from 0 (inclusive) to 1 (exclusive), that is, the returned number can be zero, but it will always be less than one.

This is an extremely powerful feature to use in your surveys when you want to pick responses randomly, or send random respondents to different parts of the questionnaire. Combined with conditions and arithmetic operations you can program extremely powerful solutions for the selections.

Example 61: *Picking n Random Items from the Answers to a Multi Question*

If you have a multi question where the respondent picks some brands and want to proceed with more detailed questions on some of the brands the respondent chooses, but not all, you can use a script to randomly pick some of the brands the respondent selected. Let's say the multi question has question ID brands, and we have a hidden multi question with the same answer list that has question ID *random_brands*.

```
var fromForm = f("brands");
var toForm = f("random_brands");
```

```
const numberOfItems : int = 3;

var available = new Set();
available = available.union(fromForm);
var selected = new Set();
if(available.size() <= numberOfItems)
 {
  selected = available;
 }
else
 {
  while(selected.size() < numberOfItems)
   {
    var precodes = available.members();
    var randomNumber : float = Math.random()*precodes.length;
    var randomIndex : int = Math.floor(randomNumber);
    var selectedCode = precodes[randomIndex];
    available.remove(selectedCode);
    selected.add(selectedCode);
   }
 }

var precodes = fromForm.domainValues();
for(var i : int = 0;i<precodes.length;i++)
 {
  var code = precodes[i];
  if(selected.inc(code))
   {
    toForm[code].set("1");
   }
  else
   {
    toForm[code].set("0");
   }
 }
```

In this script we pick responses from a set of available responses (the answers to the multi question *brands*, stored in the variable `available`) and add them to the set stored in the variable `selected`. At the end we go through all answer alternatives, and if an answer is contained in the set in the selected variable, we set that answer in the *random_brands* multi.

If there number of responses is less than or equal to `numberOfItems` (here: 3), we obviously include all those responses. But if there are more than 3 responses, we have to pick random responses from the answers given.

The random picking is done in this `while` loop:

```
  while(selected.size() < numberOfItems)
   {
    var precodes = available.members();
    var randomNumber : float = Math.random()*precodes.length;
    var randomIndex : int = Math.floor(randomNumber);
    var selectedCode = precodes[randomIndex];
    available.remove(selectedCode);
    selected.add(selectedCode);
   }
```

We run through the loop until we have `numberofItems` responses (here 3) in the `selected` set. When a random response is selected, we remove its precode from the `available` set and insert it into the `selected` set. So `available` will at any time hold the responses that have not been picked yet. At the start of each iteration inside the loop we build an array `precodes` with the available precodes. The `members` method converts a set to an array.

Let us say we start with the precodes "1","3","4" and "8". The array `precodes` will consist of the following items in the first iteration:

```
precodes[0] = "1"
precodes[1] = "3"
precodes[2] = "4"
precodes[3] = "8"
```

Math.random will give a number between 0 (inclusive) and 1 (exclusive). When this is multiplied with precodes.length (4 in our example), randomNumber will be set to a number between 0 (inclusive) and precodes.length (4) (exclusive):

```
0<=randomNumber<precodes.length
i.e.
0<=randomNumber<4
```

in our example. Let us say that the number returned from Math.random is 0.6283. Then randomNumber will be 4*0.6283 = 2.5132.

This number is rounded down to the nearest integer by using the Math.floor method in the next step:

```
var randomIndex : int = Math.floor(randomNumber);
```

This means that randomIndex will be one of 0,1,2,...,precodes.length-1, i.e. 0,1,2,3.

It is really important that Math.floor is used instead of Math.round or Math.ceil. This is the only way to make sure that the probability of selecting the indexes is the same for all of them and you get no errors. Math.round would extend the set of possible picks with the index precodes.length (i.e. the number 4 in our example). This would cause problems because 4 is not an index in the array. If we used Math.ceil, we would round up to the numbers 1,2,3 and 4, and could subtract 1 from these numbers to get the index. However, even though the probability of this happening is extremely small, there is a small chance the number 0 would be returned from Math.random(). And using Math.ceil on 0 would yield 0, and this would again cause problems.

In our example, where randomNumber was calculated to 2.5132, randomIndex will get the value 2.

precodes[2] is "4", so the precode 4 will be removed from the available set and added to the selected set.

The while loop will now continue to the next iteration with the remaining three precodes, so in our example precodes will be have the following items in the next iteration:

```
precodes[0] = "1"
precodes[1] = "3"
precodes[2] = "8"
```

Then the script will randomly pick one of these, and continue with this process until 3 items are selected.

Picking random items like this is best suited for surveys where the respondent is not allowed to modify previous answers. If the respondent goes back to the *brands* question and then forwards again, the script is run again, possibly resulting in a different set. So then the respondent will get questions for other brands. This may be confusing and cause irritation for the respondent.

Example 62: Randomly Assigning which Part of a Survey the Respondents Should Answer

To limit the number of questions each respondent has to answer in a long survey, you may want to split the survey into different parts, and randomly pick which part a particular respondent should answer.

This can be done by randomly setting the response to a hidden single question, and then route the respondents to their questions with conditions on this hidden question.

Let us say the hidden single question has question ID *part*. *part* can be set at the beginning of the survey with a script like this:

```
var form = f("part");
if(!form.toBoolean())
 {
  var precodes = form.domainValues();
  var randomNumber : float = Math.random()*precodes.length;
  var randomIndex : int = Math.floor(randomNumber);
```

```
  var code = precodes[randomIndex];
  form.set(code);
 }
```

This is very similar to the previous example. Here we pick the precode from an array of all precodes from the answer list of the hidden single question part (using `domainValues`).

The condition with `toBoolean` is used so that the single question is set only the first time the script is run. So this solution will work even when the respondent is allowed to modify previous answers.

14.2.2.4 Maximum and Minimum

```
Math.max({number1{, number2{...{, numberN}}}})
Math.min({number1{, number2{...{, numberN}}}})
```

`max` returns the greater of zero or more supplied numeric expressions. `min` returns the lesser of zero or more supplied numeric expressions. The curly brackets are used to indicate that the numerical expressions $number_1,...,number_N$ are optional.

If no arguments are provided, the return value is equal to negative infinity for `max` and positive infinity for `min`. If any argument is `NaN`, the return value is also `NaN` (Not a Number).

However, we would recommend using the Confirmit functions `Max` and `Min` (see Chapter 10.1.1.4 Max and Min) when working on questions, since these functions automatically converts the answers to numbers.

14.2.2.5 Absolute value

```
Math.abs(number)
```

`abs` returns the absolute value of a numeric expression number.

`abs` can for example be used when you want the difference between two numbers as a positive number, no matter which of them is the highest number.

```
Math.abs(x-y)
```

If `x` is 10 and `y` is 4, `x-y` will return 6. If `x` is 4 and `y` is 10, `x-y` will return –6. The absolute value will be 6 for both.

14.2.2.6 Exponents, Logarithms and Square Root

```
Math.exp(number)
```

`exp` returns e (the base of natural logarithms) raised to a power, e^{number}.

e is Euler's constant, approximately equal to 2.178.

```
Math.log(number)
```

`log` returns the natural logarithm of a *number*. The base is e, Euler's constant, approximately equal to 2.178.

```
Math.pow(base, exponent)
```

`pow` returns the value of a base expression taken to a specified power, $base^{exponent.}$

```
Math.sqrt(number)
```

Returns the square root of a numeric expression number. If *number* is negative, the return value is zero.

14.3 The String Object

The `String` object type allows strings to be accessed as objects. It allows manipulation and formatting of text strings and determination and location of substrings within strings.

14.3.1 Constructors

An instance of the `String` object can be created like this:

```
newString = new String({"stringLiteral"});
```

The curly brackets are used to indicate that the string literal is optional.

`String` objects can also be created implicitly using string literals.

```
newString = "{stringLiteral}";
```

or

```
newString : String = "{stringLiteral}";
```

14.3.2 Properties

```
strVariable.length
"String Literal".length
```

`length` returns the length of a `String` object, an integer with the number of characters in the `String` object.

14.3.3 Index

Many of the methods of the `String` object refer to **index** on a string. The index is used to refer to a characters position within a string. If you have the string

```
 0 1 2 3 4 5 6 7 8 9101112131415
"This is a string"
```

the character a has index 8, because the index starts at 0 for first character. The index of the last character will always be 1 less than the string's length. This is similar to indexing in arrays.

14.3.4 Converting to String from Other Types

Often you want to convert variables of different types to string to use some of the string methods to manipulate on the content. The easiest way of converting a variable to a string, is to concatenate it with an empty string using the + operator:

```
variable+""
```

Since string has higher order of precedence than other types, this will convert the variable to type string. You can also use the `.toString()` method.

14.3.5 Methods

A few methods of the string objects that use Regular Expressions are described in Chapter 14.4.4 String Object Methods that Uses Regular Expression Objects. The rest is described below.

14.3.5.1 Methods Returning a Character or a Character Code at a Specific Index

```
strObj.charAt(index)
```

`charAt` returns the character at the specified *`index`*. Valid values for *`index`* are between 0 and the length of the string minus 1. `charAt` with an index out of valid range returns an empty string.

```
strObj.charCodeAt(index)
```

`charCodeAt` returns an integer representing the Unicode encoding of the character at the specified *`index`*. *`index`* is a number between 0 and the length of the string minus 1. If there is no character at the specified index, `NaN` is returned.

14.3.5.2 Building a String from a Number of Unicode Characters

```
String.fromCharCode({code1{, code2{, ...{, codeN}}}})
```

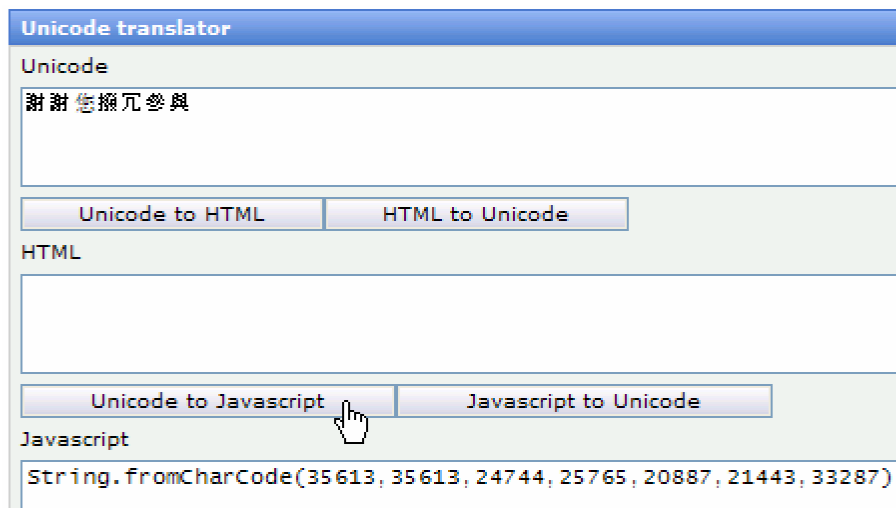
`fromCharCode` returns a string from a number of Unicode character values. If no arguments are supplied, the result is the empty string.

A String object need not be created before calling `fromCharCode`. The method can be applied directly on the object type.

In the following example, `txt` will be set to the string "Confirmit":

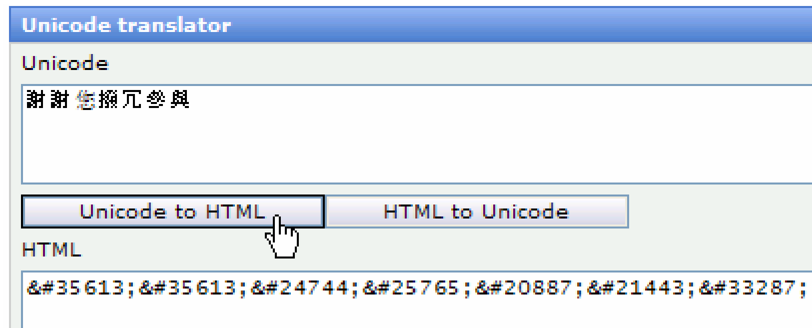
```
var txt = String.fromCharCode(67,111,110,102,105,114,109,105,116);
```

This is the way to be able to set Unicode text in scripts. Confirmit has a "Unicode to JavaScript" converter available when you edit a script node. You may use that to get a text converted into character codes like this.




```
var txt = String.fromCharCode(35613,35613,24744,25765,20887,21443,33287); //plain text
```

This can for example be used to build the body of the mail when sending MIME text emails with the `SendMail` function (see Chapter 10.1.8.1 SendMail). If you send HTML emails, you should convert the Unicode to HTML instead.




```
var txt = "&#35613;&#35613;&#24744;&#25765;&#20887;&#21443;&#33287;"; //HTML
```

14.3.5.3 Changing Case

```
strObj.toLowerCase()
```

`toLowerCase` returns a string where all alphabetic characters have been converted to lowercase.

```
strObj.toUpperCase()
```

`toUpperCase` returns a string where all alphabetic characters have been converted to uppercase.

Both of these methods have no effect on non-alphabetic characters.

Example 63: Checking a User Name and Password where Username is Case Insensitive

You can password-protect an open survey with a password and user name combination that is the same for all respondents. This can be used when you do not upload any respondent list before starting the survey, but still want to limit the access to the survey. You should be aware that this would not stop the respondents from accessing the survey more than once. See also Example 5:.

The user name and password can be given in two open text questions, *username* and *password*, the latter with the password property. If you want the password to be case sensitive, but not the username you can use a validation code like this:

```
if(f("username").get().toUpperCase() != "USERNAME" || f("password").get() !=
"Password")
 {
  RaiseError();
  SetQuestionErrorMessage(LangIDs.en,"Wrong username or password. Please try again.");
 }
```

14.3.5.4 Searching for a Substring within a String

```
strObj.indexOf(subString{, startIndex})
```

`indexOf` returns the character position of the first occurrence of a string *`subString`* within a String object. *`startIndex`* is optional, and is an integer value specifying the index to begin the search. If omitted, searching starts at the beginning of the string. If the *`subString`* is not found, -1 is returned.

If *`startIndex`* is negative, *`startIndex`* is treated as zero. If it is larger than the greatest character position index, it is treated as the largest possible index.

Searching is performed from left to right.

```
strObj.lastIndexOf(substring{, startindex})
```

returns the character position of the last occurrence of a *subString* within a String object. *startIndex* is optional, and is an integer value specifying the index to begin searching within the String object. If omitted, searching begins at the end of the string. If the *subString* is not found, a-1 is returned.

If *startIndex* is negative, *startIndex* is treated as zero. If it is larger than the greatest character position index, it is treated as the largest possible index.

Searching is performed right to left.

Example 64: On the Fly Recoding

You may use the indexOf method to search for strings within a text. This can be done to do easy recoding, e.g. if you have an open text question about car brands you can search for strings like VOLVO or FORD like this:

```
if(f("openbrands").get().toUpperCase().indexOf("FORD") != -1)
 {
  f("brands")["1"].set("1");
 }
if(f("openbrands").get().toUpperCase().indexOf("MERCEDES") != -1)
 {
  f("brands")["2"].set("1");
 }
if(f("openbrands").get().toUpperCase().indexOf("VOLVO") != -1)
 {
  f("brands")["3"].set("1");
 }
```

and so on. (*openbrands* is here the open text question, and *brands* is a hidden multi question used e.g. for reporting).

Here we use toUpperCase to convert case to make the search case insensitive, so that the strings "Volvo", "volvo" and "VOLVO" all will be recognized.

If indexOf returns anything different from –1, it means that the string has been found in the *openbrands* question.

The problem with this solution is respondents that spell brand names wrong. You can of course check for different common misspellings ("VOVLO" etc.) but usually you will need some sort of manual recoding as well.

14.3.5.5 Retrieving a Section of a String (Substring)

```
stringObj.slice(start, {end})
```

returns a section of a string. *start* is required and is the index of the first character in the section of *stringObj*. *end* is optional and is the index after the last character in the section of *stringObj*. The slice method copies up to, but not including, the element indicated by end.

In this example:

```
var txt = "The methods of the String Object can be used for text manipulation.";
var section = txt.slice(19,32);
```

section will be set to the substring

```
"String Object"
```

If *start* is negative, it is treated as *length+start* where *length* is the length of the string. If *end* is negative, it is treated as *length+end* where *length* is the length of the string. If *end* is omitted, extraction continues to the end of *stringObj*. If *end* occurs before *start*, no characters are copied to the new string.

```
stringObj.substr(start {, length })
```

returns a substring beginning at a specified location *start* and having a specified *length*.

start is required, and is the starting position (index) of the desired substring. *length* is optional and is the number of characters to include in the returned substring.

If *length* is zero or negative, an empty string is returned. If not specified, the substring continues to the end of the string.

```
stingObj.substring(start, end)
```

returns the substring at the specified location within a `String` object.

start is the index indicating the beginning of the substring and *end* is the index indicating the end of the substring. The `substring` method returns a string containing the substring from *start* up to, but not including, *end*.

The `substring` method uses the lower value of *start* and *end* as the beginning point of the substring. For example, *stringObj*`.substring(0,3)` and *stringObj*`.substring(3,0)` return the same substring.

If either *start* or *end* is `NaN` or negative, it is replaced with zero.

The length of the substring is equal to the absolute value of the difference between *start* and *end*. For example, the length of the substring returned in *stringObj*`.substring(0,3)` and *strObj*`.substring(3,0)` is three.

Example 65: Replacing Last Comma with "and" in a Listing of Answers

If you refer to a multi question, e.g. *brands*, in response piping with ^'s in a question text, then the last two items in the listing are separated with "and" in English.

```
^f("brands")^
```

If the brands "Ford","Mercedes","Volvo" are answers to the brands question, the string returned will be "Ford, Mercedes and Volvo".

However, when you refer to

```
f("brands").categoryLabels()
```

in a script node, e.g. to include the answers in an email text, the result will be an array with the elements. When this is converted to a string (for example by adding it to a string expression), you get a string that lists the elements separated with commas, but not with "and" between the last two elements: "Ford,Mercedes,Volvo".

The following script will replace the last comma with " and ". (Observe the spaces in front of and after and).

```
var body : String = "";
body += "Here are the answers on the brands question:\n\n"

var form = f("brands");
body += form.categoryLabels();

if(form.size() > 1)
 {
  body = body.substring(0,body.lastIndexOf(",")) + " and " +
body.substring(body.lastIndexOf(",")+1,body.length);
 }

SendMail("interviewer@confirmit.com",f("email"),"Answers",body);
```

If there are two answers or more, the answers string will be set to the substring from the beginning of the `answers` string to (but excluding) the last comma, the string `" and "` and the substring from the character after the last comma to the end of the `answers` string.

14.3.5.6 Splitting and Joining Strings

```
stringObj.split({separator{, limit})
```

`split` returns the array of strings that results when a string is separated into substrings. *`separator`* is a string or an instance of a Regular Expression object (see Chapter 14.4.3 The Regular Expression Object) identifying one or more characters to use in separating the string. If omitted, a single-element array containing the entire string is returned. *`limit`* is a value used to limit the number of elements returned in the array. The result of the `split` method is an array of strings split at each point where separator occurs in *`stringObj`*. *`stringObj`* itself is not modified. The *`separator`* is not returned as part of any array element.

```
string1.concat({string2{, string3{, . . . {, stringN}}}})
```

`concat` returns a string value containing the concatenation of two or more supplied strings. The result of the `concat` method is equivalent to:

```
string1 + string2 + string3 + ... + stringN.
```

Example 66: *Generating an Array from a String with Values*

Let us say you send in information of what products the respondent uses with the url to an open survey, for example like this:

http://survey.confirmit.com/wix/pXXXXXXXX.aspx?products=1x15x17x19
http://survey.confirmit.com/wix/pXXXXXXXX.aspx?products=6x11x20

1, 6, 11, 15, 17, 19 and 20 are different product codes, and the respondent can have any number of these. Sending them in like this gives more condensed urls than sending in one value (yes/no) for each product. It could possibly be a very long list of products.

In the Confirmit survey we capture the products list with `Request` (will be described in Chapter 16.1 Request), and want to set a hidden multi question products with the values sent in with the url. The *products* multi question uses precodes that are equal to the product codes that are sent in with the url.

To be able to set the *products* question, we have to split the string returned from `Request("products")` with "x" as delimiter to get an array with the product codes sent in:

```
var txt : String = Request("products")+"";
var form = f("products");

var productArray = txt.split("x");

for(var i : int = 0;i<productArray.length;i++)
 {
  form[productArray[i]].set("1");
 }
```

14.3.5.7 Retrieving the String Value

```
strObj.toString()
strObj.valueOf()
```

Both `toString` and `valueOf` returns the string value of the `String` object.

14.3.5.8 Methods that Add HTML Tags to a String

There are a number of methods available that adds HTML code to your strings. They are listed in the table below. They may for example be used in your error messages, or in expressions that will be displayed in info or question titles, texts or answers (with response piping).

Method	Equivalent to
strVariable.anchor(*anchorString*)	*strVariable* = '<A NAME="'+ *anchorString* + '">' + *strVariable* + '</A>'
strVariable.big()	*strVariable* = '<BIG>' + *strVariable* + '</BIG>'
strVariable.blink()	*strVariable* = '<BLINK>' + *strVariable* + ' </BLINK>'
strVariable.bold()	*strVariable* = '<B>' + *strVariable* + '</B>'
strVariable.fixed()	*strVariable* = '<TT>' + *strVariable* + '</TT>'
strVariable.fontcolor(*colorVal*)	*strVariable* = '<FONT COLOR="' + *colorVal* + '">' + *strVariable* + '</FONT>'
strVariable.fontsize(*intSize*)	*strVariable* = '<FONT SIZE="' + *intSize* + '">' + *strVariable* + '</FONT>'
strVariable.italics()	*strVariable* = '<I>' + *strVariable* + '</I>'
strVariable.link(*linkstring*)	*strVariable* = '<A HREF="' + *linkstring* + '">' + *strVariable* + '</A>'
strVariable.small()	*strVariable* = '<SMALL>' + *strVariable* + '</SMALL>'
strVariable.strike()	*strVariable* = '<STRIKE>' + *strVariable* + '</STRIKE>'
strVariable.sub()	*strVariable* = '_{' + *strVariable* + '}'
strVariable.sup()	*strVariable* = '^{' + *strVariable* + '}'

14.4 Regular Expressions

Regular Expressions are character-matching patterns that are used to find and/or replace character patterns in strings.

With Regular Expressions, you can:

- Test for a pattern within a string. For example, you can test an input string to see if a telephone number pattern or a credit card number pattern occurs within the string.

- Replace text. You can use a Regular Expression to identify specific text in a string and either remove it completely or replace it with other text.

- Extract a substring from a string based upon a pattern match.

The syntax of Regular Expressions is a bit hard to understand. If you have problems understanding it, you are advised to check the examples, try to modify them and test what the differences are. The Perl scripting language popularized Regular Expressions, and JScript 's support of Regular Expressions is based on that of Perl. So for further reading about Regular Expressions, you may search for Perl documentation in addition to JScript . NET documentation.

Regular Expressions are implemented as Regular Expression objects and are created as follows:

```
var re = /pattern/{flags};
```

(like Regular Expressions in Perl) or

```
var re = new RegExp("pattern",{"flags"});
```

(like normal syntax for instantiating an object in JScript .NET)

`pattern` is the pattern to be matched and the optional `flags` is a string containing `g`, `i`, and/or `m`. The `g` stands for **global**, the `i` stands for **ignore case** and the `m` for **multi-line search**. When defining a Regular Expression with the syntax `/pattern/flags`, do not use quotation marks around the strings, but when using the other notation you have to use them:

```
var re = new RegExp("confirmit","g");
```

is the same as

```
var re = /confirmit/g;
```

This matches all occurrences of "confirmit".

14.4.1 Regular Expression Syntax

A Regular Expression is a pattern of text that consists of ordinary characters (for example, letters a through z) and special characters, known as meta characters. The pattern describes one or more strings to match when searching a body of text. The Regular Expression serves as a template for matching a character pattern to the string being searched. This is a bit similar to what you are used to when for example searching in the project list in Confirmit, where you may use * as a wildcard. However, Regular Expressions is far more flexible and complex than that.

14.4.1.1 Ordinary Characters

Ordinary characters consist of all characters that are not explicitly designated as meta characters. This includes

- all upper- and lowercase alphabetic characters,
- all digits,
- all punctuation marks, and
- some symbols.

The simplest form of a Regular Expression is a single, ordinary character that matches itself in a searched string. For example, the single-character pattern

```
/a/
```

matches the letter a wherever it appears in the searched string.

You can combine a number of single characters together to form a larger expression.

```
/arm/
```

This expression describes a pattern with these three characters joined together. Notice that there is no concatenation operator. All that is required is that you just put one character after another.

The match will be found in the string

```
"Those people are harmless."
```

but not in the string

```
"Those people are my family."
```

Even though the second string has the characters a, r and m, they are not joined together, so no match is found.

14.4.1.2 Special Characters

There are a number of meta characters that require special treatment when trying to match them. To match these special characters, you must first escape them, that is, precede them with a backslash character (\).

The following table shows those special characters and their meanings. More detailed explanations on them will follow in the next chapters.

Special Character	Comment
`$`	Matches the position at the end of an input string. If the multi-line (`m`) property is set, `$` also matches the position preceding `\n` or `\r`. (newline or carriage return)
`( )`	Marks the beginning and end of a sub expression. Sub expressions can be captured for later use.
`*`	Matches the preceding sub expression zero or more times.
`+`	Matches the preceding sub expression one or more times.
`.`	Matches any single character except the newline character \n.
`[`	Marks the beginning of a bracket expression.
`?`	Matches the preceding sub expression zero or one time, or indicates a "non-greedy" quantifier.
`\`	Marks the next character as a special character, a literal, a back-reference, or an octal escape.
`^`	Matches the position at the beginning of an input string except when used in a bracket expression where it negates the character set. If the multi-line property is set, ^ also matches the position following \n or \r (newline or carriage return)
`{`	Marks the beginning of a quantifier expression.
`|`	Indicates a choice between two items (or).

To match any of these characters themselves, they have to be preceded with \:

Expression	Matches
`\$`	`$`
`\(`	`(`
`\)`	`)`

`*`	`*`
`\+`	`+`
`\.`	`.`
`\[`	`[`
`\?`	`?`
`\\`	`\`
`\^`	`^`
`\{`	`{`
`|`	`|`

```
/Confirmit\?/
```

matches the string "Confirmit?"

14.4.1.3 Non-Printable Characters

There are a number of useful non-printing characters that is occasionally used. The following table shows the escape sequences used to represent those non-printing characters:

Character	Meaning
`\cx`	Matches the control character indicated by x. For example, `\cM` matches a Control-M or a carriage return character. The value of x must be in the range of A-Z or a-z. If not, c is assumed to be a literal 'c' character.
`\f`	Matches a form-feed character.
`\n`	Matches a newline character.
`\r`	Matches a carriage return character.
`\s`	Matches any white space character including space, tab, form-feed, etc.
`\S`	Matches any non-white space character.
`\t`	Matches a tab character.
`\v`	Matches a vertical tab character.

x represents a character in the range A-Z or a-z.

14.4.1.4 Bracket Expressions

Many times, it's useful to match specified characters from a list. For example, you may want to search a string for chapter headings that are expressed as Chapter 1, Chapter 2, etc.

You can create a list of matching characters by placing one or more individual characters within square brackets ([and]). When characters are enclosed in brackets, the list is called a **bracket expression**.

Within brackets, as anywhere else, ordinary characters represent themselves, that is, they match an occurrence of themselves in the input text. Most special characters loose their meaning when they occur inside a bracket expression. There are some exceptions:

The] character ends a list if it is not the first item. To match the] character in a list, place it first, immediately following the opening [.

The \ character continues to be the escape character. To match the \ character itself, use \\.

Characters enclosed in a bracket expression match only a single character for the position in the Regular Expression where the bracket expression appears. The following Regular Expression matches 'Chapter 1', 'Chapter 2', 'Chapter 3', 'Chapter 4', and 'Chapter 5':

```
/Chapter [12345]/
```

If you want to express the matching characters using a range instead of the characters themselves, you can separate the beginning and ending characters in the range using the hyphen (-) character. The Unicode character value of the individual characters determines their relative order within a range. The following Regular Expression is equivalent to the bracketed list shown above.

```
/Chapter [1-5]/
```

When a range is specified in this manner, both the starting and ending values are included in the range. It is important to note that the starting value must precede the ending value in Unicode sort order.

If you want to include the hyphen character (-) in your bracket expression, you must do one of the following:

Escape it with a backslash:

```
[\-]
```

Put the hyphen character at the beginning or the end of the bracketed list. The following expressions match all lowercase letters and the hyphen:

```
[-a-z]
[a-z-]
```

Create a range where the beginning character value is lower than the hyphen character and the ending character value is equal to or greater than the hyphen. Both of the following Regular Expressions satisfy this requirement:

```
[!--]
[!-~]
```

i.e. characters from ! to – or from ! to ~.

If you want to find all the characters not in the list or range, you can place the caret (^) character at the beginning of the list. If the caret character appears in any other position within the list, it matches itself, that is, it has no special meaning. The following Regular Expression matches chapter headings with numbers different from 1,2,3,4 and 5 (and any other characters different from 1,2,3,4 and 5):

```
/Chapter [^12345]/
```

The same expressions above can be represented using the hyphen character (-).

```
/Chapter [^1-5]/
```

A typical use of a bracket expression is to specify matches of any upper- or lowercase alphabetic characters or any digits. The following JScript .NET expression specifies such a match:

```
/[A-Za-z0-9]/
```

Character	Description
`[xyz]`	A character set. Matches any one of the enclosed characters.
`[^xyz]`	A negative character set. Matches any character not enclosed.
`[a-z]`	A range of characters. Matches any character in the specified range.
`[^a-z]`	A negative range characters. Matches any character not in the specified range.

a, x, y and z represent characters.

14.4.1.5 Quantifiers

Sometimes, you don't know how many characters there are to match. In order to accommodate that kind of uncertainty, Regular Expressions support the concept of **quantifiers**. These quantifiers let you specify how many times a given component of your Regular Expression must occur for your match to be true.

The following table illustrates the various quantifiers and their meanings:

Character	Description
`*`	Matches the preceding subexpression **zero or more times**.
`+`	Matches the preceding subexpression **one or more times**.
`?`	Matches the preceding sub expression **zero or one time**.
`{n}`	Matches **exactly** n times, where n is a nonnegative integer..
`{n,}`	Matches **at least** n times, where n is a nonnegative integer
`{n,m}`	Matches **at least** n and **at most** m times. m and n are nonnegative integers, where n <= m.

n and m are integers.

With a large input document, chapter numbers could easily exceed nine, so you need a way to handle chapter numbers with more than one digit. Quantifiers give you that capability. The following JScript .NET Regular Expression matches chapter headings with any number of digits:

```
/Chapter [1-9][0-9]*/
```

The first bracket expression [1-9] makes sure that the first digit is in the range 1-9. The second bracket expression with quantifier [0-9]* searches for 0 or more digits in the range 0-9. The quantifier (*) appears after the range expression.

14.4.1.6 Anchors

All the examples so far have recognized patterns anywhere in a string. But you may want the patterns to match only if they e.g. appear at the beginning of the string. **Anchors** provide that capability.

Anchors allow you to fix a Regular Expression to either the beginning or end of a string. They also allow you to create Regular Expressions that occur either within a word or at the beginning or end of a word. The following table contains the list of Regular Expression anchors and their meanings:

Character	Description
`^`	Matches the position at the beginning of the input string. If the multi-line property is set, ^ also matches the position following \n or \r (newline or carriage return).
`$`	Matches the position at the end of the input string. If the multi-line property is set, `$` also matches the position preceding `\n` or `\r` (newline or carriage return).
`\b`	Matches a word boundary, that is, the position between a word and a space.
`\B`	Matches a non-word boundary.

You cannot use a quantifier with an anchor. Since you cannot have more than one position immediately before or after a newline or word boundary, expressions such as ^* are not permitted.

To match text at the beginning of a line of text, use the ^ character at the beginning of the Regular Expression. This is different from using the ^ within a bracket expression.

To match text at the end of a line of text, use the $ character at the end of the Regular Expression.

To use anchors when searching for chapter headings, the following JScript .NET Regular Expression matches a chapter heading with up to two following digits that occur at the beginning of a line and where there is no text after the heading:

```
/^Chapter [1-9][0-9]?$/
```

Matching word boundaries is a little different but adds a very important capability to Regular Expressions. A word boundary is the position between a word and a space. A non-word boundary is any other position. The following JScript .NET expression matches the first three characters of the word 'Chapter' because they appear following a word boundary:

```
/\bCha/
```

The position of the '\b' operator is critical here. If it's positioned at the beginning of a string to be matched, it looks for the match at the beginning of the word; if it's positioned at the end of the string, it looks for the match at the end of the word. For example, the following expressions match 'ter' in the word 'Chapter' because it appears before a word boundary:

```
/ter\b/
```

14.4.1.7 Alternation and Grouping

Alternation allows use of the | (pipe) character to allow a choice between two or more alternatives, a bit similar to or in Boolean expressions. Expanding the chapter heading Regular Expression, you can expand it to cover more than just chapter headings – for example sections as well. However, it's not as straightforward as you might think. You might think that the following expressions match one or two digits after either 'Chapter' or 'Section' , between the beginning and ending of a line:

```
/^Chapter|Section [1-9][0-9]?$/
```

Unfortunately, what happens is that the Regular Expressions shown above will have two different parts, so this will be equal to getting a match on either of these two expressions:

```
/^Chapter/
```

or

```
/Section [1-9][0-9]?$/
```

So it will match either the word 'Chapter' at the beginning of a line, or 'Section' and 1-2 numbers at the end of the line.

You can use parentheses to limit the scope of the alternation, that is, make sure that the alternation applies only to the two words, 'Chapter' and 'Section'. However, parentheses are tricky as well, because they are also used to create **sub expressions**. By taking the Regular Expressions shown above and adding parentheses in the appropriate places, you can make the Regular Expression match either 'Chapter 1' or 'Section 3'.

```
/^(Chapter|Section) [1-9][0-9]?$/
```

These expressions work properly except that a by-product occurs. Placing parentheses around 'Chapter|Section' establishes the proper grouping, but it also causes either of the two matching words to be captured for future use. So a sub match is captured. In this example we really do not need that sub match.

In the examples shown above, all you really want to do is use the parentheses for grouping a choice between the words 'Chapter' or 'Section'. You do not necessarily want to refer to that match later. We recommend that unless you really need to capture sub matches, do not use them. Your Regular Expressions will be more efficient since they will not have to take the time and memory to store those sub matches.

You can use ?: before the Regular Expression pattern inside the parentheses to prevent the match from being saved for possible later use. The following modification of the Regular Expressions shown above provides the same capability without saving the sub match.

```
/^(?:Chapter|Section) [1-9][0-9]?$/
```

There are times you'd like to be able to test for a pattern without including that text in the match. For instance, you might want to match the protocol in a URL (like http or ftp), but only if that URL ends with .com. Or maybe you want to match the protocol only if the URL does not end with .edu. In cases like those, you'd like to "look ahead" and see how the URL ends. A **lookahead** assertion is handy here.

There are two non-capturing meta characters used for **lookahead** matches:

A **positive lookahead**, specified using ?=, matches the search string at any point where a matching Regular Expression pattern in parentheses begins.

A **negative lookahead**, specified using ?!, matches the search string at any point where a string not matching the Regular Expression pattern begins.

If you want to search for the protocol in a url like "http://www.confirmit.com", but only if it ends with .com:

```
/^[^:]+(?=.*\.com$)/
```

Because of the anchor ^, the match is found at the beginning of the string. Then the first part

```
[^:] +
```

searches for one or more characters different from :. One or more because of the quantifier **+**, different from : because of the bracket expression [^:]

Then there is a positive lookahead:

```
(?=.*\.com$)
```

which searches through the string to find a match for

```
.* \.com$
```

Because of the anchor $ this has to be at the end of the string. . matches any character except the newline character, and because of the quantifier * we can have 0 or more of these characters before the last part, which is the character . (which has to be escaped with backslash \) followed by com – i.e. ".com".

But the match will be for the first part of the string, i.e. "http" in "http://www.confirmit.com".

Similarly, the expression

```
/^[^:]*(?!.*\.edu$)/
```

will search for the first characters until : is reached in a string not ending with ".edu".

Character	Description
`(pattern)`	Matches pattern and captures the match. The captured match can be retrieved from the resulting Matches collection, using the $0...$9 properties in JScript .NET.
`(?:pattern)`	Matches pattern but does not capture the match (it is not stored for possible later use).
`(?=pattern)`	Positive lookahead matches the search string at any point where a string-matching pattern begins. The match is not captured for possible later use. After a match occurs, the search for the next match begins immediately following the last match, not after the characters that comprised the lookahead.
`(?!pattern)`	Negative lookahead matches the search string at any point where a string not matching pattern begins. The match is not captured for possible later use. After a match occurs, the search for the next match begins immediately following the last match, not after the characters that comprised the lookahead.

14.4.1.8 Back-References

As explained above you can store a part of a matched pattern for later reuse. Placing parentheses around a Regular Expression pattern or part of a pattern causes that part of the expression to be stored into a temporary buffer.

Each captured sub match is stored as it is encountered from left to right in a Regular Expressions pattern. The sub matches are numbered beginning at 1 and continuing up to a maximum of 99 sub matches. Each different buffer can be accessed using

```
\n
```

where n is one or two decimal digits identifying a specific buffer, e.g. \1.

For example, back-references can be used to check for double occurrences of the same words, e.g. in a string like:

```
Scripting is is fun!
```

The following JScript .NET Regular Expression uses a single sub expression to check for duplicates:

```
/\b([a-z]+) \1\b/gim
```

The sub expression is everything between parentheses. That captured expression includes one or more alphabetic characters, as specified by [a-z]+. The second part of the Regular Expression (\1) is the reference to the previously captured sub match, that is, the second occurrence of the word. \b is used for word boundary, so that the check is done on complete words.

14.4.1.9 Digits and Word Characters

Character	Description
\d	Matches a digit character. Equivalent to [0-9].
\D	Matches a non-digit character. Equivalent to [^0-9].
\w	Matches any word character including underscore. Equivalent to A-Za-z0-9_].
\W	Matches any non-word character. Equivalent to [^A-Za-z0-9_].

14.4.1.10 Hexadecimal and Octal Escape Values and Unicode Characters

Character	Description
\x*n*	Matches n, where n is a hexadecimal escape value. Hexadecimal escape values must be exactly two digits long. For example, '\x41' matches "A". '\x041' is equivalent to '\x04' & "1". Allows ASCII codes to be used in Regular Expressions.
num	Identifies either an octal escape value or a back-reference (see Chapter 14.4.1.8 Back-References).
\u*n*	Matches n, where n is a Unicode character expressed as four hexadecimal digits. For example, \u00A9 matches the copyright symbol (©).

Since \ followed by a number *num* can represent both octal escape values and back-references, the following rules apply:

If *num* has one digit and *num* is preceded by at least *num* captured sub expressions, *num* is a back-reference. Otherwise, *num* is an octal escape value if *num* is an octal digit (0-7).

If *num* has two digits and *num* is preceded by at least *num* captured sub expressions, *num* is a back-reference. If the first digit is *n*, and *num* is preceded by at least *n* captures, *n* is a back-reference followed by a literal, the second digit. If neither of these applies, *num* matches an octal escape value when the number is an octal (both digits in the range 0-7).

If *num* has three digits, it matches an octal escape value when the first digit is 0-3 and the two last digits are octal digits (0-7).

14.4.2 Order of Precedence

Once you have constructed a Regular Expression, it is evaluated much like an arithmetic expression, that is, it is evaluated from left to right and follows an **order of precedence**. The following table illustrates, from highest to lowest, the order of precedence of the various Regular Expression operators:

Operator(s)	Description
\	Escape
(), (?:), (?=), []	Parentheses and Brackets

*, +, ?, {n}, {n,}, {n,m}	Quantifiers
^, $, \anymetacharacter	Anchors and Sequences
|	Alternation

14.4.3 The Regular Expression Object

We have already seen the two ways of instantiating a Regular Expression object:

```
var re = /pattern/{flags}
```

and

```
var re = new RegExp("pattern",{"flags"})
```

Regular Expression objects store patterns used when searching strings for character combinations. After the Regular Expression object is created, it is either passed to a `String` method, or a string is passed to one of the Regular Expression methods.

14.4.3.1 Methods

```
rgExp.exec(str)
```

`exec` executes a search on a string *`str`* using a Regular Expression pattern defined in an instance of a Regular Expression object *`rgExp`*, and returns an array containing the results of that search.

If the `exec` method does not find a match, it returns `null`. If it finds a match, exec returns an array. Element zero of the array contains the entire match, while elements 1 – n contain any sub matches that have occurred within the match.

```
rgExp.test(str)
```

`test` returns a Boolean value (`true` or `false`) that indicates whether or not a pattern defined in the Regular Expression *`rgExp`* exists in a searched string *`str`*.

The `test` method returns `true` if the pattern exists in the string, and `false` otherwise.

Example 67: Validation of the Format of a Phone Number

Let us say you want the respondent to specify his phone number in a particular format, allowing a digit or + as the first character and spaces between number sets, but no other characters. The following validation code uses a Regular Expression to check the formatting of the phone number, provided that the phone number is applied in an open text question with question ID *phone*:

```
var num : String = f("phone").get();
var re = /^(?:\d|\+)(?:\d| )+$/;
if(!re.test(num))
 {
  RaiseError();
  SetQuestionErrorMessage(LangIDs.en,"Please provide your phone number, only using
digits and space, and with + before your country code if it is a foreign number.");
 }
```

The Regular Expression starts the search at the beginning of the string (^). Then the first character should be either a digit or + (?:\d|\+). Because of ?: this sub expression is not stored. Then the rest of the string consists of one or more digits or spaces (?:\d|) until the end of the string is reached ($).

In this script there are no restrictions on number of spaces or digits.

Exercise 8: Restricting the Number of Digits in a Phone Number

Based on the previous example, please modify the Regular Expression so that it checks that the phone number is on the format xxx xxx xxxx – i.e. 3 groups of 3+3+4 digits separated with space. (This time with no + for country code).

See answer on page 178.

14.4.4 String Object Methods that Uses Regular Expression Objects

There are three methods of the `String` Object that uses Regular Expressions as input.

```
stringObj.match(rgExp)
```

`match` executes a search on a string using a Regular Expression pattern, and returns an array containing the results of that search.

If the `match` method does not find a match, it returns `null`.

If it finds a match, `match` returns an array. If the global flag (g) is not set, element zero of the array contains the entire match, while elements 1 – n contain any sub matches that have occurred within the match. If the global flag is set, elements 0 - n contain all matches that occurred.

```
stringObj.replace(rgExp, replaceText)
```

`replace` returns a copy of a string with text replaced using a Regular Expression. The string *`stringObj`* is not modified by the `replace` method.

`rgExp` can be an instance of a Regular Expression object or a String object or literal. If *`rgExp`* is not an instance of a Regular Expression object, it is converted to a string, and an exact search is made for the results; no attempt is made to convert the string into a Regular Expression.

`replaceText` is a String object or string literal containing the text to replace for every successful match of *`rgExp`* in *`stringObj`*. It can also be a function that returns the replacement text.

Returned from the `replace` method is a copy of *`stringObj`* after the specified replacements have been made.

```
stringObj.search(rgExp)
```

`search` returns the position of the first substring match in a Regular Expression search using the Regular Expression object *`rgExp`*.

The `search` method indicates if a match is present or not. If a match is found, search returns an integer value that indicates the index from where the match occurred. If no match is found, it returns
-1.

Example 68: Using Regular Expression to Replace Commas With Line Breaks

If you want to send an email and in the email text list the answers to a multi question from the survey, you can use `categoryLabels`. Converted into a string this will give the items separated by commas. If you want to have the answers on one line each instead of separated by commas you have to replace the commas with line breaks. If the email is sent as plain text, you then have to replace the commas with the special character \n.

This script will replace the commas in a listing of the answers given on a multi question brands:

```
var body : String = "";
body += "Here are the answers on the brands question:\n\n"

var brandsAnswers = f("brands").categoryLabels();
brandsAnswers = brandsAnswers.toString();
brandsAnswers = brandsAnswers.replace(/,/g,"\n");

body += brandsAnswers

SendMail("interviewer@confirmit.com",f("email"),"Answers",body);
```

If the mail is sent as HTML, you have to replace the commas with the HTML <br> tag instead:

```
var body : String = "";
body += "Here are the answers on the brands question:<br><br>"

var brandsAnswers = f("brands").categoryLabels();
brandsAnswers = brandsAnswers.toString();
brandsAnswers = brandsAnswers.replace(/,/g,"<br>");

body += brandsAnswers

SendMail("interviewer@confirmit.com",f("email"),"Answers",body,"","",0,0);
```

Example 69: *Postal Codes in the United Kingdom*

Postal codes in the UK can be on the following formats:

LN NLL
LLN NLL
LNN NLL
LLNN NLL
LLNL NLL

where L is a letter and N is a number. A postal code is one or two letters, followed by one or two numbers OR two letters followed by a number and a letter, AND then a space and a number and two letters.

A validation code for postal codes for UK could be like this, with the open text question *postalcode*:

```
var pcode : String = f("postalcode").get();
var re = /^(?:[a-z]{1,2}\d{1,2}|[a-z]{2}\d[a-z]) \d[a-z]{2}$/i;
if(pcode.search(re) == -1)
 {
  RaiseError();
  SetQuestionErrorMessage(LangIDs.en,"Please provide a postal code using a valid
format.");
 }
```

We allow both lower- and uppercase letters (i). For the first part we have two possibilities: The first being a combination of 1-2 letters and 1-2 digits (LN, LLN, LNN or LLNN), the second being 2 letters, 1 digit and 1 letter (LLNL). The expression within the parenthesis covers these options. The end of the string is equal for all possible postal codes: A blank, a number and two letters (NLL).

Exercise 9:

Make a script to validate a US zip code. Question ID: *zipcode*. US zip codes are 5 digit codes. So it is tempting to set up an open text numeric question with precision 5, and let the default validation do the work. However, the first number can be a zero (0), and to make sure that it is not removed when the value is stored (as it would for numeric questions), the question should be set up as an open text question with field width 5 instead of as a numeric. The easiest way to validate that all 5 characters are digits, is to use a Regular Expression.

See answer in Appendix A: Answers to Exercises.

14.5 The Array Object

We have already been using the `Array` object a lot. As we showed in chapter 7, there are two types of arrays – typed arrays and JScript arrays. The methods and properties here can be used on both types.

Typed arrays can be declared like this:

```
var arrayName : type[] = [value0, value1, …, valuen]
or
var arrayName : type[arrayLength];
```

or

```
var arrayName : type[];
arrayName = new type[arrayLength];
```

JScript arrays can be declared like this:

```
var arrayName = [value0, value1, …, valuen]
```

or

```
var arrayName = new Array(value0, value1, …, valuen);
```

or

```
var arrayName = new Array();
```

or

```
var weekday = new Array(arrayLength);
```

The `Array` Object has one property, the length of the array (number of items in the array):

```
arrayName.length
```

In this chapter we will look at the methods of the `Array` object.

14.5.1 Combining Arrays

```
arrayName.concat({item1{, item2{, . . . {, itemN}}}})
```

`concat` returns a new array consisting of a combination of *arrayName* and any other supplied items. The items to be added ($item_1$,...,$item_N$) to the array are added, in order, from left to right. If one of the items is an array, its contents are added to the end of *arrayName*. If the item is anything other than an array, it is added to the end of the array as a single array element.

14.5.2 Converting Arrays to Strings

```
array.join(separator)
```

join returns a string value consisting of all the elements of an array concatenated and separated by the specified separator character. If separator is omitted, the array elements are separated with a comma. If any element of the array is undefined or null, it is treated as an empty string ("").

```
array.toString()
array.valueOf()
```

Both toString and valueOf converts elements of an array to strings. The resulting strings are concatenated, separated by commas. This is the same as using join without specifying a separator (or specifying comma as separator).

Example 70: *Converting an Array to a String with Line Breaks between Elements*

This example is similar to Example 68:, using `categoryLabels` to list the answers to a multi question and include them in an email text. Instead of the default commas between the items listed in the array returned from `categoryLabels`, we want line breaks. If the email is sent as plain text, you then have to replace the commas with the special character `\n`.

This script will convert the array to a string and use line breaks as delimiter between the elements of a multi question brands:

```
var body : String = "";
body += "Here are the answers on the brands question:\n\n"

var brandsAnswers = f("brands").categoryLabels();
brandsAnswers = brandsAnswers.join("\n");


body += brandsAnswers

SendMail("interviewer@confirmit.com",f("email"),"Answers",body);
```

If the mail is sent as HTML, you have to replace the commas with the HTML <br> tag instead:

```
var body : String = "";
body += "Here are the answers on the brands question:<br><br>"

var brandsAnswers = f("brands").categoryLabels();
brandsAnswers = brandsAnswers.join("<br>");


body += brandsAnswers

SendMail("interviewer@confirmit.com",f("email"),"Answers",body,"","",0,0);
```

14.5.3 Removing and Adding Elements

array.pop( )

`pop` removes the last element from an array and returns it. If the array is empty, `undefined` is returned.

array.push({$item_1$ {$item_2$ {... {$item_N$ }}}})

`push` appends new elements ($item_1, ..., item_N$) to an array, and returns the new length of the array. The `push` method appends elements in the order in which they appear. If one of the arguments is an array, it is added as a single element. Use the `concat` method to join the elements from two or more arrays.

array.shift( )

`shift` removes the first element from an array and returns it.

array.unshift({$item_1${, $item_2$ {, ... {, $item_N$}}}})

`unshift` returns an array with specified elements ($item_1, ..., item_N$) inserted at the beginning. The items will appear in the same order in which they appear in the argument list.

14.5.4 Changing the Order of the Elements

array.reverse()

`reverse` returns an `Array` object with the elements reversed. The method reverses the elements of the `Array` object in place (*array*). It does not create a new `Array` object during execution. If the array

is not contiguous, the `reverse` method creates elements in the array that fill the gaps in the array. Each of these created elements has the value `undefined`.

```
array.sort({this.sortFunction})
```

`sort` returns an `Array` object with the elements sorted. `sortFunction` is optional and is the name of the function used to determine the order of the elements. Because Confirmit script nodes are wrapped inside a class, you have to use the keyword `this` to refer to the current instance (`this.sortFunction`). If the sorting function is omitted, the elements are sorted in ascending, ASCII character order. The `sort` method sorts the `Array` object in place; no new `Array` object is created during execution.

If you supply a function in the `this.sortFunction` argument, it must return one of the following values:

- A negative value if the first argument passed is less than the second argument.
- Zero if the two arguments are equivalent.
- A positive value if the first argument is greater than the second argument.

Example 71: Validating Grid with "Other, specify" Alternatives

Let us say you have a grid *q1* with two "other, specify"-items:

Quality

Please give us your impression of the quality of the products provided by these PC manufacturers:

	1 Low quality	2	3	4	5 High quality
Apple	○	○	○	○	○
Compaq	○	○	○	○	○
Dell	○	○	○	○	○
HP	○	○	○	○	○
IBM	○	○	○	○	○
Other provider 1:	○	○	○	○	○
Other provider 2:	○	○	○	○	○

In such a question, usually the "other, specify"-items should be not required, whereas the other items should be required. To achieve this, we have to set the "Not required" property on the grid question, and provide our own validation code instead.

There should also be validation checking that text is provided if a rating is provided for an "other, specify"-field and vice versa. For this, the standard "Other Specify Checking" can be used (just make sure it is turned on when generating web interview files). However, if the respondent accidentally rate one of the "other" items, there is no way for the respondent to unselect that row. Therefore we have included a removal of the radio button selection for the "other" items when they are not answered correctly.

Here is the answer list of this question with precodes used:

Save - Font - Size

Text | Answers | Scale | Validation | Masking | Preview | Results | Languages

Delete | Add | Add Predefined | Set Row Height | View

English	Precode	KeepPos	Other
Apple	1	☐	☐
Compaq	2	☐	☐
Dell	3	☐	☐
HP	4	☐	☐
IBM	5	☐	☐
Other provider 1:	97	☐	☑
Other provider 2:	98	☐	☑

This is a validation code that can be used on such a question:

```
var form = f("q1");
var precodes = a("q1").diff(set("97","98")).members(); //all precodes except "other"

var precodes = precodes.sort(this.NumSort); //sort the precodes so the items are
checked in the same order as they are displayed

var notAnswered = new Array(); //array to hold text of not answered items

for(var i : int = 0;i<precodes.length;i++)
 {
  var code = precodes[i];
  if(!form[code].toBoolean())
   {
    //add the label of not answered element to the array:
    notAnswered.push(form[code].label());
   }
 }

precodes = new Array("97","98");  //precodes of "other, specify"-elements

for(i=0;i<precodes.length;i++)
 {
  code = precodes[i];
  if(form[code].toBoolean() && !f("q1_"+code+"_other").toBoolean())
   {
    //remove answer
    form[code].set(null);
   }
 }

if(notAnswered.length > 0)
 {
  RaiseError();
  SetQuestionErrorMessage(LangIDs.en,"Please answer for all providers. There are
missing answer(s) for:<br>"+notAnswered.toString());
 }

//helper function for sorting
function NumSort(a,b)
 {
  var x = parseInt(a,10);
  var y = parseInt(b,10);
  return x-y;
 }
```

This example uses the `push` method to add elements to the end of an array. We use the `sort` method to sort the array. When the `precodes` array is set with the statement

```
var precodes = a("q1").diff(set("97","98")).members();
```

it is built from set expressions, and as we have learnt the order of the items within a set is insignificant. So the order the elements come in the array after using this expression is not necessarily the same order as in the answer list. To make sure we get them in the same order as in the answer list, so the listing in the error message does not become confusing to the respondents, we use the sort method with the helper function `NumSort` that converts its parameters (the precodes in the array) to numbers and then do subtraction and return the result. If the result is negative, the first precode is less than the second, if it is 0 they are equal and if the result is positive the first one is greater than the second. This is just as the description of the *sortFunction* above.

14.5.5 slice and splice

```
array.slice(start, {end})
```

`slice` returns a section of an array. *start* is the index to the beginning of the specified portion of the array. *end* is optional and is the index to the end of the specified portion of *arrayObj*. The `slice` method copies up to, but not including, the element indicated by *end*. If *start* is negative, it is treated as *length+start* where *length* is the length of the array. If *end* is negative, it is treated as *length+end* where *length* is the length of the array. If *end* is omitted, extraction continues to the end of *arrayObj*. If *end* occurs before *start*, no elements are copied to the new array.

```
array.splice(start, deleteCount, {item1{, item2{, ... {,itemN}}}})
```

`splice` removes elements from an array and, if necessary, inserts new elements in their place, returning the deleted elements. *start* is the index from which to start removing elements. *deleteCount* is the number of elements to remove. $item_1, ..., item_N$ are optional elements to insert into the array in place of the deleted elements.

The `splice` method modifies array by removing the specified number of elements from position *start* and inserting new elements. The deleted elements are returned as a new `Array` object.

15 Customizing Standard Error Messages

We have seen numerous examples on how to add your own validation code to your Confirmit surveys. In this chapter we will look into how to add your own texts to the standard answer checks that are provided in Confirmit.

The following standard answer checks (validations) are available in Confirmit:

Answer required checks.
All questions (except for ordinary multi questions without an exclusive item ("None of the above")) are by default required. You can set a question to be "Not required" in the properties of the question, or you can turn off the checking for all questions by deselecting "Answer required checks" when generating WI.

Exclusivity tests.
An "exclusive item" in the answer list of a multi question is an answer alternative with the "single punch" property set. (Typically a "None of the above" or "Don't know" answer alternative.) This means that the answer alternative cannot be answered in combination with any of the other items. The exclusivity check makes sure that the question has at least one answer, and that no exclusive item is selected in combination with other answers. This checking can be turned off by deselecting "Exclusivity tests" when generating WI.

Other-specify checking.
When the "other" property is set for an item in an answer list, a text box is included next to that item. The "other-specify" checking makes sure that there is a correspondence so that the text box has an answer only if that answer alternative is selected and visa versa. It can be turned off globally by deselecting "Other-specify checking" when generating WI.

Rank order tests.
For a multi or grid question with the "ordered" property set, the system checks that the answers constitute a set of consecutive integers starting at 1, and that all items have a rank. This default checking can be turned off globally by deselecting "Rank order tests".

Answer size tests for fixed-width fields.
If a field width is defined for a question, the system checks that the respondent's answer is within the limit. This cannot be turned off globally, because the database is set up according to the field width definitions, so the database cannot accept answers above this limit.

Numeric validation.
For open text and multi questions with the numeric property, the system checks that the answer consists of the symbols 0 to 9 only, and is within the limits defined in precision, scale as well as lower and upper limit. This validation cannot be turned off globally because the database is set up according to these settings and cannot accept non-numeric answers that do not correspond to these settings.

All of these types of validation have their own error messages provided in Confirmit. These error messages are provided in a number of languages, but are the same for all users of a Confirmit installation. They can be changed globally on a Confirmit server installation, but not for particular surveys.

If you want something different than the standard error messages, you have to add code in the validation code field of the question where the validation applies. In this chapter we will explain a number of functions that make it easier to build customized error messages for the default answer checks.

15.1 Functions for Standard Validation

The following functions can be used to find out if any of the standard validation in Confirmit has found an error in the respondent's answer(s):

Function	Description

`QuestionErrors()`	returns true if an error has been raised during any validation, false otherwise.
`MissingRequiredError()`	returns true if one or more required answers to a question are missing, false otherwise.
`ExclusivityError()`	returns true if an exclusive answer ("single punch") in a multi has been selected together with any other alternative, false otherwise.
`NotSpecifiedError()`	returns true if an alternative in an Other-Specify construct has been selected/answered without filling in the associated "Specify" text box, false otherwise.
`NotSelectedError()`	returns true if a "Specify" value has been entered in the text box of an Other-Specify construct without selecting/answering the associated alternative, false otherwise.
`RankError()`	returns true if the "Ordered" property has been selected for the form and the answers to the questions are non-numeric, do not start at 1 or are non-consecutive, false otherwise.
`SizeError()`	returns true if one or more open-ended answers exceeded the maximum length specified in field width, false otherwise.
`NumericError()`	returns true if the “Numeric” property has been selected and the answer is not numeric, false otherwise.
`PrecisionError()`	returns true if the “Numeric” property has been selected and the answer cannot be stored within the defined precision, false otherwise.
`ScaleError()`	returns true if the “Numeric” property has been selected and the answer contains more decimals than in the defined scale, false otherwise.
`RangeError()`	returns true if the “Numeric” property has been selected and the answer is outside the defined range, false otherwise.

Example:

```
if(MissingRequiredAnswers())
 {
  SetQuestionErrorMessage(LangIDs.en,"Please provide an answer.");
 }
```

When one of these functions returns true, an error situation has already been flagged, so you do not have to use the RaiseError function.

15.2 Template Based Error Messages

The system allows you to use templates for your error messages in custom validation. For each error type, various elements are filled in that can improve the information content of the messages you report to the respondents. Instead of simply using fixed strings for error messages, the ErrorTemplate function can be used to “plug in” information about the current question, like question texts, current answers etc.

```
ErrorTemplate(spec)
```

spec is a string with the template specification. Inside the spec string you may use different elements that are singled out with caret (^) as pre- and suffix, e.g.

```
^MISSING^
```

Example:

```
SetQuestionErrorMessage(LangIDs.en, ErrorTemplate("^MISSING^ has not been
answered."));
```

This will set the text of the error message to "label(s) has not been answered"

Templates are available for all the standard validation. There are usually several versions of the templates, which differ in how they separate words when they are listed. Some only use commas, some use commas, but "and" or "or" between the last two items. "and"/"or" is also available in several languages, so e.g. in German they will be replaced with "und"/"oder" and so on.

15.2.1 Answer Required Checks

```
MissingRequiredError()
```

returns true when one or more required answers to a question are missing. The following template elements can be used when MissingRequiredError returns true:

Template	Description
`MISSING`	Labels of all questions that lack answers, comma separated.
`MISSING_AND`	Same as above, but with the word “and” separating the two last items.
`MISSING_OR`	Same as above, but with the word “or” separating the two last items.

Example:

```
if(MissingRequiredError())
 {
  SetQuestionErrorMessage(LangIDs.en,
  ErrorTemplate("Please select an answer for ^MISSING_AND^."));
 }
```

15.2.2 Exclusivity Tests

```
ExclusivityError()
```

returns true if an exclusive answer ("single punch") in a multi has been selected together with any other alternative. The following template elements can be used when ExclusivityError returns true:

Template	Description
`SEL_EXCL`	The labels of all exclusive (single punch) answers that were selected.
`SEL_EXCL_AND`	Same as above, but with the word “and” separating the two last items.

`SEL_EXCL_OR`	Same as above, but with the word “or” separating the two last items.
`SEL_NEXCL`	The labels of all non-exclusive (multi punch) answers that were selected.
`SEL_NEXCL_AND`	Same as above, but with the word “and” separating the two last items.
`SEL_NEXCL_OR`	Same as above, but with the word “or” separating the two last items.
`DEF_EXCL`	The labels of all exclusive (single punch) answers, regardless of which of them that were selected.
`DEF_EXCL_AND`	Same as above, but with the word “and” separating the two last items.
`DEF_EXCL_OR`	Same as above, but with the word “or” separating the two last items.
`DEF_NEXCL`	The labels of all non-exclusive (multi punch) answers, regardless of which of them that were selected.
`DEF_NEXCL_AND`	Same as above, but with the word “and” separating the two last items.
`DEF_NEXCL_OR`	Same as above, but with the word “or” separating the two last items.

Examples:

```
if(ExclusivityError())
 {
  SetQuestionErrorMessage(LangIDs.en,
  ErrorTemplate("Please do not check ^SEL_EXCL_OR^ if you check other answers to the
question."));
 }

if(ExclusivityError())
 {
  SetQuestionErrorMessage(LangIDs.en,
  ErrorTemplate("^DEF_EXCL_AND^ can not be combined with any of the other answers."));
 }
```

15.2.3 Other-Specify Checking

```
NotSpecifiedError()
```

returns true if an alternative in an Other-Specify construct has been selected/answered without filling in the associated "Specify" text box.

```
NotSelectedError()
```

returns true if a "Specify" value has been entered in the text box of an Other-Specify construct without selecting/answering the associated answer alternative.

The following template elements are defined when one of these functions returns true:

Template	Description
`OTHER`	Label of the alternative/input that requires specification in an Other-Specify construct.
`SPEC`	The specification

Examples:

```
if(NotSpecifiedError())
 {
  SetQuestionErrorMessage(LangIDs.en,ErrorTemplate("Please specify if ^OTHER^ is
chosen."));
 }

if(NotSelectedError())
 {
  SetQuestionErrorMessage(LangIDs.en,
  ErrorTemplate("If you specify ^OTHER^ then please select the ^OTHER^ option."));
 }
```

15.2.4 Rank Order Tests

```
RankError()
```

returns true if the "Ordered" property has been selected for the form and the answers to the questions are non-numeric, do not start at 1 or are non-consecutive. The following template elements are defined when RankError returns true:

Template	Description
`RANK_MIN`	The label of the first member of the ranking scale, or 1 if the question is open ended.
`RANK_MAX`	The label of the nth member of the ranking scale, or n if the question is open ended, where n is the number of items to rank.

Example:

```
if(RankError())
 {
  SetQuestionErrorMessage(LangIDs.en,
  ErrorTemplate("Please enter consecutive answers in the range ^RANK_MIN^ to
^RANK_MAX^."));
 }
```

15.2.5 Answer Size For Fixed-Width Fields

```
SizeError()
```

returns true if one or more open-ended answers exceeded the maximum length specified in field width. The following template elements are defined when SizeError returns true:

Template	Description
`TOO_LONG`	The labels of the inputs that were too long.
`TOO_LONG_AND`	Same as above, but with the word “and” separating the two last items.
`MAX_SIZE`	The maximum allowed size.

Example:

```
if(SizeError())
 {
  SetQuestionErrorMessage(LangIDs.en,
  ErrorTemplate("The answers to ^TOO_LONG_AND^ were longer than ^MAX_SIZE^ characters
and have been truncated. Please review."));
 }
```

15.2.6 Numeric Validation

```
NumericError()
```

returns true if the "Numeric" property has been selected and the answer is not numeric. The following template elements are defined when NumericError returns true:

Template	Description
`NUMERIC_ERRORS`	The labels of all elements with a numeric error.
`NUMERIC_ERRORS_AND`	Same as above, but with the word “and” separating the two last items.

Example:

```
if(NumericError())
 {
  SetQuestionErrorMessage(LangIDs.en,
  ErrorTemplate("Please make sure that the answers to ^NUMERIC_ERRORS^ are numbers
only."));
 }
```

15.2.7 Precision Error Tests

```
PrecisionError()
```

returns true if the "Numeric" property has been selected and the answer cannot be stored within the defined precision. The following template elements are defined when PrecisionError returns true:

Template	Description
`PRECISION`	The defined precision
`PRECISION_ERRORS`	The labels of all elements with a precision error.
`PRECISION_ERRORS_AND`	Same as above, but with the word “and” separating the two last items.

Example:

```
if(PrecisionError())
 {
  SetQuestionErrorMessage(LangIDs.en,
  ErrorTemplate("Please enter a number with no more than ^PRECISION^ digits for
^PRECISION_ERRORS^."));
 }
```

15.2.8 Scale Error Tests

```
ScaleError()
```

returns true if the "Numeric" property has been selected and the answer contains more decimals than in the defined scale. The following template elements are defined when ScaleError returns true:

Template	Description
`SCALE`	The defined scale
`SCALE_ERRORS`	The labels of all elements with to many decimals.
`SCALE_ERRORS_AND`	Same as above, but with the word “and” separating the two last items.

Example:

```
if(ScaleError())
 {
  SetQuestionErrorMessage(LangIDs.en,
  ErrorTemplate("Please enter answers with no more than ^SCALE^ decimals for
^SCALE_ERRORS_AND^."));
 }
```

15.2.9 Range Error Tests

```
RangeError()
```

returns true if the "Numeric" property has been selected and the answer is outside the defined range. The following template elements are defined when RangeError returns true:

Template	Description
`RANGE_MIN`	The label of the defined minimum value.
`RANGE_MAX`	The label of the defined maximum value.
`RANGE_ERRORS`	The labels of all elements with too many decimals.
`RANGE_ERRORS_AND`	Same as above, but with the word “and” separating the two last items.

Example:

```
if(RangeError())
 {
  SetQuestionErrorMessage(LangIDs.en,
  ErrorTemplate("Please enter answers in the range ^RANGE_MIN^ to ^RANGE_MAX^."));
 }
```

16 Useful ASP.NET Intrinsic Objects

This chapter contains examples of some ASP.NET objects that you may find useful in your scripts, with examples of how they can be used. It is not intended as a comprehensive description of them. For more information, consult an ASP.NET reference.

16.1 Request

Information can be sent in to the server from the user either from a web form (as the questions in the survey), passed in with the web address URL or stored in a cookie. This, as well as other information about the incoming request can be retrieved using the Request object.

We will look into some of the properties of the Request object. All of these return values of the type `NameValueCollection`. This is a type that represents a sorted collection of associated `String` keys and `String` values that can be accessed either with the key or with the index (integer from 0). This means that their values can be referenced in the same way as you refer to values in an `Array` (See Chapter 7 Arrays), but also using their `String` key, e.g.

```
var aValue = Request("arg");
```

This will look through all values sent in with the request to find a key that matches the name specified in *`arg`* and return the corresponding value. If there is no match for `arg`, `null` will be returned. This means that you can use the expression

```
Request("arg")!=null
```

to check if it exists before using the value returned.

A more robust approach than using the `Request` object directly is to use the separate properties of the `Request` object to prevent naming conflicts. In the next chapters we will describe some of the most useful properties.

16.1.1 Form

```
Request.Form
```

will give the collection of form values submitted using "POST"

16.1.2 Querystring

```
Request.Querystring
```

will give the collection of values supplied on the URL or from a form submitted with "GET". If you want to send values into the survey with the url, they can be included after the question mark after *project_ID*.aspx, separated by ampersands (&), e.g.

```
http://survey.confirmit.com/wix/project_ID.aspx?variable1=value1&variable2=value2
```

The keys here are $variable_1$ and $variable_2$, so using the key $variable_1$ as in

```
Request.Querystring("variable1")
```

with the url above will return the value $value_1$.

16.1.3 Cookies

```
Request.Cookies
```

will return the cookies stored from the domain of the Confirmit server. It returns an HttpCookieCollection object, which is described closer in Chapter 16.2.2 Cookies. See also Example 73: Using Cookies.

16.1.4 ServerVariables

```
Request.ServerVariables
```

will give the information sent in the header or internal server values.

Here is a table listing an extract of the possible server variables in the collection returned from the `ServerVariables` property:

Variable	Meaning
`ALL_HTTP`	All HTTP headers sent by the client., with their names capitalized and prefixed with `HTTP_`
`ALL_RAW`	Retrieves all headers in raw form (the form they were sent to the server by the client).
`CONTENT_LENGTH`	The length of the content as given by the client.
`CONTENT_TYPE`	The MIME type of the request, such as www-url-encoded for a form being posted to the server.
`GATEWAY_INTERFACE`	The Common Gateway Interface (CGI) supported on the server.
`HTTP_<HeaderName>`	The value stored in the header *HeaderName*. Any header other than those listed in this table must be prefixed by HTTP_ in order for the ServerVariables collection to retrieve its value. Note The server interprets any underscore (_) characters in *HeaderName* as dashes in the actual header. For example if you specify HTTP_MY_HEADER, the server searches for a header sent as MY-HEADER.
`HTTP_ACCEPT`	The MIME types the client can accept.
`HTTP_ACCEPT_LANGUAGE`	The languages accepted by the client.
`HTTP_ACCEPT_ENCODING`	The compression encoding types supported by the client.
`HTTP_CONNECTION`	Indicates whether the connection allows keep-alive functionality.
`HTTP_COOKIE`	Returns the cookie string that was included with the request.
`HTTP_USER_AGENT`	Returns a string describing the browser that sent the request.
`HTTP_HOST`	The host name of the server.
`HTTPS`	Returns "`On`" if HTTPS was used for the request and "`Off`" if not.
`HTTPS_KEYSIZE`	Number of bits in the encryption used to make the SSL connection. For example, 128.
`HTTPS_SECRETKEYSIZE`	Number of bits in server certificate private key. For example, 1024.
`HTTPS_SERVER_ISSUER`	Issuer field of the server certificate.

HTTPS_SERVER_SUBJECT	Subject field of the server certificate.
LOCAL_ADDR	The IP address of the server which is handling the request.
QUERY_STRING	Query information stored in the string following the question mark (?) in the HTTP request.
REMOTE_ADDR	The IP address of the remote host making the request.
REMOTE_HOST	The host name of the client making the request, if available.
REQUEST_METHOD	The type of the HTTP request made: "GET", "POST" or "HEAD".
SERVER_NAME	The server's host name,
SERVER_PORT	The port number to which the request was sent.
SERVER_PORT_SECURE	A string that contains either 0 or 1. If the request is being handled on the secure port, then this will be 1. Otherwise, it will be 0.
SERVER_PROTOCOL	The HTTP protocol and version in use on the server.
SERVER_SOFTWARE	The name and version of the web server software running on the server.

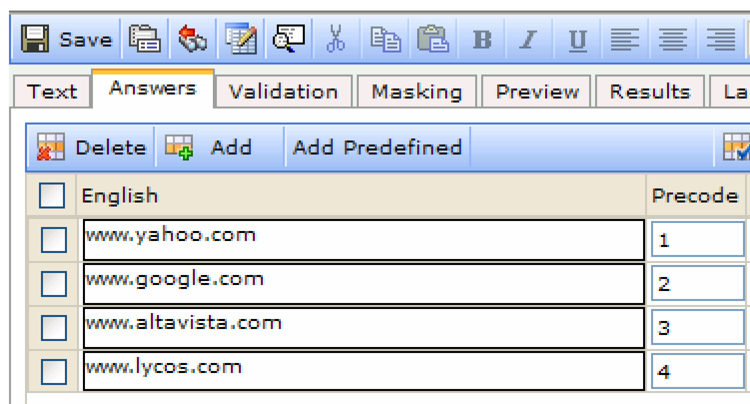
Example 72: *Sending in Values with the URL*

Say you have a pop-up survey that is triggered from pop-up scripts on several sites. You want to identify which site the respondents were surfing when the pop-up appeared.

In the pop-up scripts you should use different survey links for the different sites, like this:

http://survey.confirmit.com/wix/<project ID>.aspx?site=1
http://survey.confirmit.com/wix/<project ID>.aspx?site=2
http://survey.confirmit.com/wix/<project ID>.aspx?site=3
http://survey.confirmit.com/wix/<project ID>.aspx?site=4

Insert a hidden single question *source* in your questionnaire. The answer list should have the different sites you use and precodes that correspond to the values you use in the urls, e.g.



To set this hidden question based on the values sent in with the url, use this code in a script node at the beginning of the questionnaire.

```
f("source").set(Request("site"));
```

Alternatively, you can use the `QueryString` property:

```
f("source").set(Request.QueryString("site"));
```

It is very important that the script node with Request is the first node in the questionnaire, because once the respondent moves to the next page, the values are lost:

Now *source* can be used for reporting, quotas, logic etc. just as an ordinary question.

16.2 Response

Response is used to write content to the client (the respondent's browser), including headers and cookies. Here we will only cover the `Write` method and the property used to set cookies.

16.2.1 Write

```
Response.Write(arg)
```

You may use the `Write` method for debugging purposes, for example to get values of variables you are using in your script written to the screen when testing to help identifying why the script does not work as intended. For other purposes it is recommended to use response piping with ^-s inside one of the text fields of a question or an info instead, since you will then have better control of where on the page the text you output is placed.

`arg` can be any type of object, as it will be converted to a string when written to the client.

16.2.2 Cookies

```
Response.Cookies
```

The `Cookies` property indicates the cookies collection, which allows addition of cookies to the outgoing stream, i.e. adding cookies to the client from a Confirmit survey. It can e.g. be used to prevent respondents to an open survey from taking the survey more than once, as showed in Example 73: Using Cookies.

The class `HttpCookiesCollection` provides a wrapper for a collection of cookies. You use this to set and modify properties and values in the cookies. Most useful for us is the `Add` method:

```
cookiesCollection.Add(ccokie)
```

The `Add` method allows the addition of a single cookie to the collection. The parameter *`cookie`* is an object of the `HttpCookie` class. `HttpCookie` has two different constructors:

```
var cookie = new HttpCookie(string);
```

which creates and names a new cookie.

```
var cookie = new HttpCookie(string1, string2);
```

which creates, names (*`string1`*), and assigns a value (*`string2`*) to a new cookie.

There is currently a problem with the JScript.NET compiler that gives a type mismatch error when compiling with an `HttpCookie` object. For this reason the variable should be defined as `Object` to avoid this checking:

```
var cookie : Object = new HttpCookie(string);
```

or

```
var cookie : Object = new HttpCookie(string1, string2);
```

Here are the most relevant properties of `HttpCookie`:

```
cookie.Expires
```

The `Expires` property indicates the expiration date and time of the cookie, as a .NET `DateType` value (a bit different than the JScript .NET `Date` object. Consult a .NET reference for more details.) After this expiration date and time, the cookie will not be sent with the request, so you can not retrieve it anymore.

```
cookie.Name
```

`Name` indicates the name of the cookie.

```
cookie.Path
```

Cookies are specific to the site they originate from, so client browsers only send cookies to the DNS domain from which they were created – i.e. a www.microsoft.com cookie can not be picked up by survey.confirmit.com. But you can set a cookie to indicate the directory path on the domain that should receive it using the `Path` property. Using a Path of `"/"` indicates that all directory paths on the server should have access to the cookie.

```
cookie.Value
```

The Value property indicates the value for the cookie.

The majority of cookies are used as a single name and value. However, a cookie can have more than one value. The `Values` property allows you to set several values and also retrieve them again as a `NameValueCollection`:

```
cookie.Values
```

Here is an example on how to set a cookie:

```
var myCookie : Object = new HttpCookie("LastVisit");

var now : DateTime = DateTime.Now; //current time



myCookie.Value = now.ToString(); //convert to string

myCookie.Expires = now.AddMonths(6);

myCookie.Path = "/";

Response.Cookies.Add(myCookie);
```

The value of the cookie can later be fetched like this:

```
var myCookie : Object = Request.Cookies("LastVisit");

if(myCookie != null)

 {

  var last = myCookie.Value;

  //The variable last will now have the timestamp of the last access

 }
```

Example 73: *Using Cookies to Limit Access to an Open Survey*

Sometimes the sample for a survey is not known in advance, so that you can not set up the survey as a limited survey with cryptic, individual links. Then you have to distribute the open link to the survey instead. However, you can use cookies to try to prevent the respondents from answering more than once. This is not a bullet-proof method of course, because respondents may have set their browsers to not accepting cookies, they may delete cookies and they may also respond from different PCs,

You can set the cookie from a script node anywhere in the survey depending on how far you think the respondents should have answered before not being allowed to reenter. A cookie using the project id as name can be set with this script:

```
var myCookie : Object = new HttpCookie(CurrentPID());

var now : DateTime = DateTime.Now; //current time



myCookie.Value = "true";

myCookie.Expires = now.AddMonths(3); //set the cookie to expire 3 months later

myCookie.Path = "/";

Response.Cookies.Add(myCookie);
```

In the beginning of the survey you may then have a condition that uses this expression to check if the cookie is present:

```
Request.Cookies(CurrentPID()) != null
```

If we also wanted to check the value we would also add that to the expression (not really necessary in this scenario):

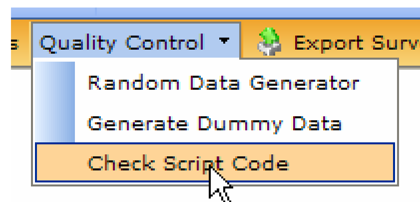
```
Request.Cookies(CurrentPID()) != null && Request.Cookies(CurrentPID()).Value == "true"
```

17 Testing Survey Scripts

All scripts should be carefully tested before setting a survey live. Confirmit provides a few helpful tools to use when testing scripts.

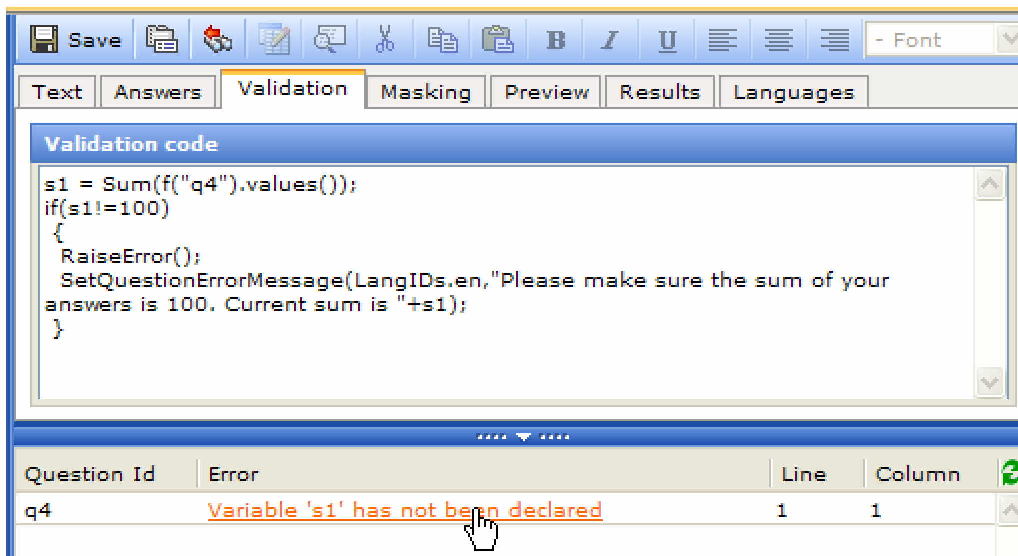
17.1 Check Script Code

You may use the Check Script Code functionality to check for syntactical errors in your script code. This is also done when you launch the survey in test or production mode, but using this functionality give you a quicker way of checking the script syntax because you do not have to generate the database and interview files.



Typical syntactical errors that may be found using the script checker is referencing undeclared variables and functions, brackets that are not matching - [ ], { } or ( ) - and similar.

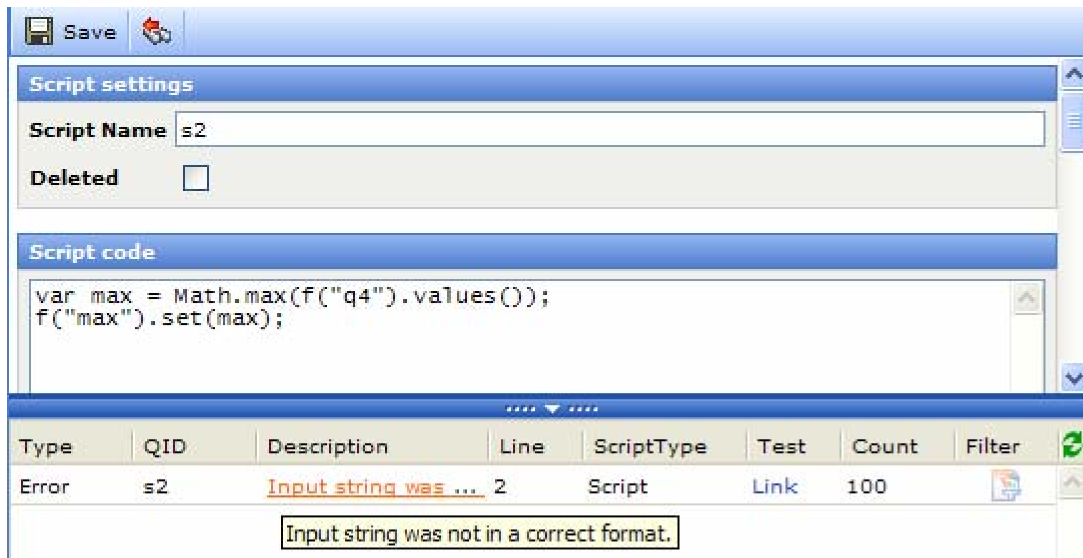
The errors found will be listed at the bottom of the screen, and by clicking each error you can open the script node, question or condition with the error.



17.2 Random Data Generator

However, you may have scripts that are syntactically correct, but which will fail run-time. For example, this could be non-existing question ids as parameter in the `f` function or trying to set a string value in a numeric question. Often these errors happen only for certain combinations of answers. To be able to test out as many combinations as possible, you can use the Random Data Generator functionality. (This is a Confirmit add-on. If you do not have access to this functionality, please contact your Confirmit account manager for more information.)

The Random Data Generator lets you run a number of automated test interviews with randomly selected responses. All script code in script nodes, masks, conditions, validation code and response piping will be executed, and any script errors will be recorded and listed for inspection at the end of the RDG run.



The list of errors from the random data generator is clickable, and clicking on the description will open the script, question or info where the error appeared. (In the example above, the problem is that the input to the `Math.max` method is not a list of numbers. In the code above that method returns `NaN` (not a number), which makes the script fail when attempting to set the question *max*, which has the numeric property. A better solution would be to use the Confirmit `Max` function (see Chapter 10.1.1.4 Max and Min), which automatically converts the values to numbers before finding the maximum.)

To help debugging you can use the results tab on the questions to see top line results for each question. It is also possible to filter these results so that you see the results only for those that got a particular error by clicking the filter icon in the rightmost column. You can also click the link in the "Test" column to access the first interview that experienced the error.

17.3 Tips for Debugging

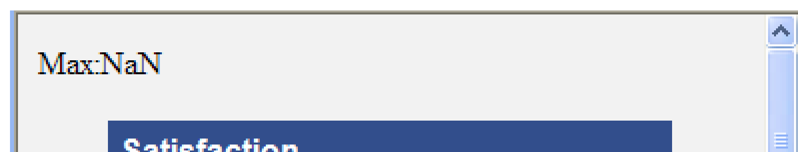
Sometimes it is hard to find out why a script is failing. Then it can be helpful to use the `Response.Write()` method to output some variables to the screen. For example, in the script above which failed when trying to set the max, question, we could comment out the last line and instead output the `max` variable to screen:

```
var max = Math.max(f("q4").values());

Response.Write("Max="+max);

//f("max").set(max);
```

The text from the `Response.Write` call will appear at the top of the page before any interview HTML. The result from the script above will look something like this:



If an error appears inside of a loop statement, it often helps to use Response.Write to help identify in which iteration the script fails:

```
for(var i : int = 0; i<precodes.length; i++)

 {

  Response.Write("i = "+i+"<br>");
```

```
< ... >

}
```

Other tricks to simplify debugging is to use `/*` and `*/` to comment out part of the code to help identify exactly where the error occurs. This can be done as a "binary search": First you comment out ½ of the code in a script. If the error still occurs, you know that it is in the other half. Then you can comment out half of that section – and so on.

It can also be very helpful to use Top line reporting or the "results" tab on a question, the "Edit" functionality under "Survey Data" or doing a data export of the responses with "error" status to look into what answers actually are stored in the question(s) involved in the script code.

18 Programming Conventions

Programming conventions are important to programmers, because they make scripts easier to read and understand. Writing scripts that are easy to understand will make searching for errors easier and will make it easier to re-use scripts. If programmers agree on a common way of writing scripts, it will be much easier to read the scripts of other programmers. Even the programmer that originally wrote a script may have problems understanding his or her own code after a few months unless it is written and commented in a way that makes it easy to read.

18.1 Comments

It is recommended to have extensive use of comments in your scripts. They will help you and others to more quickly understand the way your code works and how to use it. See chapter 3.

18.2 Naming Conventions

When naming variables and functions, always try to use as descriptive names as possible. If an array holds the precodes from a question, call it something like precodes or q1Precodes or similar, not just e.g. a. Similarly, a function that copies a multi question should be called something like CopyMulti, not f1. There is no point in making short names in JScript .NET.

A variable name must start with a lower or upper case letter or underscore, and continue with letters, digits or underscore. However, it is recommended to follow the following naming conventions:

- variable names and names of methods and properties should start with a lowercase letter in the first word. The next words should open with an uppercase letter. Examples: gridPrecodes, availableCodes, randomNumber, indexOuterLoop

- function names and names of objects should start with an uppercase in the first word, and have uppercase letters in the beginning of all subsequent words. Examples: CopyMulti, SelectItems, CalculateAverage.

18.3 Spaces and Line Breaks

Except from within string expressions, you can use line breaks and spaces as much as you like when writing JScript .NET code. Use this to make your code easier to read. Add a few line breaks between different parts of your scripts, and use line breaks in if-then-else constructs and loops to make it easier to understand the routing in your scripts.

```
while (condition)
 {
  <statements>
  if(condition)
   {
    <statements>
   }
  else
   {
    <statements>
   }
  <statements>
 }
```

With a structure like this it is easier to see where the then- and else-branch of the if-condition ends, and what statements belong to the different parts.

18.4 Curly Brackets

It is recommended that you always use the curly brackets ({ and }) in the if, switch, while, do while, for and function statements. If you have just one statement inside the curly brackets, they are optional, but

it is recommended that you make it a rule to always use them, limiting the possibility of making mistakes.

Example:

```
if(condition)
 <statement1>;
```

is the same as

```
if(condition)
 {
  <statement1>;
 }
```

so you may find it convenient to drop the curly brackets. But you may soon need to add another statement. If you add it like this:

```
if(condition)
 <statement1>;
 <statement2>;
```

or

```
if(condition)
 <statement1>; <statement2>;
```

it will be equivalent to

```
 if(condition)
 {
  <statement1>;
}
<statement2>;
```

This will be very different from

```
if(condition)
 {
  <statement1>;
  <statement2>;
}
```

18.5 Semi Colon

Statements are separated with semi colon (;) in JScript .NET. However, if you have line breaks between the statements, the semi colon is not required. However, it is recommended to always use it anyway, in case a line break is removed. You may have observed that all examples in this documentation use semi colons between the statements.

However, when working with the if, switch, while, do while, for and function statements you have to be careful with the semi colon. For example, remember that the while statement includes the statements within the curly brackets. If you place a semicolon just after the condition, like this:

```
while (condition);
 {
  <statements>
 }
```

you will actually end up with a loop that never terminates. The semicolon will be interpreted as the end of an empty statement. It will be this empty statement that will be executed until the condition is false, not the statements within the curly brackets. The code that executes will be similar to this:

```
while (condition)
 {
 }
<statements>
```

The condition is likely never to return false, since no statements are executed. This means that the loop will never terminate. This will cause a time-out for the respondent. But notice that there are no syntactical errors in the script, so you will not receive any error message on the script as such.

18.6 The Step by Step Approach

It is possible to do a lot of operations in one long statement. However, this makes the scripts hard to read.

Compare for example the following code, which randomly picks one of the precodes in an array precodes,

```
selectedCode = precodes[Math.floor(Math.random()*precodes.length));
```

with this code:

```
randomNumber = Math.random()*precodes.length;
randomIndex = Math.floor(randomNumber);
selectedCode = precodes[randomIndex];
```

The first code will be a bit more efficient than the latter, because you do not have to store values in the variables randomNumber and randomIndex. However, because it will be much easier to understand what happens in the last one, that approach is recommended.

To increase readability, use temporary variables and do the calculations step by step.

18.7 Writing Efficient Code

Here are a few tips on how to write code that is efficient:

- Define types on your variables, constants and functions when possible.
- Always make sure that the script does not run through unnecessary iterations in a loop, or unnecessary statements in an iteration. Use `break` or `continue` to terminate or skip to the next iteration.
- If you need to call the `f` function for the same question more than once in a script, store it in a variable and refer to that variable instead.
- Avoid using calls to user-defined functions in precode masks. Set a hidden multi question instead and refer to it with the `f` function.
- Use methods of the form objects like `domainValues` and `categories` when possible instead of hard coding precodes(1,2,3,…) in your scripts.
- Use short-circuiting evaluations in expressions involving logical and (`&&`) and or (`||`): Place conditions most likely to be `true` first for the logical or (`||`) operator, and conditions most likely to be `false` first for the logical and operator (`&&`).
- Only use local variables in functions (use the `var` keyword).

APPENDIX A: ANSWERS TO EXERCISES

Exercise 1:

```
"207 is not the same as 207, but this isn't really true"
```

Exercise 2:

a. x=5,y=9,z=0

1. x=5,y=9,z=8

1. x=4,y=33,z=8

1. x=35,y=35,z=8

Exercise 3:

a. Code sample 1: x=5 and y=5
Code sample 2: x=5 and y=6

1. Code sample 1: x=5 and y=5
Code sample 2: x=5 and y=5

Exercise 4:

a. f("importance")["2"].valueLabel()

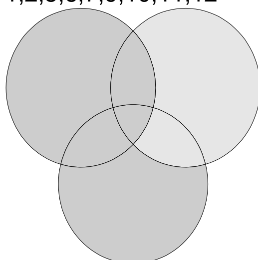
1. f("importance")["2"].label()

Exercise 5:

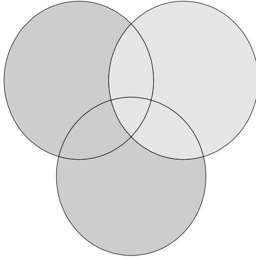
```
if(!f("q2").toBoolean())
 {
  var precodes = f("q2").domainValues();
  for(var i : int = 0;i<precodes.length;i++)
   {
    var code = precodes[i];
    f("q2")[code].set("3");
   }
 }
```

Exercise 6:

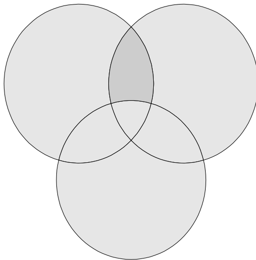
a. 1,2,5,6,7,9,10,11,12



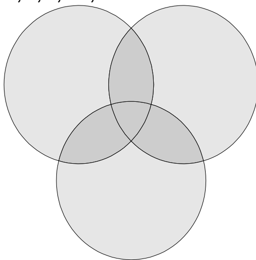
1. 1,2,6,7,9,10,12



1. 11



1. 2,4,5,10,11



Exercise 7:

```
var d = IsDateFmt(f("date4").get(),"MM/DD YYYY");

if(!d)
 {
  RaiseError();
  SetQuestionErrorMessage(LangIDs.en,"Invalid date. Please correct using the format
MM/DD YYYY");
 }
else
 {
  var dt = new Date();
  dt.setFullYear(d.year,d.month-1,d.day);
  var current = new Date();
  if(dt.valueOf() < current.valueOf())
   {
    RaiseError();
    SetQuestionErrorMessage(LangIDs.en,"Please enter a date after the current date.");
   }
  else if(dt.getDay() != 1)
   {
    RaiseError();
    SetQuestionErrorMessage(LangIDs.en,f("date4")+" is not a Monday. Please register
for Mondays only.");
   }
 }
```

Exercise 8:

```
var num : String = f("phone").get();
var re = /^\d{3} \d{3} \d{4}$/;
```

```
if(!re.test(num)
 {
  RaiseError();
  SetQuestionErrorMessage(LangIDs.en,"Please provide your phone number on the format
xxx xxx xxxx");
 }
```

Exercise 9:

```
var zipcode : String = f("zipcode").get();
var re = /^\d{5}$/;
if(zipcode.search(re) == -1)
 {
  RaiseError();
  SetQuestionErrorMessage(LangIDs.en,"Please provide a valid zip code (digits
only).");
 }
```

APPENDIX B: FURTHER READING

For further reading on JScript .NET, we recommend:

MSDN (Microsoft):
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/jscript7/html/jsoriprogrammingwithjscriptnet.asp

APPENDIX C: CONFIRMIT LANGUAGE CODES

Language code	Language	Sub name	Combident
54	Afrikaans		af
28	Albanian		sq
1	Arabic		ar
1025	Arabic	Saudi Arabia	ar_sa
2049	Arabic	Iraq	ar_iq
3073	Arabic	Egypt	ar_eg
4097	Arabic	Libya	ar_li
5121	Arabic	Algeria	ar_al
6145	Arabic	Morocco	ar_mo
7169	Arabic	Tunisia	ar_tu
8193	Arabic	Oman	ar_om
9217	Arabic	Yemen	ar_ye
10241	Arabic	Syria	ar_sy
11265	Arabic	Jordan	ar_jo
12289	Arabic	Lebanon	ar_le
13313	Arabic	Kuwait	ar_ku
14337	Arabic	U.A.E.	ar_ua
15361	Arabic	Bahrain	ar_ba
16385	Arabic	Qatar	ar_qa
43	Armenian		hy
77	Assamese		as
44	Azeri		az
1068	Azeri	Latin	az_la
2092	Azeri	Cyrillic	az_cy
45	Basque		eu
35	Belarusian		be
69	Bengali		bn

2	Bulgarian		bg
3	Catalan		ca
4	Chinese		zh
1028	Chinese	Taiwan	zh_ta
2052	Chinese	PRC	zh_pr
3076	Chinese	Hong Kong SAR, PRC	zh_hk
4100	Chinese	Singapore	zh_si
5124	Chinese	Macau SAR	zh_ma
5	Czech		cs
6	Danish		da
19	Dutch		nl
1043	Dutch	Netherlands	nl_nl
2067	Dutch	Belgium	nl_be
9	English		en
1033	English	United States	en_us
2057	English	United Kingdom	en_uk
3081	English	Australia	en_au
4105	English	Canada	en_ca
5129	English	New Zealand	en_nz
6153	English	Ireland	en_ir
7177	English	South Africa	en_sa
8201	English	Jamaica	en_ja
9225	English	Caribbean	en_ca
10249	English	Belize	en_be
11273	English	Trinidad	en_tr
12297	English	Zimbabwe	en_zi
13321	English	Philippines	en_ph
37	Estonian		et
56	Faeroese		fo
41	Farsi		fa

11	Finnish		fi
12	French		fr
1036	French	Standard	fr_fr
2060	French	Belgium	fr_be
3084	French	Canada	fr_ca
4108	French	Switzerland	fr_sw
5132	French	Luxembourg	fr_lu
6156	French	Monaco	fr_mo
55	Georgian		ka
7	German		de
1031	German	Standard	de_de
2055	German	Switzerland	de_sw
3079	German	Austria	de_au
4103	German	Luxembourg	de_lu
5127	German	Liechtenstein	de_li
8	Greek		el
71	Gujarati		gu
13	Hebrew		he
57	Hindi		hi
14	Hungarian		hu
15	Icelandic		is
33	Indonesian		id
16	Italian		it
1040	Italian	Standard	it_it
2064	Italian	Switzerland	it_sw
17	Japanese		ja
75	Kannada		kn
96	Kashmiri		ks
2144	Kashmiri	India	ks_in
63	Kazak		kk

87	Konkani		ki
18	Korean		ko
1042	Korean	Korea	ko_ko
2066	Korean	Johab	ko_jo
38	Latvian		lv
39	Lithuanian		lt
1063	Lithuanian	Lithuania	lt_lt
2087	Lithuanian	Classic	lt_cl
47	Macedonian		mk
62	Malay		ms
1086	Malay	Malaysian	ms_ms
2110	Malay	Brunei Darussalam	ms_br
76	Malayalam		ml
88	Manipuri		ma
78	Marathi		mr
97	Nepali		ne
2145	Nepali	India	ne_in
20	Norwegian		no
1044	Norwegian	Bokmål	no_bo
2068	Norwegian	Nynorsk	no_ny
72	Oriya		or
21	Polish		pl
22	Portuguese		pt
1046	Portuguese	Brazil	pt_br
2070	Portuguese	Standard	pt_st
70	Punjabi		pa
24	Romanian		ro
25	Russian		ru
79	Sanskrit		sa
26	Serbian / Croatian		sr

1050	Serbian / Croatian	Croatian	sr_yu
2074	Serbian / Croatian	Latin	sr_la
3098	Serbian / Croatian	Cyrillic	sr_cy
89	Sindhi		sd
27	Slovak		sk
36	Slovenian		sl
10	Spanish		es
1034	Spanish	Traditional Sort	es_es
2058	Spanish	Mexican	es_me
3082	Spanish	Modern Sort	es_ms
4106	Spanish	Guatemala	es_gu
5130	Spanish	Costa Rica	es_cr
6154	Spanish	Panama	es_pa
7178	Spanish	Dominican Republic	es_dr
8202	Spanish	Venezuela	es_ve
9226	Spanish	Colombia	es_co
10250	Spanish	Peru	es_pe
11274	Spanish	Argentina	es_ar
12298	Spanish	Ecuador	es_eq
13322	Spanish	Chile	es_ch
14346	Spanish	Uruguay	es_ur
15370	Spanish	Paraguay	es_pa
16394	Spanish	Bolivia	es_bo
17418	Spanish	El Salvador	es_el
18442	Spanish	Honduras	es_ho
19466	Spanish	Nicaragua	es_ni
20490	Spanish	Puerto Rico	es_pr
65	Swahili		sw
29	Swedish		sv
1053	Swedish	Sweden	sv_sv

2077	Swedish	Finland	sv_fi
73	Tamil		ta
68	Tatar		tt
74	Telugu		te
30	Thai		th
31	Turkish		tr
34	Ukrainian		uk
32	Urdu		ur
1056	Urdu	Pakistan	ur_pa
2080	Urdu	India	ur_in
67	Uzbek		uz
1091	Uzbek	Latin	uz_la
2115	Uzbek	Cyrillic	uz_cy
42	Vietnamese		vi
512	Welsh		cy

APPENDIX D: CODEPAGE

Arabic ASMO-708	708
Arabic (DOS)	720
Arabic (ISO)	28596
Arabic (Windows)	1256
Baltic (ISO)	28594
Baltic (Windows)	1257
Central European (DOS)	852
Central European (ISO)	28592
Central European (Windows)	1250
Chinese Simplified (GB2312)	936
Chinese Simplified (HZ)	52936
Chinese Traditional	950
Cyrillic (DOS)	866
Cyrillic (ISO)	28595
Cyrillic (KOI8-R)	20866
Cyrillic (Windows)	1251
Greek (ISO)	28597
Greek (Windows)	1253
Hebrew (DOS)	862
Hebrew (ISO)	28598
Hebrew (Windows)	1255
Japanese (JIS)	50220
Japanese (JIS-Allow 1-byte Kana)	50221
Japanese (JIS-Allow 1-byte Kana - SO/SI)	50222
Japanese (EUC)	51932

Japanese (Shift-JIS)	932
Korean	949
Korean (ISO)	50225
Latin 3 (ISO)	28593
Thai (Windows)	874
Turkish (Windows)	1254
Turkish (ISO)	28599
Ukrainian (KOI8-U)	21866
Unicode (UTF-7)	65000
Unicode (UTF-8)	65001
Vietnamese (Windows)	1258
Western European (Windows)	1252
Western European (ISO)	1252

EXAMPLES

Example 1: Screening Based on a Single Question

Example 2: Filtering a Single Question Based on Answers to a Multi

Example 3: Excluding a Column (Question) in a 3D grid

Example 4: Piping in the Response to a Single Question

Example 5: Password Check

Example 6: Setting complete status before the end of the survey

Example 7: Removing an Answer in a Single or Grid Question

Example 8: Screening on a Numeric Question

Example 9: Checking that a Multi Question has been Answered

Example 10: Response Piping from a Single Question with Other Specify

Example 11: Replacing "NO RESPONSE" in Response Piping

Example 12: Using switch to set Values for each of the Answer Alternatives on a Single

Example 13: Validating Sums in a 3D Grid Using a while Loop

Example 14: Validating Sums in a 3D Grid Using a do while Loop

Example 15: Copying a Multi to do Response Piping with "Other, specify"

Example 16: Validating Sums in a 3D Grid Using a for Loop and break

Example 17: Calculating Averages in a Grid

Example 18: Calculating Averages on a Single Question in a Loop

Example 19: Validation of Ranking when the Number of Elements is Optional

Example 20: Validating Sum on a Multi Numeric Question

Example 21: Validating Sums in a 3D Grid with the Sum Function

Example 22: Finding the Number of Answers on Three Multi Questions

Example 23: Calculating Averages on 3 Numeric Multi Questions in a 3D Grid

Example 24: Finding Maximum and Minimum Values on Numeric Multi Questions

Example 25: Building a Condition on a Range of Precodes

Example 26: Making a Multi Question Required (Generic Code)

Example 27: Building a Cryptic URL to be displayed in an info node

Example 28: Setting Interview Status Before End of Survey

Example 29: Recording the Respondent's Browser Type and Version

Example 30: Quota Check

Example 31: Presetting a Quota Question to Check Several Quotas

Example 32: Checking that a Response in a Multi Open Text Question is an Integer

Example 33: Validating Date Format of Open Text Date Question

Example 34: Validating Date with Dropdowns for the Date Parts

Example 35: Validation of Email Address Format

Example 36: Excluding Respondents from Specific Networks

Example 37: Send Confirmation Email at the End of a Survey

Example 38: Invitation Email to a Different Part of the Same Survey

Example 39: Redirect to Another Site Before the End Page

Example 40: Function to Make a Multi Question Required

Example 41: Function to Copy a Multi Question

Example 42: Returning a Calculated Value from a Function

Example 43: Referencing a Question in a Loop

Example 44: Using a Variable instead of Repeated Calls on the f Function

Example 45: Deleting the Content of any Question

Example 46: Copying the Contents of any Form into Another

Example 47: Filtering an Answer List by the First Characters in the Answer

Example 48: Checking the Number of Answers on a Multi Question

Example 49: Filtering an Answer List on Items Selected in Two Previous Questions

Example 50: Filtering Answers Not Selected in a Previous Question

Example 51: Always Including a "Don't know" Answer Alternative

Example 52: Joining Answers in one Multi Question

Example 53: Using a Function to Filter an Answer List Based on the Answers on a Grid

Example 54: Using a Hidden Multi to Filter an Answer List Based on a Grid

Example 55: Calculating Time Spent

Example 56: Validating Date Format and that it is a Valid Date after Current Date

Example 57: Validating that a Date with Dropdowns is within the next two Weeks

Example 58: Finding the Weekday

Example 59: Converting Data of Birth into Age (in number of years)

Example 60: Rounding to a Number with Two Digits

Example 61: Picking n Random Items from the Answers to a Multi Question

Example 62: Randomly Assigning which Part of a Survey the Respondents Should Answer

Example 63: Checking a User Name and Password where Username is Case Insensitive

Example 64: On the Fly Recoding

Example 65: Replacing Last Comma with "and" in a Listing of Answers

Example 66: Generating an Array from a String with Values

Example 67: Validation of the Format of a Phone Number

Example 68: Using Regular Expression to Replace Commas With Line Breaks

Example 69: Postal Codes in the United Kingdom

Example 70: Converting an Array to a String with Line Breaks between Elements

Example 71: Validating Grid with "Other, specify" Alternatives

Example 72: Sending in Values with the URL

Example 73: Using Cookies to Limit Access to an Open Survey

INDEX

—#—

- (operator) 33
/* */ (comment) 20
// (comment) 20
\' (string formatting character) 27
\" (string formatting character) 27
\\ (string formatting character) 27
\n (string formatting character) 27
\r (string formatting character) 27
\t (string formatting character) 27
\b (string formatting character) 27
+ *(operator)* 32
* (operator) 33
/ (operator) 33
% (operator) 33
++ (operator) 33
-- (operator) 33
&& (operator) 33
|| (operator) 33
! (operator) 33
== (operator) 34
=== (operator) 34
!= (operator) 34
!== (operator) 34
< (operator) 34
<= (operator) 34
> (operator) 34
>= (operator) 34
+ (operator) 34
= (operator) 34
+= (operator) 35
-= (operator) 35
*= (operator) 35
/= (operator) 35
%= (operator) 35
? : (operator) 35
; (semi colon) 40
{ } (curly brackets) 40
{ } (curly brackets) 42
$ *(regular expression syntax)* 142
() *(regular expression syntax)* 142
* (regular expression syntax) 142
+ (regular expression syntax) 142
. (regular expression syntax) 142
[] (regular expression syntax) 142
? (regular expression syntax) 142
\ (regular expression syntax) 142
^ (regular expression syntax) 142
{} (regular expression syntax) 142
| (regular expression syntax) 142
\$ *(regular expression syntax)* 142
\(*(regular expression syntax)* 142
\) *(regular expression syntax)* 142
* *(regular expression syntax)* 143
\+ *(regular expression syntax)* 143
\. *(regular expression syntax)* 143
\[*(regular expression syntax)* 143
\? *(regular expression syntax)* 143
\\ *(regular expression syntax)* 143
\^ *(regular expression syntax)* 143
\{ *(regular expression syntax)* 143
\| *(regular expression syntax)* 143
\c (regular expression syntax) 143
\f (regular expression syntax) 143
\n (regular expression syntax) 143
\r (regular expression syntax) 143
\s (regular expression syntax) 143
\S (regular expression syntax) 143
\t (regular expression syntax) 143
\v *(regular expression syntax)* 143
[] (regular expression syntax) 144
- (regular expression syntax) 144
\- (regular expression syntax) 144
^ (regular expression syntax) 144
* (regular expression syntax) 145
+ (regular expression syntax) 145
? (regular expression syntax) 145
{} (regular expression syntax) 145
^ (regular expression syntax) 146

$ *(regular expression syntax)* 146
\b (regular expression syntax) 146
\B (regular expression syntax) 146
| (regular expression syntax) 146
() (regular expression syntax) 147
?: (regular expression syntax) 147
?= (regular expression syntax) 147
?! (regular expression syntax) 147
\ (regular expression syntax) 148
\d (regular expression syntax) 149
\D (regular expression syntax) 149
\w (regular expression syntax) 149
\W (regular expression syntax) 149
\x (regular expression syntax) 149
\ (regular expression syntax) 149
\u (regular expression syntax) 149
{ } (curly brackets) 174
; (semi colon) 175

—A—

a (function) 105
abs (method) 133
acos (method) 129
add (method) 113
Add (method) 168
addition *(operator)* 32
anchor *(method)* 140
and (operator) 33
AppendErrorMessage (function) 17
AppendQuestionErrorMessage (function) 17
argument 71, 92
  parameter array 95
array
  literal 45
array 45–47
  declare 45
array
  declare 46
Array (object) 153–57
  concat (method) 153
  join (method) 153
  length (property) 47
  methods 153–57
  pop (method) 154
  push (method) 154
  reverse (method) 154
  shift (method) 154
  slice (method) 157
  sort (method) 10, 155
  splice (method) 157
  toString (method) 153
  unshift (method) 154
  valueOf (method) 153
asin (method) 129
assignment (statement) 40
atan (method) 129
atan2 (method) 129
`Average (function)` 74

—B—

big *(method)* 140
binary operator 32
blink *(method)* 140
block 40
bold *(method)* 140
Boolean 26
  false 26
  true 26
break (statement) 42, 66
`BrowserType (function)` 80
`BrowserVersion (function)` 80
byte 25

—C—

case .... See switch (statement)
categories (method) 49
categoryLabels (method) 49
ceil (method) 129
char 26
charAt (method) 135
charCodeAt (method) 135
ClearErrorMessage (function) 17
ClearQuestionErrorMessage (function) 17
CODED (property) 102

coercion 37
column masks 14
combident 18, 181
comments 20
compound 100
COMPOUND (property) 102
concat (method) 139, 153
conditional expression ternary operator 35
conditions 12
const 21, 40
constant 21
  name 21
  scope 21
constructor 99
continue (statement) 67
conversion 27
  explicit 27
  implicit 28
  narrowing 27
  widening 27
Cookies (property) 166, 168
cos (method) 129
Count (function) 74
curly brackets 40, 42
`CurrentForm (function)` 76
`CurrentID (function)` 77
`CurrentLang (function)` 77
`CurrentPID (function)` 77
`CurrentSID (function)` 77

—D—

data declaration 21
data type 21, 24
  array 45–47
  Array 153–57
  Boolean 26
  byte 25
  char 26
  coercion 37
  conversion 27
  decimal 26
  double 26
  float 25
  int25
  long 25
  Number 26
  object 98–157
  sbyte 25
  short 25
  string 26
  uint 25
  ulong 25
  ushort 25
Date (object) 115–28
  Constructors 115
  getDate (method) 120
  getDay (method) 120
  getFullYear (method) 119
  getHours (method) 120
  getMilliseconds (method) 122
  getMinutes (method) 121
  getMonth (method) 119
  getSeconds (method) 121
  getTime (method) 122
  getTimezoneOffset (method) 122
  getUTCDate (method) 120
  getUTCDay (method) 120
  getUTCFullYear (method) 119
  getUTCHours (method) 120
  getUTCMilliseconds (method) 122
  getUTCMinutes (method) 121
  getUTCMonth (method) 119
  getUTCSeconds (method) 121
  getYear (method) 119
  methods 116–28
  parse (method) 116
  setDate (method) 120
  setFullYear (method) 119
  setHours (method) 120
  setMilliseconds (method) 122
  setMinutes (method) 121
  setMonth (method) 119
  setSeconds (method) 121
  setTime (method) 122
  setUTCDate (method) 120

setUTCFullYear (method) 119
setUTCHours (method) 120
setUTCMilliseconds (method) 122
setUTCMinutes (method) 121
setUTCMonth (method) 119
setUTCSeconds (method) 121
setYear (method) 119
toGMTString (method) 123
toLocaleString (method) 123
toString (method) 123
toUTCString (method) 123
UTC (method) 117
valueOf (method) 122
DateType (object) 169
day (property) 124
decimal 26
decimal numbers 24
declaration (statement) 40
decrement (operator) 33
DEF_EXCL (error template) 161
DEF_EXCL_AND (error template) 161
DEF_EXCL_OR (error template) 161
DEF_NEXCL (error template) 161
DEF_NEXCL_AND (error template) 161
DEF_NEXCL_OR (error template) 161
DICHOTOMY (property) 102
diff (method) 108
division (operator) 33
do while (statement) 62
domainLabels (method) 49
domainValues (method) 49
double 26

—E—

E (property) 128
ECMAScript 8
else .... See if (statement)
equal (operator) 34
ErrorTemplate (function) 159–64
DEF_EXCL 161
DEF_EXCL_AND 161
DEF_EXCL_OR 161
DEF_NEXCL 161
DEF_NEXCL_AND 161
DEF_NEXCL_OR 161
MAX_SIZE 162
MISSING 160
MISSING_AND 160
MISSING_OR 160
NUMERIC_ERRORS 163
NUMERIC_ERRORS_AND 163
OTHER 161
PRECISION 163
PRECISION_ERRORS 163
PRECISION_ERRORS_AND 163
RANGE_ERRORS 164
RANGE_ERRORS_AND 164
RANGE_MAX 164
RANGE_MIN 164
RANK_MAX 162
RANK_MIN 162
SCALE 164
SCALE_ERRORS 164
SCALE_ERRORS_AND 164
SEL_EXCL 160
SEL_EXCL_AND 160
SEL_EXCL_OR 161
SEL_NEXCL 161
SEL_NEXCL_AND 161
SEL_NEXCL_OR 161
SPEC 161
TOO_LONG 162
TOO_LONG_AND 162
ExclusivityError (function) 159, 160
exec (method) 150
exp (method) 133
Expires (property) 169
explicit conversion 27
expression 32

—F—

f (function) 48–58, 100–104, 106
categories (method) 49
categoryLabels (method) 49

CODED (property) 102
COMPOUND (property)................................ 102
compund.. 100
DICHOTOMY (property) 102
diff (method) 108
domainLabels (method).................................... 49
domainValues (method).................................... 49
function call.. 100
get (method) 48
inc (method)... 106
isect (method)... 107
label (method)... 48
members (method) 111
methods...........................29, 48–58, 103, 106–12
NUMERIC (property) 102
OPEN (property)... 102
properties... 102
set (method) 48
size *(method)*... 107
toBoolean (method) 30
toNumber (method)... 29
union (method) 107
value (method).. 48
valueLabel (method)... 48
values (method).. 49
f (function)... 106–12
false... 26
Filter (function)... 106
fixed *(method)*.. 140
float.. 25
floating-point data 25
floor (method) 130
fontcolor *(method)* 140
fontsize *(method)*.. 140
for (statement) 63
Form (property).. 165
form objects ....See f (function)
formatting characters (strings) 27
`Forward (function)`.. 80
fromCharCode (method).................................... 135
function 71
a 105
AppendErrorMessage...................................... 17
AppendQuestionErrorMessage 17
argument 71, 92
parameter array.. 95
`Average` 74
`BrowserType`.. 80
`BrowserVersion` 80
call 71, 92
ClearErrorMessage 17
ClearQuestionErrorMessage 17
Count.. 74
`CurrentForm`.. 76
`CurrentID` 77
`CurrentLang`.. 77
`CurrentPID` 77
`CurrentSID` 77
definition 92
ErrorTemplate 159–64
ExclusivityError.......................................159, 160
f 48–58, 100–104, 106–12
Filter 106
`Forward` 80
`GetRespondentValue`................................. 78
`GetStatus` 79
global variable 96
`InRange` 75
`InRangeExcl` 75
InterviewEnd..78, 123
InterviewStart 78, 123
`IsDate` 84, 124
IsDateFmt...83, 124
`IsEmail` 87
`isEmailTaken`.. 83
`IsInRdgMode`.. 80
`IsInteger` 83
IsNet 87
`IsNumeric` 83
`isUserNameTaken` 82
local variable 96
`Max` 75
`Min` 75
MissingRequiredError.............................159, 160

naming conventions ... 174
nnset ... 105
NotSelectedError ... 159, 161
NotSpecifiedError ... 161
nset ... 105
NumericError ... 159, 163
parseFloat ... 28
PrecisionError ... 159, 163
qf ... 81
QuestionErrors ... 159
RaiseError ... 16
RangeError ... 159, 164
RankError ... 159, 162
Redirect ... 91
RequestIP ... 81
return (statement) ... 96
ScaleError ... 159, 164
SendMail ... 88
set ... 105
SetErrorMessage ... 17
SetInterviewEnd ... 78
SetInterviewStart ... 78
SetQuestionErrorMessage ... 17
SetRespondentValue ... 78
SetStatus ... 79
SizeError ... 162
Sum ... 71
function (Statement) ... 92

—G—

get (method) ... 48
getDate (method) ... 120
getDay (method) ... 120
getFullYear (method) ... 119
getHours (method) ... 120
getMilliseconds (method) ... 122
getMinutes (method) ... 121
getMonth (method) ... 119
GetRespondentValue (function) ... 78
getSeconds (method) ... 121
GetStatus (function) ... 79
getTime(method) ... 122
getTimezoneOffset(method) ... 122
getUTCDate (method) ... 120
getUTCDay (method) ... 120
getUTCFullYear (method) ... 119
getUTCHours (method) ... 120
getUTCMilliseconds (method) ... 122
getUTCMinutes (method) ... 121
getUTCMonth (method) ... 119
getUTCSeconds (method) ... 121
getYear (method) ... 119
global variable ... 96
greater than (operator) ... 34
greater than or equal (operator) ... 34

—H—

hexadecimal numbers ... 24
hidden question ... 21
HttpCookie (object)
Expires (property) ... 169
Name (property) ... 169
Path (property) ... 169
Value (property) ... 169
Values (property) ... 169
HttpCookiesCollection (object)
Add (method) ... 168
HttpCookiesCollection (object) ... 168

—I—

if (statement) ... 41
implicit conversion ... 28
inc (method) ... 106
increment (operator) ... 33
indexOf (method) ... 136
infinity ... 25
InRange (function) ... 75
InRangeExcl (function) ... 75
int25
integers ... 24
InterviewEnd (function) ... 78, 123
InterviewStart (function) ... 78, 123
IsDate (function) ... 84, 124
day (property) ... 124

month (property) 124
year (property) 124
IsDateFmt (function) 83, 124
day (property) 124
month (property) 124
year (property) 124
isect (method)... 107
`IsEmail (function)` 87
`isEmailTaken (function)` 83
isFinite (method).. 29
`IsInRdgMode (function)` 80
`IsInteger (function)` 83
isNaN (method) 29
IsNet (function) 87
`IsNumeric (function)` 83
`isUserNameTaken (function)` 82
italics *(method)* 140

—J—

JavaScript... 8
join (method).. 153
JScript array 46
declare... 46

—L—

label (method).. 48
label (statement)... 67
langIDs 18, 181
language
combident 18, 181
lastIndexOf (method) 137
length (property) 47, 134
less than (operator) 34
less than or equal (operator)................................ 34
link *(method)*.. 140
LiveScript... 8
LN10 (property) 128
LN2 (property) 128
local variable.. 96
log (method) 133
LOG10E *(property)*.. 128
LOG2E (property)...128
long 25
loops...59–70

—M—

match (method) 151
Math (object) 128–33
abs (method) 133
acos (method) 129
asin (method) 129
atan (method) 129
atan2 (method) 129
ceil (method)..129
cos (method) 129
E (property) 128
exp (method) 133
floor (method) 130
LN10 (property) 128
LN2 (property) 128
log (method) 133
LOG10E *(property)*..128
LOG2E (property) 128
max (method) 133
methods..129–33
min (method) 133
PI *(property)* 128
pow (method) 133
properties 128
random (method)...130
round (method)..129
sin (method) 129
sqrt (method) 133
SQRT1_2 *(property)*...128
SQRT2 (property)..129
tan (method) 129
`Max (function)` 75
max (method) 133
MAX_SIZE (error template)................................162
members (method) 111
method 98
parseInt 28
`Min (function)` 75

min (method) 133
MISSING (error template) 160
MISSING_AND (error template) 160
MISSING_OR (error template) 160
MissingRequiredError (function) 159, 160
modulus (operator) 33
month (property) 124
multiplication (operator) 33

—N—

Name (property) 169
NameValueCollection (object) 165, 169
naming conventions 174
NaN (not a number) 25, 26, 28, 29
narrowing conversion 27
negative 0 25
negative infinity 25
new (operator) 35, 46, 98
nnset *(function)* 105
not (operator) 33
not equal (operator) 34
NotSelectedError (function) 159, 161
NotSpecifiedError (function) 161
nset (function) 105
null 22
Number 26
NUMERIC (property) 102
numeric data 24
NUMERIC_ERRORS (error template) 163
NUMERIC_ERRORS_AND (error template) 163
NumericError (function) 159, 163

—O—

object 98–157
  Array 45–47, 153–57
  constructor 99
  Date 115–28
  form objects .... See f (function)
  instance 98
  Math 128–33
  method 98
  property 98
  RegExp 140–52
  Set 105–14
  String 134–40, 151–52
octal numbers 24
OPEN (property) 102
operand 32
operator 32–39
  arithmetic 32
    addition 32
    decrement 33
    division 33
    increment 33
    modulus 33
    multiplication 33
    subtraction 33
  assignment 34
    %= (operator) 35
    *= (operator) 35
    /= (operator) 35
    += (operator) 35
    = (operator) 34
    -= (operator) 35
  binary 32
  comparision 34
    equal 34
    greater than 34
    greater than or equal 34
    less than 34
    less than or equal 34
    not equal 34
    strictly equal 34
    strictly no equal 34
  conditional expression ternary 35
  logical 33
    and 33

not 33
or 33
new 35, 46, 98
precedence 38
string 34
string concatenation 34
unary 32
or (operator) 33
OTHER (error template) 161

—P—

parameter .... See argument
parameter array 95
parse (method) 116
parseFloat (function) 28
parseFloat (method) 29
parseInt (method) 28, 29
Path (property) 169
PI *(property)* 128
pop (method) 154
positive 0 25
positive infinity 25
pow (method) 133
PRECISION (error template) 163
PRECISION_ERRORS (error template) 163
PRECISION_ERRORS_AND (error template) ... 163
PrecisionError (function) 159, 163
precode masks 13, 105–14
property 98
push (method) 154

—Q—

qf (function) 81
QueryString (property) 165
question id 21
QuestionErrors (function) 159

—R—

RaiseError (function) 16
random (method) 130
RANGE_ERRORS (error template) 164
RANGE_ERRORS_AND (error template) 164
RANGE_MAX (error template) 164
RANGE_MIN (error template) 164
RangeError (function) 159, 164
RANK_MAX (error template) 162
RANK_MIN (error template) 162
RankError (function) 159, 162
Redirect (function) 91
RegExp (object) 140–52
Constructors 141, 150
exec (method) 150
methods 150–51
syntax 141–50
test (method) 150
regular expressions .... See RegExp (object)
remove (method) 113
replace (method) 151
Request (object)
Cookies (property) 166, 168
Form (property) 165
QueryString (property) 165
ServerVariables (property) 166
RequestIP (function) 81
Response (object)
Write (method) 168
Response (object) 168
response piping 15
return (statement) 96
reverse (method) 154
round (method) 129

—S—

sbyte 25
SCALE (error template) 164
scale masks 13, 105–14
SCALE_ERRORS *(error template)* 164
SCALE_ERRORS_AND (error template) 164
ScaleError (function) 159, 164
script nodes 18
search (method) 151
SEL_EXCL (error template) 160
SEL_EXCL_AND (error template) 160

SEL_EXCL_OR (error template) 161
SEL_NEXCL (error template) 161
SEL_NEXCL_AND (error template).................... 161
SEL_NEXCL_OR (error template)...................... 161
SendMail (function).................................... 88
ServerVariables (property)................................ 166
set... 23
set (function)... 105
set (method) 29, 48
Set *(object)*
  add (method) 113
  diff (method) 108
  isect (method).. 107
  members (method) 111
  remove (method) 113
  size *(method)*.. 107
  union (method) 107
Set (object) 105–14
  a (function).. 105
  Constructor 105
  Filter (function)... 106
  inc (method).. 106
  methods.. 106–13
  nnset *(function)*.. 105
  nset (function).. 105
  set (function).. 105
setDate (method)... 120
SetErrorMessage (function)................................. 17
setFullYear (method) 119
setHours (method)... 120
SetInterviewEnd (function)................................... 78
SetInterviewStart (function) 78
setMilliseconds (method).................................. 122
setMinutes (method).. 121
setMonth (method) 119
SetQuestionErrorMessage (function) 17
SetRespondentValue (function).............. 78
setSeconds (method) 121
SetStatus (function).................................. 79
setTime(method) 122
setUTCDate (method) 120
setUTCFullYear (method).................................. 119
setUTCHours (method) 120
setUTCMilliseconds (method) 122
setUTCMinutes (method) 121
setUTCMonth (method) 119
setUTCSeconds (method) 121
setYear (method)..119
shift (method) 154
short 25
short circuit evaluation... 39
sin (method) 129
size *(method)*...107
SizeError (function) 162
slice (method)...................................137, 157
small *(method)*..140
sort (method) 10, 155
SPEC (error template)...161
splice (method)..157
split (method)...139
sqrt (method) 133
SQRT1_2 *(property)*...128
SQRT2 (property)...129
statement.. 40
  assignment 40
  break 42, 66
  continue 67
  declaration 40
  declare array 45, 46
  do while 62
  for 63
  function 92
  function call 71, 92
  if 41
  label 67
  loops 59–70
  object instantiation.. 98
  return 96
  switch 42
  while 59
strictly equal (operator).. 34
strictly not equal (operator).................................. 34
strike *(method)* 140
string.. 26
  formatting characters...................................... 27
String (object) 134–40, 151–52

anchor *(method)* 140
big *(method)* 140
blink *(method)* 140
bold *(method)* 140
charAt (method) 135
charCodeAt (method) 135
concat (method) 139
constructors 134
fixed *(method)* 140
fontcolor *(method)* 140
fontsize *(method)* 140
fromCharCode (method) 135
index 134
indexOf (method) 136
italics *(method)* 140
lastIndexOf (method) 137
length (property) 134
link *(method)* 140
match (method) 151
methods 134–40, 151–52
properties 134
replace (method) 151
search (method) 151
slice (method) 137
small *(method)* 140
split (method) 139
strike *(method)* 140
sub *(method)* 140
substr (method) 138
substring (method) 138
sup *(method)* 140
toLowerCase (method) 136
toString (method) 139
toUpperCase (method) 136
valueOf (method) 139
string concatenation (operator) 34
sub *(method)* 140
substr (method) 138
substring (method) 138
subtraction (operator) 33
Sum *(function)* 71
sup *(method)* 140
switch (statement) 42

—T—

tan (method) 129
test (method) 150
text substitution 15
toBoolean 23
toBoolean (method) 30
toGMTString (method) 123
toLocaleString (method) 123
toLowerCase (method) 136
toNumber (method) 29
TOO_LONG (error template) 162
TOO_LONG_AND (error template) 162
toString (method) 29, 123, 139, 153
toUpperCase (method) 136
toUTCString (method) 123
true 26
typed array 45
declare 45

—U—

uint 25
ulong 25
unary operator 32
undefined 22
union (method) 107
unshift (method) 154
ushort 25
UTC (method) 117
UTC (Universal Coordinated Time) 115

—V—

validation code 16, 158–64
value (method) 48
Value (property) 169
valueLabel (method) 48
valueOf (method) 29, 122, 139, 153
values (method) 49
Values (property) 169
var 21, 40
variable 21
global 96

local 96
name..... 21
naming conventions..... 174
scope..... 21

—W—

while (statement) 59
widening conversion..... 27
Write (method)..... 168

—Y—

year (property)..... 124