Scripting Manual

Using JScript in Server-Side Scripting in Confirmit

Version 1.1
December 2002

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

The manual is developed based on JScript and Confirmit as of December 1st, 2002. 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 **9**

1.1 CONDITIONS 9

1.2 FILTERING ANSWER LISTS, SCALES AND LOOPS 11

1.2.1 Precode Masks and Scale Masks *11*

1.2.2 Column Masks *12*

1.3 TEXT SUBSTITUTION/RESPONSE PIPING 13

1.4 VALIDATION CODE 14

1.5 SCRIPT NODES 17

2 Comments **19**

3 Types and Variables **20**

3.1 NAMING 20

3.2 DATA DECLARATION 20

3.3 TYPES 20

3.3.1 Number *21*

3.3.2 Boolean *21*

3.3.3 String *22*

3.3.4 null *23*

3.3.5 undefined *25*

3.4 CONVERSION BETWEEN TYPES/CONVERSION FUNCTIONS 25

3.4.1 Conversion Functions in JScript *26*

3.4.2 Conversions Methods in Confirmit *26*

4 Operators and Expressions **29**

4.1 TERMINOLOGY 29

4.2 ARITHMETIC OPERATORS 30

4.3 LOGICAL OPERATORS 31

4.4 COMPARISON OPERATORS 31

4.5 STRING OPERATORS 31

4.6 ASSIGNMENT OPERATORS 32

4.7 THE CONDITIONAL EXPRESSION TERNARY OPERATOR 33

4.8 OPERATOR PRECEDENCE 34

5 Simple Statements 36

5.1 DATA DECLARATIONS 36

5.2 ASSIGNMENT STATEMENTS 36

5.3 THE IF STATEMENT 36

5.3.1 if 36

5.3.2 if-else 37

5.3.3 Using Curly Brackets in if Statements 38

5.4 THE SWITCH STATEMENT 38

6 Arrays 41

6.1 DECLARING ARRAYS 41

6.2 THE LENGTH PROPERTY 42

7 Methods of the Form Objects 44

7.1 GET AND SET 44

7.2 LABEL 44

7.3 VALUE 44

7.4 VALUELABEL 44

7.5 DOMAINVALUES 45

7.6 DOMAINLABELS 45

7.7 CATEGORIES 45

7.8 CATEGORYLABELS 45

7.9 VALUES 45

7.10 APPLYING THE METHODS ON DIFFERENT TYPES OF QUESTIONS 45

7.10.1 Open Text Question 46

7.10.2 Single Question 47

7.10.3 Multi Question 48

7.10.4 Grid Question 49

7.10.5 Referencing the Elements of a Multi or a Grid 50

7.10.6 Loops 51

7.10.7 3D Grid......52

7.10.8 Implicit Conversion of Arrays to Strings......52

7.11 OVERVIEW – METHODS OF BASIC VARIABLE OBJECTS IN CONFIRMIT 54

8 Loop Statements......55

8.1 THE WHILE STATEMENT......55

8.2 THE DO WHILE STATEMENT 59

8.3 THE FOR STATEMENT......60

8.4 LOOP NODES IN CONFIRMIT......63

8.5 THE BREAK STATEMENT 64

8.6 THE CONTINUE STATEMENT......65

8.7 THE LABEL STATEMENT/NESTED LOOPS 66

9 Functions......71

9.1 BUILT-IN FUNCTIONS IN CONFIRMIT......71

9.1.1 Arithmetic Functions......71

9.1.2 Range 75

9.1.3 Context Information......76

9.1.4 Browser Information......80

9.1.5 Quota check 81

9.1.6 Panel......83

9.1.7 Classification Functions 83

9.1.8 General Utilities......88

9.2 CREATING YOUR OWN FUNCTIONS 92

9.2.1 Defining functions......92

9.2.2 Function Call 93

9.2.3 Functions with a Fixed Number of Arguments......93

9.2.4 Functions with a Variable Number of Arguments......95

9.2.5 The return Statement......95

9.2.6 Local Variables......96

10 Objects 98

10.1 OBJECT-ORIENTED PROGRAMMING......98

10.1.1 Defining and Instantiating Object Types *98*

10.1.2 Object Composition............ *99*

10.1.3 Inheritance *100*

10.1.4 Polymorphism *100*

10.2 OBJECTS IN JSCRIPT 101

10.2.1 Properties............ *101*

10.2.2 Methods............ *101*

10.2.3 Constructors: Creating Instances of Objects *101*

11 Confirmit's f Function 102

11.1 CALLING THE F FUNCTION 102

11.2 STORING THE FORM OBJECT IN A VARIABLE 102

11.3 COMPOUNDS 102

11.4 PROPERTIES 104

11.5 METHODS 107

12 Working with Sets 108

12.1 CONSTRUCTOR............ 108

12.2 FUNCTIONS RETURNING SETS............ 108

12.2.1 The a Function *108*

12.2.2 set, nset and nnset *108*

12.2.3 The Filter Function *109*

12.2.4 The f Function............ *109*

12.3 METHODS OF THE SET OBJECT............ 109

12.3.1 inc............ *109*

12.3.2 size *110*

12.3.3 union, isect and diff............ *110*

12.3.4 Combining Set Operators............ *113*

12.3.5 members............ *115*

12.4 USER-DEFINED FUNCTIONS IN PRECODE OR SCALE MASKS 117

13 Predefined JScript Objects 119

13.1 THE DATE OBJECT............ 119

13.1.1 Constructors *119*

13.1.2 Methods *120*

13.1.3 Confirmit Date Functions *128*

13.2 THE MATH OBJECT 134

13.2.1 Properties *134*

13.2.2 Methods *135*

13.3 THE STRING OBJECT 141

13.3.1 Constructors *141*

13.3.2 Properties *141*

13.3.3 Index *142*

13.3.4 Converting to String from Other Types *142*

13.3.5 Methods *142*

13.4 REGULAR EXPRESSIONS 149

13.4.1 Regular Expression Syntax *149*

13.4.2 Order of Precedence *158*

13.4.3 The Regular Expression Object *159*

13.4.4 String Object Methods that Uses Regular Expression Objects *160*

13.5 THE ARRAY OBJECT 162

13.5.1 Combining Arrays *162*

13.5.2 Converting Arrays to Strings *162*

13.5.3 Removing and Adding Elements *163*

13.5.4 Changing the Order of the Elements *163*

13.5.5 slice and splice *166*

14 Customizing Standard Error Messages 168

14.1 FUNCTIONS FOR STANDARD VALIDATION 169

14.2 TEMPLATE BASED ERROR MESSAGES 170

14.2.1 Missing Required *170*

14.2.2 Exclusivity Tests *171*

14.2.3 Other-Specify Verification *172*

14.2.4 Answer Size Tests *172*

14.2.5 Rank Order Tests *173*

14.2.6 Numeric Error Tests *173*

14.2.7 Precision Error Tests *174*

14.2.8 Scale Error Tests *174*

14.2.9 Range Error Tests *175*

15 Programming Conventions 176

15.1 COMMENTS 176

15.2 NAMING CONVENTIONS 176

15.3 SPACES AND LINE BREAKS 176

15.4 CURLY BRACKETS 177

15.5 SEMI COLON 178

15.6 THE STEP BY STEP APPROACH 178

15.7 WRITING EFFICIENT CODE 179

APPENDIX A Answers to Exercises 180

APPENDIX B Further Reading 184

APPENDIX C Confirmit Language Codes 185

APPENDIX D Codepage 191

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),
- for filtering the lists displayed in questions and the iterations in loops based on previous answers (precode 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 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 the fundamentals of JScript 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 3.1. Currently JScript is in version 5.6.

JScript is not a cut-down version of another language (it is only distantly and indirectly related to Java, for example), nor is it a simplification of anything. It is, however, limited. You cannot write stand-alone applications in it, for example, and it has no built-in support for reading or writing files. Moreover, JScript scripts can run only in the presence of an interpreter or "host", such as Active Server Pages (ASP) (as the server-side scripts in Confirmit), Internet Explorer (for client-side scripts), or Windows Script Host.

When doing server-side programming there are very few differences between JavaScript and JScript. Since the scripts are running on the server, they are browser-independent. We will not be looking at client-side scripts (scripts that run on the respondent's browser), so for example scripting of HTML elements is outside of the scope of this documentation.

JScript is an interpreted programming language, which means that the code will only be checked at run-time. So when you use scripts you should always test your questionnaires thoroughly before going "live".

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

1.1 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:

Answers

English	Precode	W
Below 18		
18-25		
26-35		
36-45		
46-55		
56-65		
Above 65		

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:

Condition Properties

Expression:
f('age') == '1'

Branches: If..Then

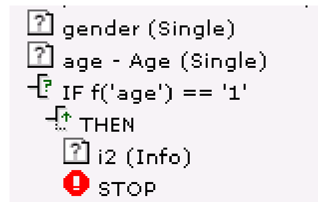
Deleted: ☐ Do not perform delete: ☐

Save | Check expression

Condition builder

Variable	Operators	Precode
age	==	Below 18 [1]
	!=	18-25 [2]
	<	26-35 [3]
	<=	36-45 [4]
	>	46-55 [5]
	>=	56-65 [6]
	And	Above 65 [7]
	Or	
	!	
	)	

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.

1.2 *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:

1. 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.

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

1.2.1 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 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 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".

Answerlist

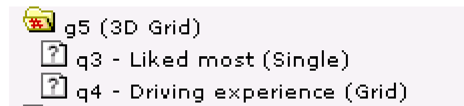
List name: Cars

Answers

English	Precode	Weight	ColWidth	BgColor	Punch	KeepPos
Acura						
Am General						
Aston Martin						
Audi						
Bentley						
BMW						
Buick						
Cadillac						

Add | Add predefined | Clear | Delete rows | Save

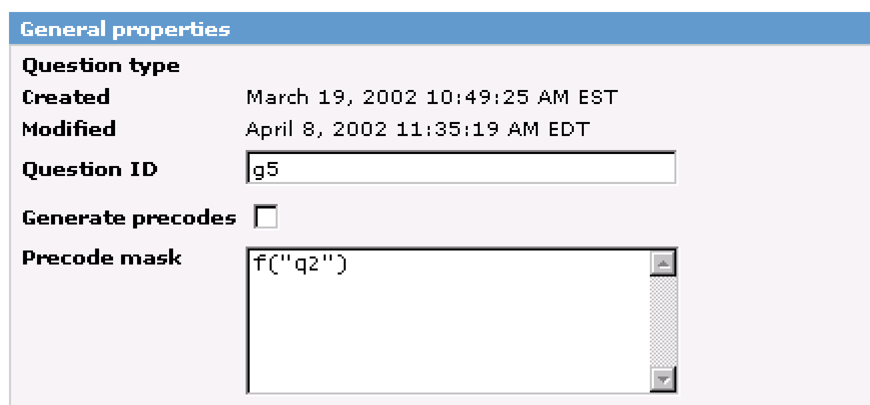
The list is used in two questions: First you have a multi question q2 which asks what cars you have tested, and then a 3D grid question g5 with a single question q3 which asks what car you did like most of the ones you tested, and a grid question q4 which asks you to rate the driving experience of the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

1.2.2 Column 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-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.

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".)

q4 -

Text | Scale | Properties | Preview | Tracking | Languages | Sa

General properties

Question type Grid

Created June 6, 2002 10:16:00 AM EDT

Modified June 6, 2002 10:16:00 AM EDT

Question ID q4

Field width

Column mask f('age') != '2'

Scale mask

1.3 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 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")^
```

q6 -

Text | Answers | Properties | Preview | Tracking | Languages | Save

English

Title

Number of times the favorite car has been tested

Text

How many times have you tested ^f("q3")^?

Instruction

1.4 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:

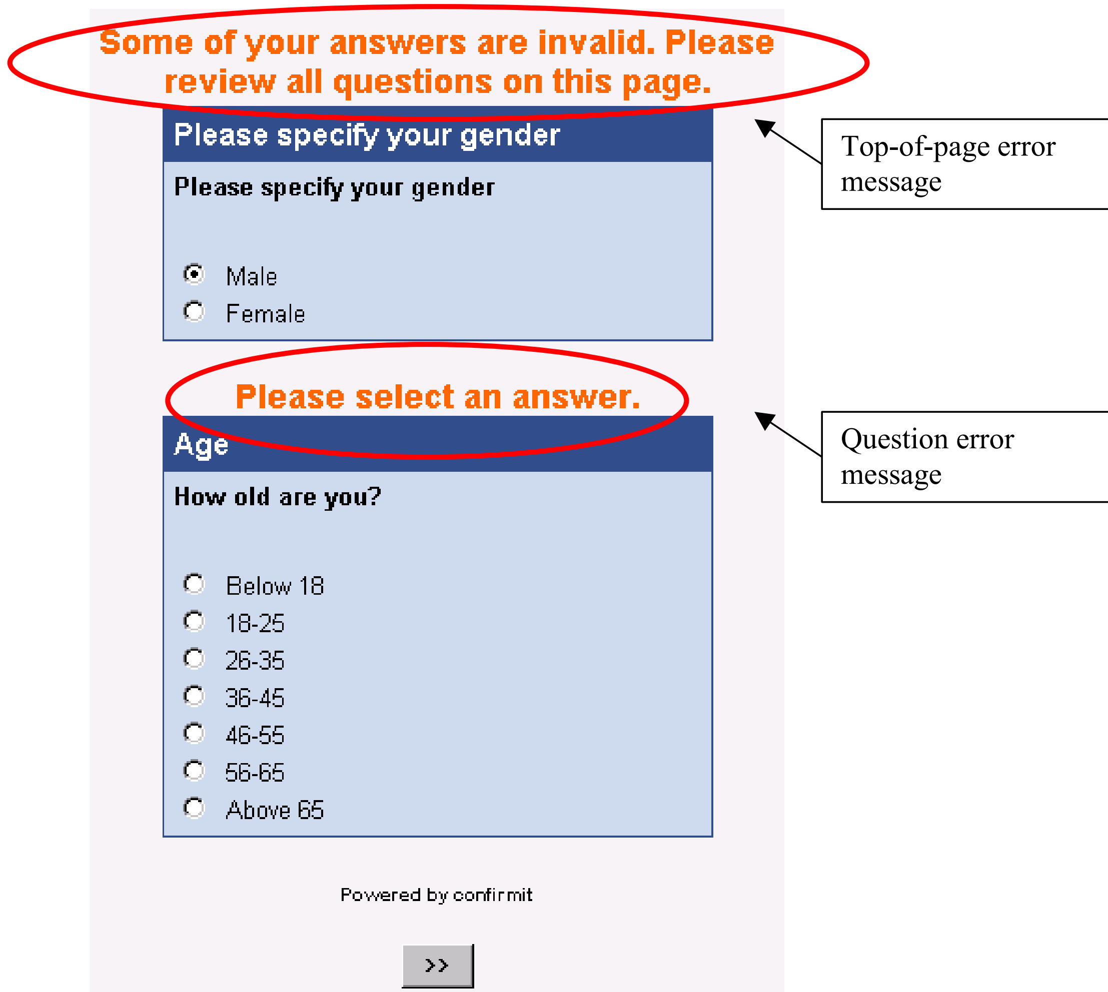
```
if(expression)

 {

   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 4 and 6).

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 APPENDIX B.

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.");

 }
```

password -

Text | Properties | Preview | Tracking | Languages | Save

General properties

Question type	OpenText
Created	March 20, 2002 7:57:25 AM EST
Modified	March 20, 2002 8:02:08 AM EST
Question ID	password
Field width	
Rows	1 Columns
Password	☑
Numeric	☐
Recoded var	☐
Background var	☐
Panel var	☐
Hidden	☐
Not required	☐
Indexed	☐
Deleted	☐

Validation code

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

1.5 Script Nodes

Script nodes typically contain code for

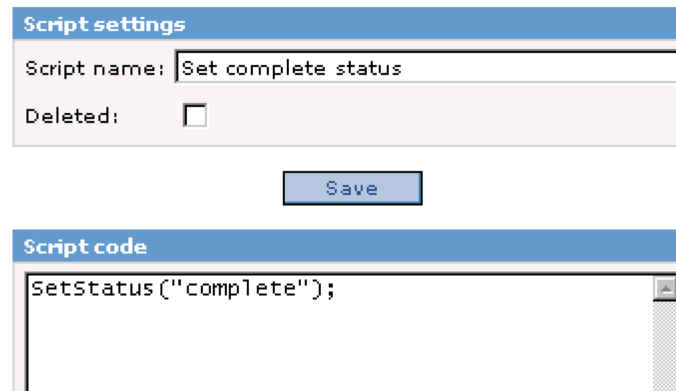
- Internal programming purposes.
- Defining functions used in precode masks, conditions, text substitution, other script nodes or validation code.
- Assigning values to different variables.
- Performing actions like sending an email, redirecting the respondent to a different URL, etc.

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 9.1.3.6, in a script node:

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



2 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 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 */
```

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.

3 Types and Variables

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**. The scope of a JScript variable used in Confirmit is limited. It can only be accessed within the same interview page. It is therefore recommended never to use a variable outside the script node or validation code where it is defined. So if you need to access the variable later in the questionnaire, use a hidden question. A hidden question can be accessed from anywhere in the questionnaire.

3.1 Naming

Variable names in JScript 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.**

There are some reserved words that cannot be used as question IDs in Confirmit and some that cannot be used as variable names in JScript. See appendix C in the Confirmit User Manual and a JScript reference manual.

3.2 Data Declaration

To identify a variable to the JScript interpreter, use an **assignment** statement and give it a value:

```
counter = 1;
```

This will define it as a *global* variable (but remember that the scope is limited to the script node/validation code where it is defined). It will initially have the value `1`. From now on every time you write `counter` in your script it will be replaced with the value `1` – until you have a new assignment, which changes its value.

3.3 Types

JScript is a *loosely typed* language. Loosely typed means you do not have to declare the data types of variables explicitly. In fact, JScript takes it one step further. You <u>cannot</u> explicitly declare data types in JScript. Actually the same variable can be used to contain values of different types in your script.

JScript will automatically convert the variable from one type to another when needed. This makes it easier to write efficient code, but also easier to make mistakes.

There are five *primitive types* of values in JScript: **Number**, **Boolean**, **String**, **Null** and **Undefined**. (Primitive types are types that can be assigned a single literal value. We will be looking at more complex types later.)

3.3.1 Number

In JScript, there is no distinction between integer and floating-point values. A JScript number can be either. (Internally, JScript represent all numbers as floating-point values).

Integer literals can be represented in

- Decimal (base 10) (`0,...,9,10,...,19,...`)
- Hexadecimal (base 16) – prefixed with 0x (`0,...,9,A,...,F,10,11,...,19,1A,...,1F,20,...`)
- Octal (base 8) – prefixed with 0 (`0,...,7,10,11,...,17,20,21,...`)

Decimal numbers are of course used most of the time. Both octal and hexadecimal numbers can be negative, but cannot have a decimal portion, and cannot be written in scientific (exponential) notation.

Floating-point literals:

- Numbers containing a decimal point. JScript uses . ("dot"/"period") as decimal point. (e.g. `9.34258`)
- Numbers containing an exponent (e or E) (e.g. `12.5e7`, which is the same as `125000000`)

Up to 20 significant digits may be used. For negative numbers the number will be prefixed by – (minus).

Additionally, JScript contains numbers with special values. 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.
- Positive Infinity. This is used when a positive number is too large to be represented in JScript.
- Negative Infinity. This is used when a negative number is too large to be represented in JScript.
- Positive and Negative 0. JScript differentiates between positive and negative zero.

3.3.2 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 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`.

3.3.3 String

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 values are enclosed in single (') or double (") quotation marks. You may use any of these marks, but always close the string 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 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
`\f`	Form feed
`\t`	Horizontal tab
`\b`	Backspace

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.

3.3.4 null

The `null` data type has only one value in JScript:

```
null
```

The `null` keyword cannot be used as the name of a function or variable.

A variable that contains `null` contains "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). It is used to identify a null, empty, missing or non-existent reference. You can use it to set an initial value that is different from other valid values, or you can erase the contents of a variable (without deleting the variable) by assigning it the null value. In this way you can prevent errors from working with variables that are not initialized.

Notice that in JScript, `null` is not the same as the number `0` (zero).

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 3.4.2.2. `set` is used to set the value of a question. It will be explained in chapter 7.1.

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

3.3.5 undefined

```
undefined
```

The `undefined` value is returned when you use a variable that has been declared, but has never had a value assigned to it.

3.4 Conversion between Types/Conversion Functions

JScript automatically converts values from one type to another when they are used in expressions. There are rather complex rules for the order of precedence of the different operators and types involved. JScript will favor string operators over all others, followed by floating-point, integer, and logical operators.

The table below shows what the result will be when using the operator + between values of different types.

row + column	string "12.34"	integer 123	float .123	logical true	logical false	null
string "test"	test12.34	test123	test0.123	testtrue	testfalse	testnull
integer 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:

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

See answer on page 180.

3.4.1 Conversion Functions in JScript

3.4.1.1 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. The expression

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

parses the *`string`* as an integer of base *`radix`*. *`radix`* is optional. But since a leading zero would indicate that the number is an octal number (see chapter 3.3.1), 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.

3.4.1.2 parseFloat

`parseFloat` returns a floating-point number converted from a string.

```
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}`).`

3.4.1.3 toString

`toString` is a method of any JScript object and 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.

3.4.2 Conversions Methods in Confirmit

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

3.4.2.1 toNumber

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

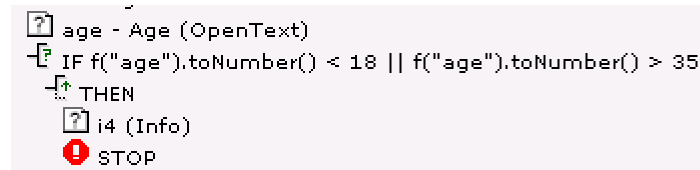
```
f(qID).toNumber()
```

`qID` is the question ID.

Example 8 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, which is 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
```



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

3.4.2.2 toBoolean

`toBoolean` is a method which 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

4 Operators and Expressions

4.1 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 operators. We have left out some operators that seldom are used in Confirmit scripts. For a full overview, please refer to a JScript language reference.

4.2 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.

4.3 Logical Operators

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

4.4 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 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 (`!=`/`!==`).

4.5 String Operators

Operator	Description

+	String concatenation

Strings can be joined with the string concatenation operator +:

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

will return

```
"Confirmit"
```

4.6 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".

4.7 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.

musical - Musical

Text | Answers | Properties | Preview | Tracking | Languages | Save

Answers

English	Precode	Weight	ColWidth	BgColor	Punch	KeepPos	Other
South Pacific							
Starlight Express							
Sunset Boulevard							
Sweeney Todd							
Tommy							
West Side Story							
Other, please specify	98						Yes

Add | Add predefined | Clear | Delete rows | Save

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 7.1). 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.

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 statements inside ^'s, however you can call a function that returns a string.

4.8 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 reads and evaluates expressions from left to right. The table below lists the order of precedence for the operators in JScript.

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

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 on page 180.

5 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 (but there is nothing wrong in using it between statements that occur on separate lines)
- A long 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 3.3.3).

5.1 Data Declarations

We have used this to declare JScript variables.

```
variableName = expression;
```

as for example in

```
x = 12;
```

which both identifies the variable x to the JScript interpreter and causes it to be implicitly **declared** as a variable initialized to the value `12`, which is a number.

5.2 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

```
y = x - 2;
```

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

5.3 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`.

5.3.1 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.

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.

5.3.2 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++;
```

a) …with x=4 and y=5

b) …with x=5 and y=4

a) See answers on page 180.

5.3.3 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.

5.4 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.

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);

  }
```

6 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. Objects will be covered in chapter 10, 11, 12 and 13. The Array object in particular is covered in chapter 13.5.

Arrays are objects that are capable of storing a sequence of values. These values are stored in indexed locations within the array.

For example, we can build an array containing the seven weekdays. First we declare a new array `weekday`:

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

Then we can store the names of the weekdays in the array with these statements:

```
weekday[0] = "Monday";

weekday[1] = "Tuesday";

weekday[2] = "Wednesday";

weekday[3] = "Thursday";

weekday[4] = "Friday";

weekday[5] = "Saturday";

weekday[6] = "Sunday";
```

You can then access the weekdays by referring to the individual elements of the array. The individual elements are referenced using the name of the array followed by the index of the array element enclosed in brackets. For example will

```
weekday[3]
```

return `"Thursday"`.

The **length** of an array is the number of elements in it. The array `weekday` has a length of 7.

6.1 Declaring Arrays

There are three ways of declaring an array:

```
arrayName = new Array(arrayLength);

arrayName = new Array();

arrayName = new Array(value0, value1, …, valuen);
```

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

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

The array could have been declared without specifying length:

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

Initially, the length would then have been 0. JScript automatically extends the length of an array when new array elements are initialized, so if we had gone through the same assignment statements as above

setting weekday[0] to weekday[6] we would end up with the same array of length 7. And even when the array is initially declared with new Array(7) we could add an 8th element

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

(which would have been a good idea to be able to finish projects before deadlines). Now the array has length 8.

The declaration and assignment of values could have been done in one operation:

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

giving the same array as in the previous examples (before adding "Lastday"). The expression

```
weekday[0]
```

would return "Monday" when declaring the array in one line as above, and the length of the array would be 7.

The indexing of arrays always starts on 0, so the first element will always have the index 0. The length of the array will consequently always be the index of the last element+1.

Even though you have empty elements in an array (with no value assigned) the length will always be 1 one more than the last index.

```
answers = new Array();

answers[5] = 28;

answers[55] = 45;
```

After the first statement (the declaration), the length of the array `answers` is 0. After the second statement the length of the array is 6, and after the third statement the length is 56 – even though the array has values assigned to only two items, namely those with index 5 and 55.

There are no restrictions on the values of the elements of an array. They can be of any type and don't even have to be of the same type. They can also refer to other arrays or objects:

```
mixedArray = new Array(1,false,2.3413,"Hello",new Array("Jim","Bob","Mike"))
```

The last element, `mixedArray[4]` contains an array as its value. To access these elements you have to use a second set of subscripts, for example will

```
mixedArray[4][0]
```

return the value `"Jim"`.

6.2 The length *Property*

As previously mentioned, JScript 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 10 and also the methods supported by the array object (chapter 13.5).

The **length** of an array is a property of an array. Properties of JScript 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`.

7 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 7.10.

We will be looking on the `f` function in chapter 11 as well.

7.1 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.

7.2 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`.

7.3 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).

7.4 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.

7.5 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.

7.6 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.

7.7 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.

7.8 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`.

7.9 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.

7.10 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.

i5 - Open text

Text | Properties | Preview | Tracking | Languages | Save

English

Title

Open text

Text

f("name").get(): ^f("name").get()^
f("name").label(): ^f("name").label()^

Instruction

7.10.1 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

7.10.2 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

<< >>

7.10.3 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

7.10.4 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

7.10.5 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.

7.10.5.1 Element of a Multi Question

Multi:

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

Powered by Confirmit

<< >>

7.10.5.2 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

<< >>

7.10.6 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).

Loop properties

Loop ID: weekdays
Field width:
Precode mask:
Randomize
Deleted
Title
English: Weekdays

Loop members

English	Precode	KeepPos	Active
Monday	1		Yes
Tuesday	2		Yes
Wednesday	3		Yes
Thursday	4		Yes
Friday	5		Yes
Saturday	6		Yes
Sunday	7		Yes

Add | Add predefined | Clear | Delete rows | Save

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.

7.10.7 3D Grid

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

7.10.8 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

i. "Important"

ii. "Design"

Areas of Importance

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

a) b)

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

Powered by confirmit

See answers on page 180.

7.11 Overview – Methods of Basic Variable Objects in Confirmit

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

8 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: the `while` statement, the `do while` statement and the `for` statement. The loop statements in JScript 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 loop, a set of statements is repeated until a condition is evaluated to `false`.

8.1 *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:

```
precodes = f("q2").domainValues(); //array with all precodes

i=0;

correctSum = true; //Boolean variable. Will be set to false when a sum is not correct

while(i<precodes.length && correctSum)

 {

  code = precodes[i]; //current precode

  //calculate the sum for one row:

  sum = 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 55.

The first statement,

```
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`.

```
i=0;

correctSum = 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 `0`, the first statement

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

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

A variable `sum` is then set in the statement

```
sum = 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.)

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

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

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

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

```
sum = 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":

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

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

```
sum = 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+".");

 }
```

8.2 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.

Example 14 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:

```
precodes = f("q2").domainValues(); //array with all precodes

i=0;

do

 {

  code = precodes[i]; //current precode

  //calculate the sum for one row:

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

  if(sum != 24)

   {

    correctSum = false;

   }

  else

   {

    correctSum = true;

   }

  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("q2")[code].label()+" is "+sum+".");

 }
```

8.3 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 can not 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")^
```

musicals_hidden - Musicals

Text | Answers | Properties | Preview | Tracking | Languages | Save

Answers

English	Precode	Weight	ColWidth	BgColor	Punch	KeepPos	Other
South Pacific							
Starlight Express							
Sunset Boulevard							
Sweeney Todd							
Tommy							
West Side Story							
^f("musicals_98_other")^	98						

Add | Add predefined | Clear | Delete rows | Save

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:

```
precodes = f("musicals").domainValues(); //all precodes

for(i=0;i<precodes.length;++i) //run through all precodes

 {

  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

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

the array 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]):

```
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

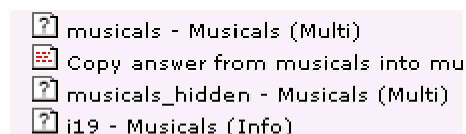
```
f("musicals_hidden")["2"].set(f("musicals")["2"].get());
```

which copies the second element from muscials 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:



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 answer like this:

Musicals

Which musicals have you seen?

- [x] Annie
- [ ] Anything Goes
- [ ] Beauty and the Beast
- [ ] Cats
- [ ] Chess
- [ ] Chicago
- [ ] A Chorus Line
- [x] Evita
- [ ] Fiddler On the Roof
- [ ] Godspell
- [ ] Grease!
- [ ] Guys and Dolls
- [ ] Hair
- [ ] Hello, Dolly!
- [ ] The King and I
- [ ] La Cage Aux Folles
- [ ] Les Miserables
- [ ] The Lion King
- [ ] Little Shop of Horrors
- [ ] Miss Saigon
- [ ] The Music Man
- [ ] My Fair Lady
- [ ] Oklahoma!
- [ ] Oliver!
- [ ] The Phantom of the Opera
- [ ] Rent
- [ ] Show Boat
- [ ] Song & Dance
- [ ] The Sound of Music
- [x] South Pacific
- [ ] Starlight Express
- [ ] Sunset Boulevard
- [ ] Sweeney Todd
- [ ] Tommy
- [ ] West Side Story
- [ ] Other, please specify Abba

the result of the piping will be:

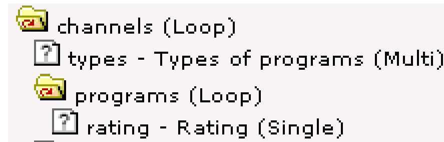
Musicals

You have seen these musicals: Annie, Evita, South Pacific and Abba

8.4 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"`.

8.5 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:

```
precodes = f("q2").domainValues(); //array with all precodes

for(i=0; i<precodes.length; i++)

 {

  code = precodes[i]; //current precode

  //calculate the sum for one row:

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

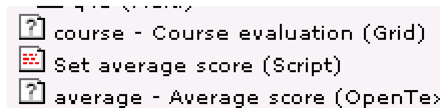
8.6 The continue Statement

```
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.




```
precodes = f("course").domainValues(); //all precodes of the rows in the grid

sum = 0; //this will hold the sum of the scores

count = 0; //this will hold the number of items

for(i=0;i<precodes.length;i++)  //iterate through the rows in the grid

 {

  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 divition with zero

 {

  f("average").set(sum/count); //calculate average

 }

else
```

```
{
 f("average").set(null); //sett 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.

8.7 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 3.1).

Specifying a label:

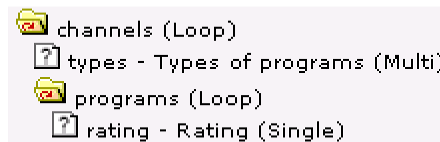
```
label:
```

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 loops in Confirmit (8.4) with two nested loops: 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 rating for each channel. Here is the list of iterations in the `channels` question:

Loop properties

Loop ID	channels
Field width	
Precode mask	
Randomize	☐
Deleted	☐

Title

Engli:

Loop members

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

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

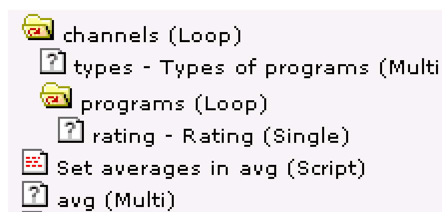
avg -

Text | Answers | Properties | Preview | Tracking | Languages | Save

Answers

English	Precode	Weight	ColWidth	BgColor	Puncl
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 have 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:

```
cprecodes = f("channels").domainValues(); //all precodes of the channels loop

Outer:

for(i=0;i<cprecodes.length;i++) //iterate through channels

 {

  ccode = cprecodes[i]; //precode of current channel

  //initialize the counter and sum for this channel:
```

```
  sum = 0;  count = 0;


  //the precodes of the program types for this channel:

  pprecodes = f("types",ccode).categories();


  Inner:

  for(j=0;j<pprecodes.length;j++) //run through the precodes in the programs loop

   {

    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 on page 180.

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:

```
error = false;

answers = f("ranking").categories();

 //an array with precodes of the items which has been ranked



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(i=0;i<answers.length;i++)

 {

  a = answers[i]; //precode of the item

  b = parseInt(f("ranking")[a].get(),10); //value (rank) assigned to that item

  /* check that rank b 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[b] != true && b>0 && b <= answers.length)

   {

    ranks[b] = 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.");

 }
```

9 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.

9.1 Built-in Functions in Confirmit

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

9.1.1 Arithmetic Functions

9.1.1.1 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 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 those 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:

```
precodes = f("q2").domainValues(); //array with all precodes

i=0;

correctSum = true; //Boolean variable. Will be set to false when a sum is not correct

while(i<precodes.length && correctSum)

 {

  code = precodes[i]; //current precode

  //calculate the sum for one row:

  sum = 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+".");

 }
```

9.1.1.2 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 21 *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 subheaders 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 answers on `q1`, 1 answers on `q2` and 3 answers on `q3`, the result of this function call will be the value `6` (`2+1+3`).

9.1.1.3 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 22 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).

9.1.1.4 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 23 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`.

9.1.2 Range

9.1.2.1 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 24 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:

Answers

English	Precode	Weight	ColWidth	BgColor	Punch
21 - 25	3				
26 - 30	4				
31 - 35	5				
36 - 40	6				
41 - 45	7				
46 - 50	8				
51 - 55	9				

Add | Add predefined | Clear | Delete rows | Save

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)
```

9.1.3 Context Information

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

Example 25 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.");

}
```

9.1.3.2 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/wi/<*project number*>/i.asp?r=1&s=JWUDIKLS

(If you are not using Confirmit on FIRM's ASP servers , but have a server installation of Confirmit, you of course have to replace survey.confirmit.com with the domain of your Confirmit interview servers. 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).

9.1.3.3 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.

9.1.3.4 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 26 *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.

9.1.3.5 InterviewStart and InterviewEnd

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 their values in scripts

```
InterviewStart()
```

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

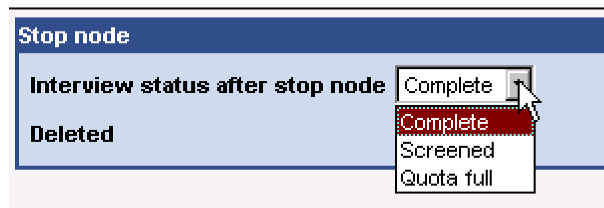
```
InterviewEnd()
```

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

We will get back to examples on how to use these functions in chapter 13.1.3.1 where they are used together with the `Date` object.

9.1.3.6 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 operates 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 9.1.8.2).

The `SetStatus` function will just set the interview status, not the `interview_end` time stamp. `interview_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 27 *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 count as completes for reporting etc.

9.1.3.7 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.

9.1.4 Browser Information

9.1.4.1 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 28 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());
```

9.1.4.2 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.nl/guidelines/internet_guidelines.htm**.

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 9.1.7.4 and Example 36.

9.1.4.3 Request

If you want to send values into the survey with the url, they can be included after the question mark after i.asp, separated by ampersands (&), e.g.

http://survey.confirmit.com/wi/<*project ID*>/i.asp?$variable_1=value_1\&variable_2=value_2\&\ldots\&variable_N=value_N$

You can also post or get to the interview page (i.asp) from a different web page.

To access these values, use `Request`:

```
Request(arg)
```

`arg` is the variable name, e.g. *variable₁*. `Request` returns the value that is assigned to that variable in the url, e.g. *value₁*.

It is very important that the script node with `Request` is the first node in the questionnaire, because once the respondent move to the next page, the values are lost.

Example 29 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/wi/<project ID>/i.asp?site=1
http://survey.confirmit.com/wi/<project ID>/i.asp?site=2
http://survey.confirmit.com/wi/<project ID>/i.asp?site=3
http://survey.confirmit.com/wi/<project ID>/i.asp?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.

English	Precode
www.yahoo.com	1
www.google.com	2
www.altavista.com	3
www.lycos.com	4

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"));
```

Now `source` can be used for reporting, quotas, logic etc. just as an ordinary question.

9.1.5 Quota check

9.1.5.1 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.

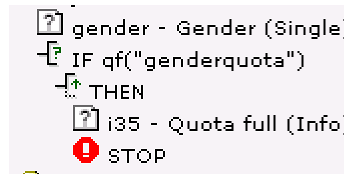
Please note that the usage of redirects, quotas, status-screened or other solutions with the principle purpose of avoiding the survey `"complete"` status being reached by a respondent having offered a reasonable amount of responses, is prohibited and will be regarded as an attempt to avoid transaction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

```
answers = f("brands").categories(); //precodes of all brands selected

for(i=0;i<answers.length;i++) //iterate through the precodes

 {

  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 it will have checked all of them `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:

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

9.1.6 Panel

9.1.6.1 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();

}
```

9.1.7 Classification Functions

9.1.7.1 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.

<u>Example 32</u> 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 9.1.2.1.

9.1.7.2 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.

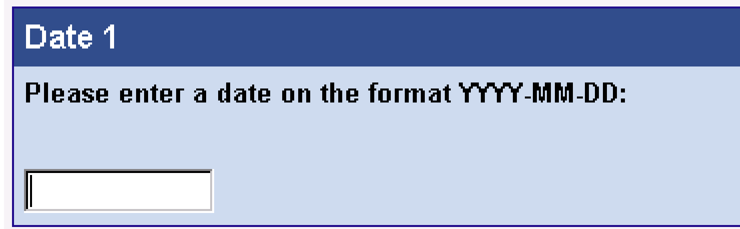
For both `IsDateFmt` and `IsDate` century interpolation for 2-digit years is handled with a break-point value of 10: values between 0-10 are interpreted as the years 2000-2010, while 11-99 is treated as 1911-1999.

`IsDateFmt` and `IsDate` returns 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 10, and chapter 13.1.3.2 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.

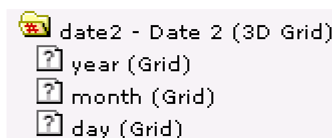


Here is a script you can use in the validation code of such a question:

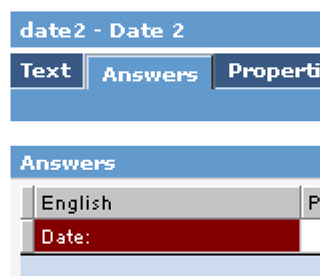
```
if(!IsDateFmt(f("date1").get(),"YYYY-MM-DD"))

 {

  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):

year -

Text | Scale | Properties | Preview | T

Answers

English	Precode	W
2000	2000	
2001	2001	
2002	2002	
2003	2003	

month -

Text | Scale | Properties | Preview | Tra

Answers

English	Precode	Weig
January	1	
February	2	
March	3	
April	4	
May	5	
June	6	
July	7	

Add | Add predefined | Clear | Dele

day -

Text | Scale | Properties | Preview | Tra

Answers

English	Precode	Weig
1	1	
2	2	
3	3	
4	4	
5	5	
6	6	
7	7	

Add | Add predefined | Clear | Dele

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.");

 }
```

9.1.7.3 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.");

 }
```

9.1.7.4 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 subnetting 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 9.1.4.2.

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")
```

9.1.8 General Utilities

9.1.8.1 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, 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 APPENDIX D 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 13.3.5.2.

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 have provided her or his email address in an open text question called `email`:

```
//build body of email:

body = "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

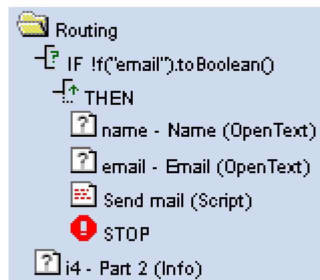
need more than one email address in either to, cc or bcc, separate the addresses with semi colons (;) inside the string, e.g. `"support@confirmit.com;consulting@confirmit.com"`.

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.");

 }
```

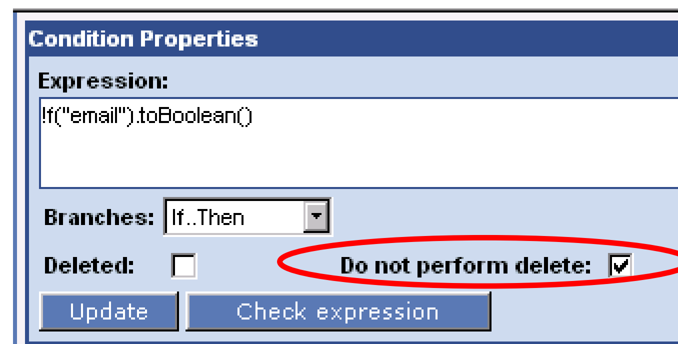
After the `email` question there is a script node, which does the emailing (change "survey.confirmit.com" if you are not using FIRM's ASP):

```
body = "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="+CurrentSID()+"\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")^
```

9.1.8.2 Redirect

```
Redirect(url,{noexit})
```

`Redirect` can be used to redirect the respondents to a different site (*`url`*). Usually this is done to bypass the default “Thank you” page and send the respondents to a customized "Thank you" page instead, but it can also be used to redirect respondents to a different survey or to 3rd party interview applications, e.g. conjoint applications (which may redirect back to the Confirmit survey).

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 9.1.3.6). 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");
```

```
Redirect("http://www.google.com",false);
```

`Redirect` is used in combination with `SetStatus` so that the respondent gets the correct status before being redirected out of the survey.

9.2 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.

9.2.1 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,p2,...,pn)

 {

   <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.

Example 40 Function to Make a Multi Question Required

In Example 25 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.

9.2.2 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 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();
```

9.2.3 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 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)

 {

  precodes = f(from).domainValues();

  for(i=0;i<precodes.length;i++)

   {

    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 21, 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:

officesa - Offices

Text | Answers | Properties | Preview

Answers

English	Precode	W
New York	1	
Los Angeles	2	
Chicago	3	

officesb - Offices

Text | Answers | Properties | Preview

Answers

English	Precode	We
London	4	
Paris	5	
Rome	6	

officesc - Offices

Text | Answers | Properties | Preview

Answers

English	Precode	Weig
Kuala Lumpur	7	
Singapore	8	
Sydney	9	
Hong Kong	10	

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");
```

9.2.4 Functions with a Variable Number of Arguments

In JScript you can define functions that take a variable number of arguments, using the *arguments* array. JScript automatically creates the *arguments* array for each function invocation.

```
FunctionName.arguments.length
```

returns the number of arguments in the function call. To refer to each argument, use indexing like in all other arrays: `0` for the first argument, `1` for the second, and so on:

```
FunctionName.arguments[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 9.1.1).

9.2.5 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,b)

 {

  return (n/b)*100;
```

```
 }
```

9.2.6 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`.

```
loopPrecodes = f("loop1").domainValues();

for(i=0;i<loopPrecodes.length;i++)

 {

  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)

 {

  precodes = f(qID).domainValues();

  for(i=0;i<precodes.length;i++)

   {

    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`:

```
var variableName;
```

or

```
var variableName = expression;
```

So, if we rewrite the function definition using local variables, the script will do what we intend:

```
function CheckGridAnswer(qID,iter,val)

 {

  var precodes = f(qID).domainValues();

  var i;

  for(i=0;i<precodes.length;i++)

    {

     var code = precodes[i];

     if(f(qID,iter)[code].get() == val)

       {

        return true;

       }

    }

  return false;

 }
```

10 Objects

JScript is an **object-based** language.It is however not a full **object-oriented language** like Java or C++, because it is lacking some Object-Oriented Programming features**.** It does however provide several important Object-Oriented Programming features that simplify the design of script code. These features make it possible to write scripts in a more intuitive, modular and reusable manner.

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 (in the form of an array or object) that is a member of an object.

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)
```

Chapter 10.1 may to some seem a bit theoretical since it requires some knowledge of the concepts of Object-Oriented Programming to understand it fully. But don't panic if you do not understand it; it is not really required to understand it to be able to use the built-in objects in JScript and Confirmit. It is provided to explain some concepts of Object-Oriented Programming and their support in JScript. If you really dislike theory, skip directly to chapter 10.2. The examples and explanations given in the rest of this chapter and in chapters 11, 12 and 13 will provide you with enough information to be able to take full advantage of the objects in JScript and Confirmit.

We will not be covering how you can build your own object types in JScript, since this usually is of limited interest for the kind of scripts you write in Confirmit. For more information on building your own JScript objects, please use a JScript or JavaScript reference.

10.1 Object-Oriented Programming

Let us start by looking at the basic concepts of an object-oriented language, and identify which of these features that are included in JScript.

Object-Oriented Programming (OOP) is a type of programming in which programmers define not only the data type of a data structure, but also the types of operations (functions) that can be applied to the data structure. In this way, the data structure becomes an object that includes both data and functions.

This is similar to how we defined the data types of JScript at the beginning of this manual. A definition of a data type consists of a set of all values permitted for that type and a set of all operations allowed to be used on those values.

10.1.1 Defining and Instantiating Object Types

An **object type** is a template from which specific objects of that type are created. In Object-Oriented languages like Java an object type is referred to as a **class.**

Above we described JScript objects as collections of properties and methods. As an example let us think of a member of a panel as an object type. We call this object type `PanelMember` and give it properties `email`, `username`, `password`, `points`, as well as an array `completedProjects` with project IDs of surveys that the panelist has completed and an array `newProjects` with project IDs of new surveys. Then we could define methods `changePassword` and `changeEmail` to change the panelist's password and email, a method `addPoints` to add points to the panelist's score, a method `removePoints` to remove points, and a method `addProject` to add a new survey to the list of available surveys.

By **defining** this object type, we haven't created any panelists. We have just defined a template for the creation of a panelist. To create a specific panelist, we will have to **instantiate** it, i.e. creating a specific instance of that type of object, e.g. a panelist with username "James".

When you have defined an object type, you can use that object type to define other objects types.

10.1.2 Object Composition

One way of using object types to define other object types is to define primitive object types that serve as simple building blocks from which you can build more complex types. This is referred to as **object composition**.

The `PanelMember` object type we defined in the previous chapter could be used as a part of a panel object type. A panel object type could be part of an object type for lists of panels.

One of the most important aspects of object composition is the capability to support **object reuse**. It is desirable that objects are defined so that they are easy to reuse. This simplifies the task of writing new scripts and makes it less prone to errors, thus saving time.

JScript object types may be defined in terms of other pre-defined or user-defined object types.

10.1.2.1 Encapsulation

Encapsulation is the packaging of the properties and methods of an object into a container with an appropriately defined interface. The interface of an object must provide the properties and methods that give you what you need to interface with the object, but without providing methods and properties that would allow it to be misused.

This is not supported in JScript.

10.1.2.1.1 Modularity

An object should be complete in and of itself and not accessing other objects outside their defined interfaces. This is called **modularity**. Modular objects are said to be "loosely coupled", which means that dependencies between objects are minimized so that changes in the internals of one object do not require changes in other objects that make use of the object. This ensures that your scripts will continue to work even though you reprogram the internals of an object.

JScript does not provide any features that enforce modularity. So it is up to the programmer to define objects in a modular fashion.

10.1.2.1.2 Information Hiding

The objects should be built so that only what is required to interface the object should be available from outside the object. There may be methods that are used internally which only are accessible from other methods within the object, but not from the outside. This is called **information hiding**. In the `panelMember` object there could for example be methods to initialize the `points` property when a

new panelist is created (by setting it to 0 points), but you don't want that method to be available from the outside.

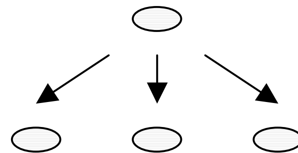
JScript does not support information hiding.

10.1.3 Inheritance

The second way of constructing object types from other object types is through **inheritance**. This is a top-down approach, in which you at the higher level create more abstract object types. From this higher level, you can create more concrete object types in a parent-child relationship. The child inherits all of the properties and methods of its parents. So then you have either the option of not redefining these properties and methods, or you can redefine some of them if needed. You can define more sophisticated, tailored object types from other already defined object types by adding properties and methods needed to differentiate the new objects from their parents.

JScript does not provide any language features that support inheritance between object types.

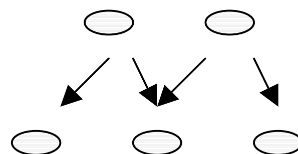
10.1.3.1 Single Inheritance



Some languages (for example Java) require that a child inherits methods and properties from just one parent. The parent can have several children. This is called **single inheritance**.

10.1.3.2 Multiple Inheritance

Other languages (for example C++) support **multiple inheritance**, where a child can inherit methods and properties from more than one parent.



10.1.4 Polymorphism

Polymorphism is when you have more than one implementation of a method inside an object type. This means that you define several methods with the same name. The difference is their type and the number of arguments they receive. This allows you to use a standard method to do a particular operation and to define different forms of the method to be used with different arguments. This promotes standardization and reusability.

JScript's way of supporting polymorphism is by using the `arguments` array for function definitions.

10.2 Objects in JScript

10.2.1 Properties

Properties are used to access the data values contained in an object. The properties of an object in JScript can both be updated and read. However, some properties of the predefined JScript objects are read-only.

An object's properties are accessed by combining the object's name and its property name:

```
objectName.propertyName
```

10.2.2 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 we learnt when we studied JScript functions, arguments in method calls are optional.

10.2.3 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:

```
a = new Array();
```

An array is a special object type. The general syntax for creating an instance of an object is like this:

```
variable = 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.

11 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 7).

11.1 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) (`loopiter1,loopiter2,...,loopitern`) 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")
```

11.2 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 examples 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.

11.3 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:

```
sleep = f("q2");

work = f("q3");

leisure = f("q4");
```

```
precodes = sleep.domainValues(); // array with all precodes

i=0;

correctSum = true;

while(i<precodes.length && correctSum) //iterate through precodes (rows)

 {

  code = precodes[i]; //current precode

  //calculate sum:

  sum = 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.

11.4 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=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)

 {

  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=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;

}
```

11.5 Methods

The methods of the form objects are listed below. The methods are described in detail in chapters 3.4.2, 7, and 12.3.

Methods	CODED	OPEN	NUMERIC	DICHOTOMY	COMPOUND
toBoolean	✓	✓	✓	✓	✓
toNumeric	✓	✓	✓	✓	
get	✓	✓	✓	✓	
set	✓	✓	✓	✓	
label	✓	✓	✓	✓	✓
value	✓				
valueLabel	✓				
domainValues	✓				✓
domainLabels	✓				✓
categories					✓
categoryLabels					✓
values					✓
inc	✓				✓
size	✓				✓
union	✓				✓
isect	✓				✓
diff	✓				✓
members	✓				✓

12 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.

12.1 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`.

12.2 Functions Returning Sets

12.2.1 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)
```

12.2.2 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"`.

12.2.3 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())
```

It is important that the questions prefix and car is set up on different pages. Use for example a directive to separate them.

12.2.4 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 11).

12.3 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 just an instance of the set object, because these methods are not defined for form objects.

12.3.1 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")
```

12.3.2 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)

 {

  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):

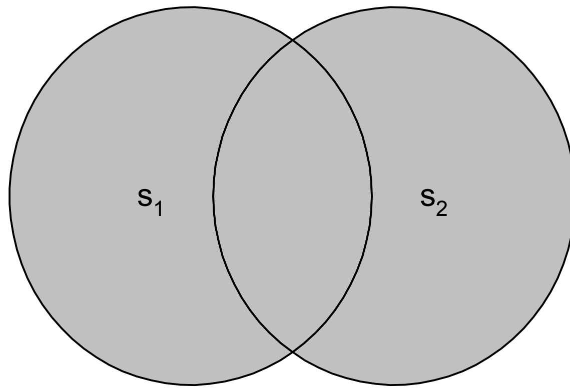
```
if(!CheckNumberOfAnswers(3))

 {

  RaiseError();

  SetQuestionErrorMessage(LangIDs.en,"Please do not select more than 3 items.");

 }
```

12.3.3 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 is: Showing the items answered on both questions, on any of the questions, on one but not the other etc.

```
s1.union(s2)
```

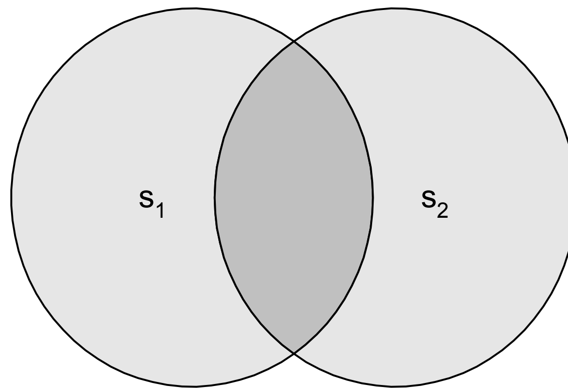
The **union** of two sets s_1 and s_2 is the set obtained by combining the members of both sets. So `union` will return a set consisting of all elements in s_1 **or** s_2.



s_1.union(s_2)

s_1.isect(s_2)

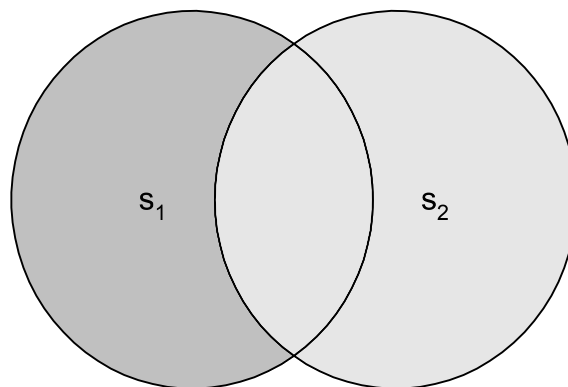
The **intersection** of two sets s_1 and s_2 is the set of elements common to s_1 and s_2. So `isect` will return a set consisting of the elements that are both in s_1 **and** s_2.



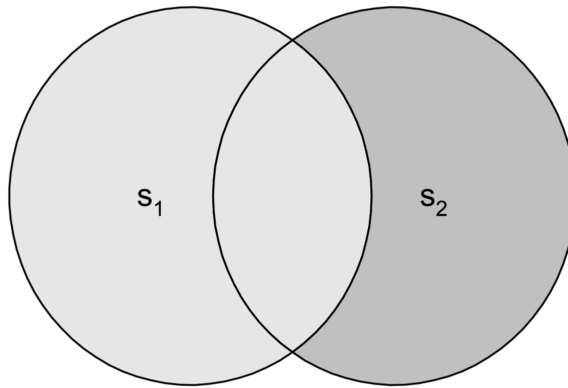
s_1.isect(s_2)

s_1.diff(s_2)

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)



s2.diff(s1)

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:

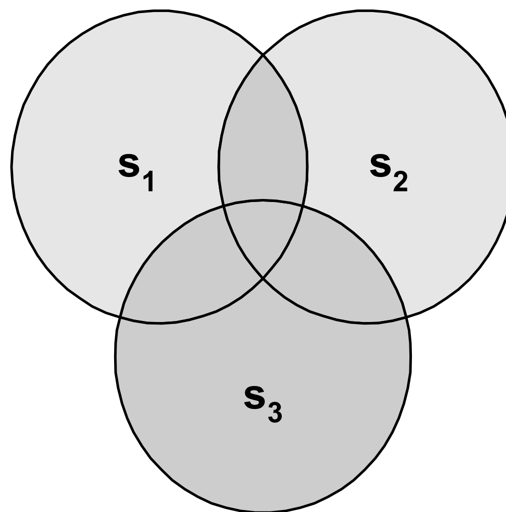
```
f("q1").union(set("DK"))
```

12.3.4 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 both in the sets s_1 and s_2, or in set s_3:

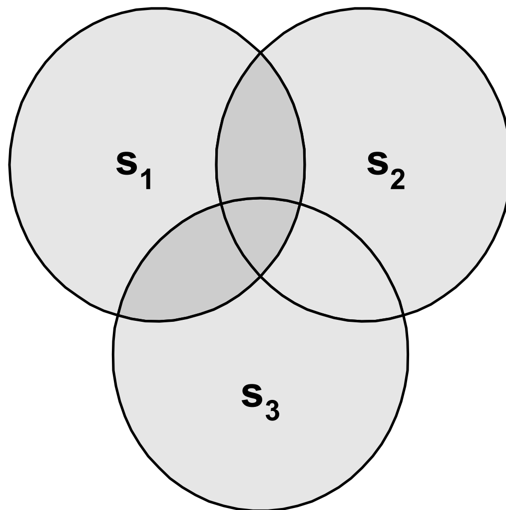


s_1.isect(s_2).union(s_3)

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

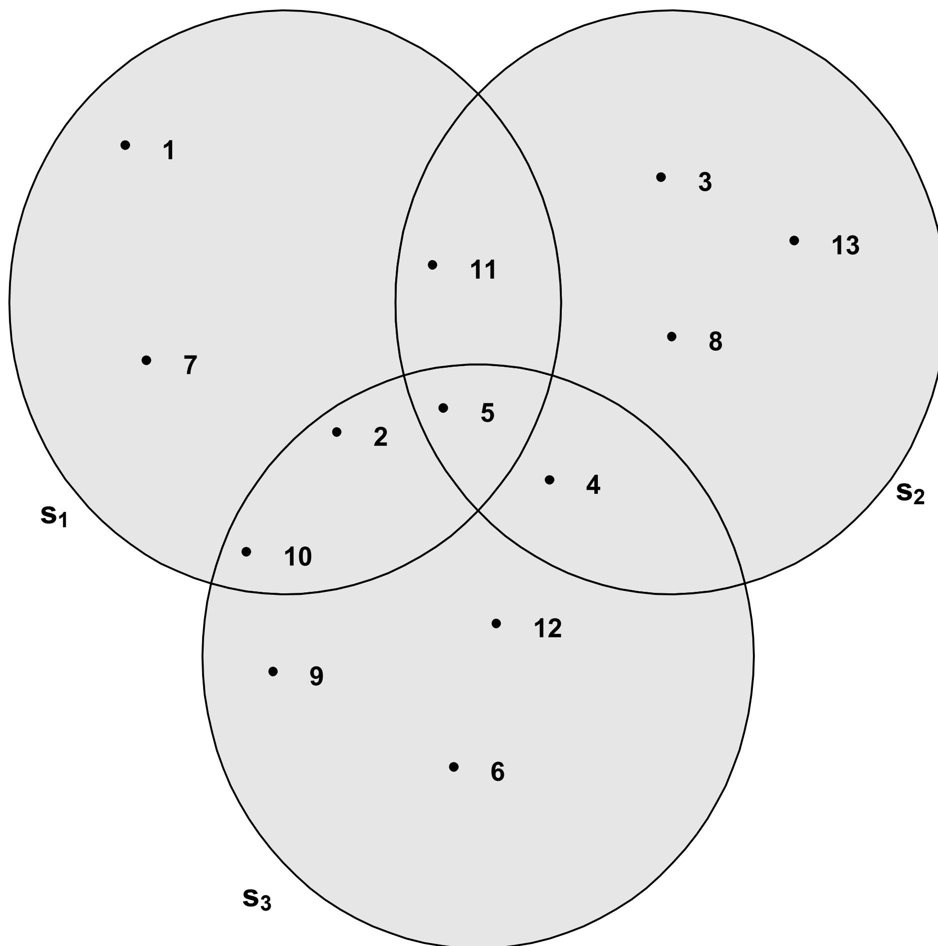
```
s1.isect(s2.union(s3))
```

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:



`s₁.isect(s₂.union(s₃))`

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))

b) s_1.union(s_3).diff(s_2)

c) s_1.isect(s_2.diff(s_3))

d) s_1.isect(s_2).union(s_2.isect(s_3)).union(s_3.isect(s_1))

See answers on page 181.

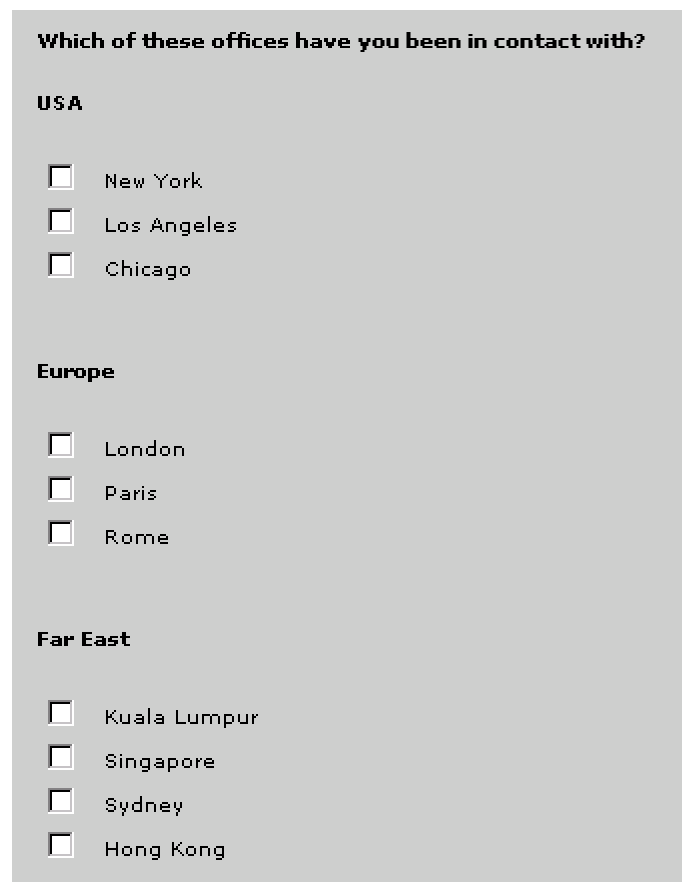
12.3.5 members

```
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:



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

s = f("officesa").union(f("officesb")).union(f("officesc"));



precodes = s.members(); //convert the set of precodes to an array



for(i=0;i<precodes.length;i++) //iterate through those precodes

 {

  code = precodes[i]; //current precode

  f("offices")[code].set("1"); //set this item to "selected"

 }
```

12.3.5.1 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 return the set resulting of adding *precode* to the set s_1. If *precode* is already in the set s_1, the set returned will be equal to s_1. `remove` will return the set resulting of removing *precode* from the set s_1. If *precode* is not in the set s_1, the set returned will be equal to s_1.

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;

  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`):

```
ScoreFilter("q3")
```

12.4 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:

1. The time it takes for the respondent to load the survey pages will increase because there are more scripts to execute.

2. 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 `q3`.

```
var form = f("q3");

var precodes = form.domainValues();
```

```
var i;

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")
```

13 Predefined JScript Objects

In this chapter we will describe some of the objects provided in JScript. 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 Objects, for example objects like `window`, `frame`, `document`, `navigator`, `form` etc. used in client-side scripting, please consult a JScript or JavaScript reference.

13.1 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 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. Which should be sufficient for most of us.

13.1.1 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 <u>integer</u>, *`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 <u>string</u>, *`dateVal`* is parsed according to the rules in the `parse` method (see description in 13.1.2.1.1).

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 integers 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.

13.1.2 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.

13.1.2.1 Static Methods

13.1.2.1.1 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 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.

13.1.2.1.2 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 13.1.1).

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 redefines that number as two minutes and 30 seconds.

The difference between the `UTC` method and the `Date` object constructor that accepts a date is that the `UTC` method assumes `UTC`, and the `Date` object constructor assumes local time. (Local time when doing server side scripting will be using 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)
```

13.1.2.2 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 a 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 months use the numbers 0 to 11. 0 is January and 11 is December.

13.1.2.2.1 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 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.

13.1.2.2.2 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."

13.1.2.2.3 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

13.1.2.2.4 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.

13.1.2.2.5 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.

13.1.2.2.6 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.

13.1.2.2.7 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.

13.1.2.2.8 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.

13.1.2.2.9 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.

13.1.2.2.10 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).

13.1.2.3 Conversion Methods

13.1.2.3.1 valueOf

```
dateObj.valueOf()
```

`valueOf` returns the stored time value in milliseconds since midnight, January 1, 1970 UTC.

13.1.2.3.2 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 is provided for backwards compatibility only. It is recommended that you use the `toUTCString` method instead.

13.1.3 Confirmit Date Functions

Confirmit provides four 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). `IsDate` and `IsDateFmt` are used for validation of dates.

13.1.3.1 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 `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())

 {

  d1 = new Date(InterviewStart());

  d2 = new Date();



  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 a hidden open text question with numeric property (`time_part2`):

```
if(!f("time_part2").toBoolean())

 {

  d1 = new Date(InterviewStart());

  d2 = new Date();



  diff = ((d2.getTime() - d1.getTime())/1000)-f("time_part1").toNumber();



  f("time_part1").set(diff);

 }
```

13.1.3.2 IsDateFmt and IsDate

These functions are also described in 9.1.7.2. 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`).

```
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

 {

  dt = new Date();

  dt.setFullYear(d.year,d.month-1,d.day);

  current = new Date();

  if(dt.valueOf() > current.valueOf())

    {

     RaiseError();

     SetQuestionErrorMessage(LangIDs.en,"Please do not enter a date after the current date.");

    }

 }
```

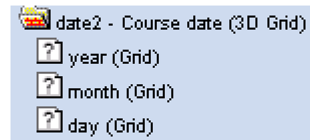
If the date provided is not valid, `IsDateFmt(f("date1").get(),"YYYY-MM-DD")` will return `null`. Used in a condition `null` will yield the Boolean value `false`. If the date is valid, `IsDateFmt` will return an object. Used in a condition this will yield `true`. So the condition `!d` will provide an 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:



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` it has to be a number between 1 and 12 and for `day` it has to be a number 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.

```
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

 {

  dt = new Date();

  dt.setFullYear(d.year,d.month-1,d.day);
```

```
  current = new Date();

  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:

```
if(!IsDateFmt(f("date3").get(),"YYYY-MM-DD"))

 {

  RaiseError();

  SetQuestionErrorMessage(LangIDs.en,"Invalid date. Please correct using the format
YYYY-MM-DD");

 }
```

Then you can have a script node with the following function:

```
function FindDay(qID)

 {

  var d = IsDateFmt(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:

English

Title

Result

Text

The weekday of ^f("date3")^: ^FindDay("date3")^.

Instruction

This will for example give this output if the answer on the date question was 2002-04-20:

Result

The weekday of 2002-04-20: Saturday.

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 on page 181.

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 respondents age and set it in a hidden question `age` with the following script, which is placed in the validation code of the question:

```
d = IsDateFmt(f("dob").get(),"YYYY-MM-DD");

if(!d)

 {

  RaiseError();

  SetQuestionErrorMessage(LangIDs.en,"Invalid date. Please use the format YYYY-MM-
DD");

 }

else

 {

  birthday = new Date();
```

```
  birthday.setFullYear(d.year,d.month-1,d.day);
  today = new Date();
  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);
}
```

13.2 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 13.2.2.2 and the random method in chapter 13.2.2.3.

13.2.1 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.

13.2.2 Methods

13.2.2.1 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.

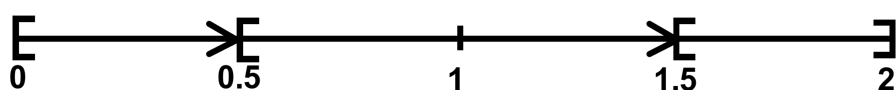
```
Math.atan2(x,y)
```

Returns the angle of the polar coordinate corresponding to (x,y)

13.2.2.2 Rounding

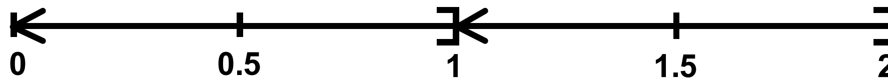
```
Math.round(x)
```

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.



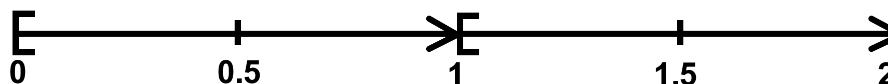
```
Math.ceil(x)
```

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`:

```
yearly = f("yearly").toNumber();

monthly = yearly/12;

f("monthly").set(Math.round(monthly*100)/100);
```

13.2.2.3 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`.

```
fromForm = f("brands");

toForm = f("random_brands");

numberOfItems = 3;



available = new Set();

available = available.union(fromForm);

selected = new Set();

if(available.size() <= numberOfItems)

 {

  selected = available;

 }

else

 {

  while(selected.size() < numberOfItems)

   {

    precodes = available.members();

    randomNumber = Math.random()*precodes.length;

    randomIndex = Math.floor(randomNumber);

    selectedCode = precodes[randomIndex];

    available.remove(selectedCode);

    selected.add(selectedCode);

   }

 }



precodes = fromForm.domainValues();

for(i=0;i<precodes.length;i++)

 {

  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)

   {

    precodes = available.members();

    randomNumber = Math.random()*precodes.length;

    randomIndex = Math.floor(randomNumber);

    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)`:

```
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:

```
randomIndex = Math.floor(randomNumber);
```

This means that `randomIndex` will be one of `0,1,2,...,precodes.length-1`, i.e. 0,1,2,3 in our example.

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:

```
form = f("part");

if(!form.toBoolean())

 {

  precodes = form.domainValues();

  randomNumber = Math.random()*precodes.length;

  randomIndex = Math.floor(randomNumber);

  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.

13.2.2.4 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 $\texttt{number}_1, \ldots, \texttt{number}_N$ 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 9.1.1.4) when working on questions, since these functions automatically converts the answers to numbers.

13.2.2.5 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.

13.2.2.6 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.

13.3 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.

13.3.1 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}";
```

13.3.2 Properties

```
strVariable.length

"String Literal".length
```

`length` returns the length of a `String` object, an integer that indicates the number of characters in the `String` object.

13.3.3 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 (just like the index of an array). The index of the last character will always be 1 less than the string's length. This is similar to indexing in arrays.

13.3.4 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 any other type, this will convert the variable to type string.

13.3.5 Methods

A few methods of the string objects that use Regular Expressions are described in chapter 13.4.4. The rest is described below.

13.3.5.1 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.

13.3.5.2 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"`:

```
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.

```
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 9.1.8.1). If you send HTML emails, you should convert the Unicode to HTML instead.

```
txt = "&#35613;&#35613;&#24744;&#25765;&#20887;&#21443;&#33287;"; //HTML
```

13.3.5.3 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.");

 }
```

13.3.5.4 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.

13.3.5.5 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:

```
txt = "The methods of the String Object can be used for text manipulation.";

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 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

You refer to a multi question, e.g. `brands`, in response piping with `^`'s in a question text 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`).

```
body = "";

body += "Here are the answers on the brands question:\n\n"



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.

13.3.5.6 Splitting and Joining Strings

```
stringObj.split({separator{, limit})
```

`split` returns the array of strings that results when a string is separated into substrings. *`seperator`* is a string or an instance of a Regular Expression object (see 13.4.3) 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/wi/pXXXXXXXX/i.asp?products=1x15x17x19
http://survey.confirmit.com/wi/pXXXXXXXX/i.asp?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 the `Request` function (see 9.1.4.3), 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:

```
txt = Request("products")+"";

form = f("products");



productArray = txt.split("x");
```

```
for(i=0;i<productArray.length;i++)

 {

  form[productArray[i]].set("1");

 }
```

13.3.5.7 Retrieving the String Value

```
strObj.toString()

strObj.valueOf()
```

Both toString and valueOf returns the string value of the String object.

13.3.5.8 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* + '}'

13.4 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.

Regular Expressions are implemented as Regular Expression objects and are created as follows:

```
re = /pattern/{flags};
```

(like Regular Expressions in Perl) or

```
re = new RegExp("pattern",{"flags"});
```

(like normal syntax for instantiating an object in JScript)

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:

```
re = new RegExp("confirmit","g");
```

is the same as

```
re = /confirmit/g;
```

This matches all occurrences of "confirmit".

13.4.1 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.

13.4.1.1 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.

13.4.1.2 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 subexpression. Subexpressions can be captured for later use.
`*`	Matches the preceding subexpression zero or more times.
`+`	Matches the preceding subexpression one or more times.
`.`	Matches any single character except the newline character `\n`.

`[`	Marks the beginning of a bracket expression.
`?`	Matches the preceding subexpression 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?"

13.4.1.3 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 whitespace character including space, tab, form-feed, etc.
`\S`	Matches any non-whitespace character.
`\t`	Matches a tab character.
`\v`	Matches a vertical tab character.

x represents a character in the range A-Z or a-z.

13.4.1.4 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:

1. Escape it with a backslash:

   ```
   [\-]
   ```

2. Put the hyphen character at the beginning or the end of the bracketed list. The following expressions matches all lowercase letters and the hyphen:

   ```
   [-a-z]

   [a-z-]
   ```

3. 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 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* represents characters.

13.4.1.5 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 subexpression **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 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.

13.4.1.6 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 nonword 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 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 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/
```

13.4.1.7 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 **subexpressions**. 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 submatch is captured. In this example we really do not need that submatch.

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 submatches, 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 submatches.

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 submatch.

```
/^(?: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.
`(?: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.

13.4.1.8 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 submatch is stored as it is encountered from left to right in a Regular Expressions pattern. The submatches are numbered beginning at 1 and continuing up to a maximum of 99 submatches. 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 Regular Expression uses a single subexpression to check for duplicates:

```
/\b([a-z]+) \1\b/gim
```

The subexpression 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 submatch, that is, the second occurrence of the word. `\b` is used for word boundary, so that the check is done on complete words.

13.4.1.9 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_]`.

13.4.1.10 Hexadecimal and Octal Escape Values and Unicode Characters

Character	Description
`\xn`	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 13.4.1.8).
`\un`	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 subexpressions, *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 subexpressions, *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).

13.4.2 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

13.4.3 The Regular Expression Object

We have already seen the two ways of instantiating a Regular Expression object:

```
re = /pattern/{flags}
```

and

```
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.

13.4.3.1 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 submatches 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`:

```
num = f("phone").get();
```

```
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 subexpression 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 is 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 182.

13.4.4 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 submatches 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`:

```
brandsAnswers = f("brands").categoryLabels();

brandsAnswers = brandsAnswers.toString();

brandsAnswers = brandsAnswers.replace(/,/g,"\n");
```

If the mail is sent as HTML, you have to replace the commas with the HTML <br> tag instead:

```
brandsAnswers = f("brands").categoryLabels();

brandsAnswers = brandsAnswers.toString();

brandsAnswers = brandsAnswers.replace(/,/g,"<br>");
```

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`:

```
pcode = f("postalcode").get();

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 is not removed when the value is stored, 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 on page 182.

13.5 The Array Object

We have already been using the `Array` object a lot. As we showed in chapter 6, an array is declared in three ways:

```
arrayName = new Array(arrayLength);

arrayName = new Array();

arrayName = new Array(value0, value1, …, valuen);
```

The Array Object has one property, the length of the array (number of items):

```
arrayName.length
```

In this chapter we will look at the methods of the Array object.

13.5.1 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.

13.5.2 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`:

```
brandsAnswers = f("brands").categoryLabels();

brandsAnswers = brandsAnswers.join("\n");
```

If the mail is sent as HTML, you have to replace the commas with the HTML <br> tag instead:

```
brandsAnswers = f("brands").categoryLabels();

brandsAnswers = brandsAnswers.join("<br>");
```

13.5.3 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({item1 {item2 {... {itemN }}}})
```

`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({item1{, item2 {, ... {, itemN}}}})
```

`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.

13.5.4 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({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. If 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 *`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:

q1 - Quality

Text | Answers | Scale | Properties | Preview | Tracking | Languages | Save

Answers

English	Precode	Weight	ColWidth	BgColor	Punch	KeepPos	Other
Apple	1						
Compaq	2						
Dell	3						
HP	4						
IBM	5						
Other provider 1:	97						**Yes**
Other provider 2:	98						**Yes**

Add | Add predefined | Clear | Delete rows | Save

This is a validation code that can be used on such a question:

```
form = f("q1");
precodes = a("q1").diff(set("97","98")).members(); //all precodes except "other"

//sort the precodes so the items are checked in the same order as they are displayed:
precodes = precodes.sort(NumSort);

notAnswered = new Array(); //array to hold text of not answered items

for(i=0;i<precodes.length;i++)
 {
  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()) ||
(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

```
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.

13.5.5 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.

14 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 apply. In this chapter we will explain a number of functions that make it easier to build customized error messages for the default answer checks.

14.1 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.
`SizeError()`	returns `true` if one or more open-ended answers exceeded the maximum length specified in field width, `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.
`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.

14.2 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.

14.2.1 Missing Required

```
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^."));

 }
```

14.2.2 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."));
```

```
}
```

14.2.3 Other-Specify Verification

```
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."));

 }
```

14.2.4 Answer Size Tests

```
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."));

 }
```

14.2.5 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^."));

 }
```

14.2.6 Numeric Error Tests

```
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."));

 }
```

14.2.7 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(LangID.en,
  ErrorTemplate("Please enter a number with no more than ^PRECISION^ digits for
^PRECISION_ERRORS^."));

 }
```

14.2.8 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^."));

 }
```

14.2.9 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^."));

 }
```

15 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.

15.1 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 2.

15.2 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.

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`.

15.3 Spaces and Line Breaks

Except from within string expressions, you can use line breaks and spaces as much as you like when writing JScript 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.

15.4 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>;
```

Which will be very different from

```
if (condition)

 {

  <statement1>;

  <statement2>;

}
```

15.5 Semi Colon

Statements are separated with semi colon (`;`) in JScript. 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.

15.6 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.

15.7 Writing Efficient Code

Here are a few tips on how to write code that is efficient:

1. 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.

2. 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.

3. Avoid using function calls in precode masks. Set a hidden multi question instead.

4. Use methods of the form objects like `domainValues` and `categories` instead of hard coding precodes(1,2,3,…) in your scripts.

5. 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:

1. x=5,y=9,z=0
2. x=5,y=9,z=8
3. x=4,y=33,z=8
4. 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

b) Code sample 1: x=5 and y=5
   Code sample 2: x=5 and y=5

Exercise 4:

a) `f("importance")["2"].valueLabel()`

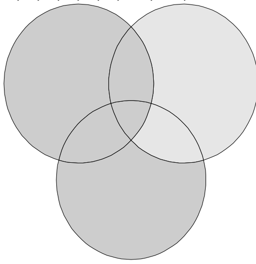
b) `f("importance")["2"].label()`

Exercise 5:

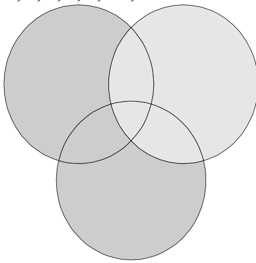
```
if(!f("q2").toBoolean())

 {

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

  for(i=0;i<precodes.length;i++)

    {

     code = precodes[i];

     f("q2")[code].set("3");

    }

 }
```

Exercise 6:

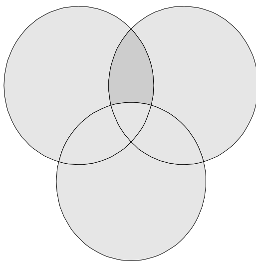
a) 1,2,5,6,7,9,10,11,12



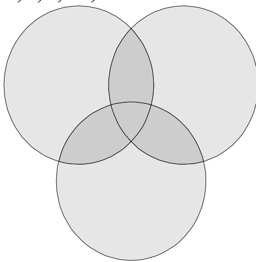
b) 1,2,6,7,9,10,12



c) 11



d) 2,4,5,10,11



Exercise 7:

```
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
 {
  dt = new Date();
  dt.setFullYear(d.year,d.month-1,d.day);
  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:

```
num = f("phone").get();
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:

```
zipcode = f("zipcode").get();
re = /^\d{5}$/;
if(pcode.search(re) == -1)
 {
  RaiseError();
```

```
  SetQuestionErrorMessage(LangIDs.en,"Please provide a valid zip code (digits
only).");

 }
```

APPENDIX B FURTHER READING

For further reading on JScript, we recommend the following sources:

Microsoft Windows Script Technologies:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/script56/html/js56jsoriJScript.asp

James Jaworski: Mastering JavaScript and JScript (Sybex, ISBN 0-7821-2492-5)

ECMAScript Language Specification:
http://www.ecma.ch/ecma1/STAND/ECMA-262.HTM

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 10

Example 2 Filtering a Single Question Based on Answers to a Multi 11

Example 3 Excluding a Column (Question) in a 3D grid 12

Example 4 Piping in the Response to a Single Question 13

Example 5 Password Check 16

Example 6 Setting complete status before the end of the survey 17

Example 7 Removing an Answer in a Single or Grid Question 23

Example 8 Screening on a Numeric Question 27

Example 9 Checking that a Multi Question has been Answered 27

Example 10 Response Piping from a Single Question with Other Specify 33

Example 11 Replacing "NO RESPONSE" in Response Piping 34

Example 12 Using Switch to set Values for each of the Answer Alternatives on a Single 39

Example 13 Validating Sums in a 3D Grid Using a while Loop 55

Example 14 Validating Sums in a 3D Grid Using a do while Loop 59

Example 15 Copying a Multi to do Response Piping with "Other, specify" 60

Example 16 Validating Sums in a 3D Grid Using a for Loop and break 64

Example 17 Calculating Averages in a Grid 65

Example 18 Calculating Averages on a Single Question in a Loop 66

Example 19 Validation of Ranking when the Number of Elements is Optional 69

Example 20 Validating Sums in a 3D Grid with the Sum Function 71

Example 21 Finding the Number of Answers on Three Multi Questions 73

Example 22 Calculating Averages on 3 Numeric Multi Questions in a 3D Grid 74

Example 23 Finding Maximum and Minimum Values on Numeric Multi Questions 75

Example 24 Building a Condition on a Range of Precodes 75

Example 25 Making a Multi Question Required (Generic Code) 76

Example 26 Building a Cryptic URL to be displayed in an info node 77

Example 27 Setting Interview Status Before End of Survey 79

Example 28 Recording the Respondent's Browser Type and Version 80

Example 29 Sending in Values with the URL 81

Example 30 Quota Check 82

Example 31 Presetting a Quota Question to Check Several Quotas 82

Example 32 Checking that a Response in a Multi Open Text Question is an Integer 83

Example 33 Validating Date Format of Open Text Date Question 85

Example 34 Validating Date with Dropdowns for the Date Parts 85

Example 35 Validation of Email Address Format 87

Example 36 Excluding Respondents from Specific Networks 88

Example 37 Send Confirmation Email at the End of a Survey 89

Example 38 Invitation Email to a Different Part of the Same Survey 90

Example 39 Redirect to Another Site Before the End Page 91

Example 40 Function to Make a Multi Question Required 92

Example 41 Function to Copy a Multi Question 93

Example 42 Returning a Calculated Value from a Function 95

Example 43 Referencing a Question in a Loop 102

Example 44 Using a Variable instead of Repeated Calls on the f Function 103

Example 45 Deleting the Content of any Question 105

Example 46 Copying the Contents of any Form into Another 106

Example 47 Filtering an Answer List by the First Characters in the Answer 109

Example 48 Checking the Number of Answers on a Multi Question 110

Example 49 Filtering an Answer List on Items Selected in Two Previous Questions 112

Example 50 Filtering Answers Not Selected in a Previous Question 112

Example 51 Always Including a "Don't know" Answer Alternative 112

Example 52 Joining Answers in one Multi Question 115

Example 53 Using a Function to Filter an Answer List Based on the Answers on a Grid 116

Example 54 Using a Hidden Multi to Filter an Answer List Based on a Grid 117

Example 55 Calculating Time Spent 128

Example 56 Validating Date Format and that it is a Valid Date after Current Date 130

Example 57 Validating that a Date with Dropdowns is within the next two Weeks 131

Example 58 Finding the Weekday 132

Example 59 **Converting Data of Birth into Age (in number of years)** **133**

Example 60 **Rounding to a Number with Two Digits** **136**

Example 61 **Picking n Random Items from the Answers to a Multi Question** **137**

Example 62 **Randomly Assigning which Part of a Survey the Respondents Should Answer** **140**

Example 63 **Checking a User Name and Password where Username is Case Insensitive** **144**

Example 64 **On the Fly Recoding** **144**

Example 65 **Replacing Last Comma with "and" in a Listing of Answers** **146**

Example 66 **Generating an Array from a String with Values** **147**

Example 67 **Validation of the Format of a Phone Number** **159**

Example 68 **Using Regular Expression to Replace Commas With Line Breaks** **161**

Example 69 **Postal Codes in the United Kingdom** **161**

Example 70 **Converting an Array to a String with Line Breaks between Elements** **163**

Example 71 **Validating Grid with "Other, specify" Alternatives** **164**

INDEX

- # -

- *(operator)* 30

-- *(operator)* 30

- *(regular expression syntax)* 152

! *(operator)* 31

!= *(operator)* 31

!== *(operator)* 31

$ *(regular expression syntax)* 150, 154

% *(operator)* 30

%= *(operator)* 32

&& *(operator)* 31

() *(regular expression syntax)* 150, 155

* *(operator)* 30

* *(regular expression syntax)* 150, 154

*= *(operator)* 32

. *(regular expression syntax)* 150

/ *(operator)* 30

/* */ *(comment)* 19

// *(comment)* 19

/= *(operator)* 32

; *(semi colon)* 36, 178

? *(regular expression syntax)* 151, 154

? : *(operator)* 33

?! *(regular expression syntax)* 156

?: *(regular expression syntax)* 156

?= *(regular expression syntax)* 156

[] *(regular expression syntax)* 151, 152

\ *(regular expression syntax)* 151, 157, 158

\- *(regular expression syntax)* 153

\' *(string formatting character)* 22

\$ *(regular expression syntax)* 151

\(*(regular expression syntax)* 151

\) *(regular expression syntax)* 151

* *(regular expression syntax)* 151

\. *(regular expression syntax)* 151

\? *(regular expression syntax)* 151

\[*(regular expression syntax)* 151

\\ *(regular expression syntax)* 151

\\ *(string formatting character)* 22

\" *(string formatting character)* 22

\^ *(regular expression syntax)* 151

\{ *(regular expression syntax)* 151

\| *(regular expression syntax)* 151

\+ *(regular expression syntax)* 151

\b *(regular expression syntax)* 155

\B *(regular expression syntax)* 155

\b *(string formatting character)* 22

\c *(regular expression syntax)* 152

\d *(regular expression syntax)* 158

\D *(regular expression syntax)* 158

\f *(regular expression syntax)* 152

\f *(string formatting character)* 22

\n *(regular expression syntax)* 152

\n *(string formatting character)* 22

\r *(regular expression syntax)* 152

\r *(string formatting character)* 22

\s *(regular expression syntax)* 152
\S *(regular expression syntax)* 152
\t *(regular expression syntax)* 152
\t *(string formatting character)* 22
\u *(regular expression syntax)* 158
\v *(regular expression syntax)* 152
\w *(regular expression syntax)* 158
\W *(regular expression syntax)* 158
\x *(regular expression syntax)* 158
^ *(regular expression syntax)* 151, 153, 154
{ } *(curly brackets)* 38, 177
{} *(regular expression syntax)* 151, 154
| *(regular expression syntax)* 151, 155
|| *(operator)* 31
+ *(operator)* 30, 32
+ *(regular expression syntax)* 150, 154
++ *(operator)* 30
+= *(operator)* 32
< *(operator)* 31
<= *(operator)* 31
= *(operator)* 32
-= *(operator)* 32
== *(operator)* 31
=== *(operator)* 31
> *(operator)* 31
>= *(operator)* 31

- A -

a *(function)* 108
abs *(method)* 140
acos *(method)* 135
add *(method)* 116
addition *(operator)* 30
anchor *(method)* 148
and *(operator)* 31
AppendErrorMessage *(function)* 15
AppendQuestionErrorMessage *(function)* 15
argument ..... 71, 92, 93. *See* argument
  arguments array 95
Array *(object)* 41–43, 162–67
  concat *(method)* 162
  declare 41
  join *(method)* 162
  length *(property)* 41, 42
  methods 162–67
  pop *(method)* 163
  push *(method)* 163
  reverse *(method)* 163
  shift *(method)* 163
  slice *(method)* 167
  sort *(method)* 164
  splice *(method)* 167
  toString *(method)* 163
  unshift *(method)* 163
  valueOf *(method)* 163
asin *(method)* 135
assignment *(statement)* 20, 36
atan *(method)* 135
atan2 *(method)* 135

- B -

big *(method)* 148

binary operator 29
blink *(method)* 148
bold *(method)* 148
Boolean 21
  false 21
  true 21
break *(statement)* 38, 64
BrowserType *(function)* 80
BrowserVersion *(function)* 80

- C -

case .... *See* switch *(statement)*
categories *(method)* 45
categoryLabels *(method)* 45
ceil *(method)* 136
charAt *(method)* 142
charCodeAt *(method)* 142
ClearErrorMessage *(function)* 15
ClearQuestionErrorMessage *(function)* 15
CODED (property) 104
column masks 12
combident 16, 185
comments 19
compound 102
COMPOUND (property) 105
concat *(method)* 147, 162
conditional expression ternary operator 33
conditions 9
constructor 101
continue *(statement)* 65
conversion 25
cos *(method)* 135
Count *(function)* 73
curly brackets 38
CurrentForm *(function)* 76
CurrentID *(function)* 77
CurrentLang *(function)* 77
CurrentPID *(function)* 77
CurrentSID *(function)* 77

- D -

data types 20–25
  Array 41–43, 162–67
  Boolean 21
  conversion 25
  null 23
  number 21
  object 98–167
  string 22
  undefined 25
Date *(object)* 119–34
  Constructors 119
  getDate *(method)* 124
  getDay *(method)* 124
  getFullYear *(method)* 123
  getHours *(method)* 125
  getMilliseconds *(method)* 126
  getMinutes *(method)* 125
  getMonth *(method)* 123
  getSeconds *(method)* 126
  getTime *(method)* 127
  getTimezoneOffset *(method)* 127

getUTCDate *(method)* 124
getUTCDay *(method)* 124
getUTCFullYear *(method)* 123
getUTCHours *(method)* 125
getUTCMilliseconds *(method)* 126
getUTCMinutes *(method)* 125
getUTCMonth *(method)* 123
getUTCSeconds *(method)* 126
getYear *(method)* 123
methods 120–34
parse *(method)* 120
setDate *(method)* 124
setFullYear *(method)* 123
setHours *(method)* 125
setMilliseconds *(method)* 126
setMinutes *(method)* 125
setMonth *(method)* 123
setSeconds *(method)* 126
setTime *(method)* 127
setUTCDate *(method)* 124
setUTCFullYear *(method)* 123
setUTCHours *(method)* 125
setUTCMilliseconds *(method)* 126
setUTCMinutes *(method)* 125
setUTCMonth *(method)* 123
setUTCSeconds *(method)* 126
setYear *(method)* 123
toGMTString *(method)* 128
toLocaleString *(method)* 127
toString *(method)* 127
toUTCString *(method)* 127
UTC *(method)* 121
valueOf *(method)* 127
day *(property)* 129
decimal 21
decimal point 21
declaration *(statement)* 36
decrement *(operator)* 30
DEF_EXCL *(error template)* 171
DEF_EXCL_AND *(error template)* 171
DEF_EXCL_OR *(error template)* 171
DEF_NEXCL *(error template)* 171
DEF_NEXCL_AND *(error template)* 171
DEF_NEXCL_OR *(error template)* 171
DICHOTOMY (property) 105
diff *(method)* 111
division *(operator)* 30
do while *(statement)* 59
domainLabels *(method)* 45
domainValues *(method)* 45

- E -

E *(property)* 134
ECMAScript 9
else ....*See* if *(statement)*
equal *(operator)* 31
ErrorTemplate *(function)* 170–75
DEF_EXCL 171
DEF_EXCL_AND 171
DEF_EXCL_OR 171
DEF_NEXCL 171

DEF_NEXCL_AND 171
DEF_NEXCL_OR 171
MAX_SIZE 173
MISSING 170
MISSING_AND 170
MISSING_OR 170
NUMERIC_ERRORS 174
NUMERIC_ERRORS_AND 174
OTHER 172
PRECISION 174
PRECISION_ERRORS 174
PRECISION_ERRORS_AND 174
RANGE_ERRORS 175
RANGE_ERRORS_AND 175
RANGE_MAX 175
RANGE_MIN 175
RANK_MAX 173
RANK_MIN 173
SCALE 175
SCALE_ERRORS 175
SCALE_ERRORS_AND 175
SEL_EXCL 171
SEL_EXCL_AND 171
SEL_EXCL_OR 171
SEL_NEXCL 171
SEL_NEXCL_AND 171
SEL_NEXCL_OR 171
SPEC 172
TOO_LONG 173
TOO_LONG_AND 173
ExclusivityError *(function)* 169, 171
exec *(method)* 159
exp *(method)* 141
exponent 21
expression 29

- F -

f *(function)* 44–54, 102–7, 109
- categories *(method)* 45
- categoryLabels *(method)* 45
- CODED *(property)* 104
- COMPOUND *(property)* 105
- compund 102
- DICHOTOMY *(property)* 105
- diff *(method)* 111
- domainLabels *(method)* 45
- domainValues *(method)* 45
- function call 102
- get *(method)* 44
- inc *(method)* 109
- isect *(method)* 111
- label *(method)* 44
- members *(method)* 115
- methods 26, 44–54, 107, 109–16
- NUMERIC *(property)* 104
- OPEN *(property)* 104
- properties 104
- set *(method)* 44
- size *(method)* 110
- toBoolean *(method)* 27
- toNumber *(method)* 26

- union *(method)* 110
- value *(method)* 44
- valueLabel *(method)* 44
- values *(method)* 45

f *(function)* 109–16
false 21
Filter *(function)* 109
fixed *(method)* 148
float
- decimal point 21
- exponent 21

float *(number)* 21
floor *(method)* 136
fontcolor *(method)* 148
fontsize *(method)* 148
for *(statement)* 60
form objects *See* f *(function)*
formatting characters *(strings)* 22
Forward *(function)* 79
fromCharCode *(method)* 142
function 71
- a 108
- AppendErrorMessage 15
- AppendQuestionErrorMessage 15
- argument 71, 92, 93
  - arguments array 95
- BrowserType 80
- BrowserVersion 80
- call 71, 93
- ClearErrorMessage 15
- ClearQuestionErrorMessage 15
- Count 73
- CurrentForm 76
- CurrentID 77
- CurrentLang 77
- CurrentPID 77
- CurrentSID 77
- definition 92
- ErrorTemplate 170–75
- ExclusivityError 169, 171
- f 44–54, 102–7, 109–16
- Filter 109
- Forward 79
- GetStatus 79
- global variable 96
- InRange 75
- InRangeExcl 75
- InterviewEnd 128
- InterviewStart 128
- IsDate 84, 129
- IsDateFmt 84, 129
- IsEmail 87
- IsInteger 83
- IsNet 88
- IsNumeric 83
- IsUserNameTaken 83
- local variable 96
- Max 75
- Min 75
- MissingRequiredError 169, 170

naming conventions 176

nnset 108

NotSelectedError............................ 169, 172

NotSpecifiedError 169, 172

nset 108

NumericError 169, 173

parseFloat... 26

parseInt... 26

PrecisionError 169, 174

qf 81

QuestionErrors 169

RaiseError 14

RangeError.................................... 169, 175

RankError...................................... 169, 173

Redirect.. 91

Request... 80

RequestIP 80

return *(statement)* 95

ScaleError 169, 174

SendMail 88

set 108

SetErrorMessage 15

SetQuestionErrorMessage......................... 15

SetStatus... 79

SizeError 169, 172

Sum 71

- G -

get *(method)* 44

getDate *(method)*.. 124

getDay *(method)* 124

getFullYear *(method)* 123

getHours *(method)* 125

getMilliseconds *(method)* 126

getMinutes *(method)* 125

getMonth *(method)* 123

getSeconds *(method)* 126

GetStatus *(function)*......................................79

getTime*(method)* 127

getTimezoneOffset*(method)*........................127

getUTCDate *(method)* 124

getUTCDay *(method)* 124

getUTCFullYear *(method)* 123

getUTCHours *(method)* 125

getUTCMilliseconds *(method)* 126

getUTCMinutes *(method)*...........................125

getUTCMonth *(method)* 123

getUTCSeconds *(method)* 126

getYear *(method)*..123

global variable 96

greater than *(operator)* 31

greater than or equal *(operator)* 31

- H -

hexadecimal..21

hidden question 20

- I -

if *(statement)* 36

inc *(method)*..109

increment *(operator)* 30

indexOf *(method)* 144

infinity
- negative ... 21
- positive ... 21

InRange *(function)* ... 75
InRangeExcl *(function)* ... 75
integer
- decimal ... 21
- hexadecimal ... 21
- octal ... 21

integer *(number)* ... 21
InterviewEnd *(function)* ... 128
InterviewStart *(function)* ... 128
IsDate *(function)* ... 84, 129
- day *(property)* ... 129
- month *(property)* ... 129
- year *(property)* ... 129

IsDateFmt *(function)* ... 84, 129
- day *(property)* ... 129
- month *(property)* ... 129
- year *(property)* ... 129

isect *(method)* ... 111
IsEmail *(function)* ... 87
IsInteger *(function)* ... 83
IsNet *(function)* ... 88
IsNumeric *(function)* ... 83
IsUserNameTaken *(function)* ... 83
italics *(method)* ... 148

- J -

JavaScript ... 9
join *(method)* ... 162

- L -

label *(method)* ... 44
label *(statement)* ... 66
langIDs ... 16, 185
language
- combident ... 16, 185

lastIndexOf *(method)* ... 144
length *(property)* ... 41, 42, 141
less than *(operator)* ... 31
less than or equal *(operator)* ... 31
link *(method)* ... 148
LiveScript ... 9
LN10 *(property)* ... 134
LN2 *(property)* ... 134
local variable ... 96
log *(method)* ... 141
LOG10E *(property)* ... 134
LOG2E *(property)* ... 134
loops ... 55–70

- M -

match *(method)* ... 160
Math *(object)* ... 134–41
- abs *(method)* ... 140
- acos *(method)* ... 135
- asin *(method)* ... 135
- atan *(method)* ... 135
- atan2 *(method)* ... 135
- ceil *(method)* ... 136
- cos *(method)* ... 135

E *(property)* 134
exp *(method)* 141
floor *(method)* 136
LN10 *(property)* 134
LN2 *(property)* 134
log *(method)* 141
LOG10E *(property)* 134
LOG2E *(property)* 134
max *(method)* 140
methods 135–41
min *(method)* 140
PI *(property)* 134
pow *(method)* 141
properties 134
random *(method)* 136
round *(method)* 135
sin *(method)* 135
sqrt *(method)* 141
SQRT1_2 *(property)* 135
SQRT2 *(property)* 135
tan *(method)* 135
Max *(function)* 75
max *(method)* 140
MAX_SIZE *(error template)* 173
members *(method)* 115
method 98, 101
Min *(function)* 75
min *(method)* 140
MISSING *(error template)* 170
MISSING_AND *(error template)* 170
MISSING_OR *(error template)* 170
MissingRequiredError *(function)* 169, 170
modulus *(operator)* 30
month *(property)* 129
multiplication *(operator)* 30

- N -

naming conventions 176
NaN (Not a Number) 21
negative 0 21
negative infinity 21
negative number 21
new *(operator)* 41, 101
nnset *(function)* 108
not *(operator)* 31
not equal *(operator)* 31
NotSelectedError *(function)* 169, 172
NotSpecifiedError *(function)* 169, 172
nset *(function)* 108
null 23
number 21
float 21
decimal point 21
exponent 21
integer 21
decimal 21
hexadecimal 21
octal 21
NaN (Not a Number) 21
negative 21
negative infinity 21

positive and negative 0 21
positive infinity 21
NUMERIC (property) 104
NUMERIC_ERRORS *(error template)* 174
NUMERIC_ERRORS_AND *(error template)* 174
NumericError *(function)* 169, 173

- O -

object 98–167
Array 41–43, 162–67
composition 99
constructor 101
Date 119–34
define 99
encapsulation 99
form objects .... *See* f *(function)*
information hiding 99
inheritance 100
multiple 100
single 100
instance 99, 101
Math 134–41
method 98, 101
modularity 99
polymorphism 100
property 98, 101
RegExp 149–62
Set 108–18
String 141–48, 160–62
type 98
octal 21
OPEN (property) 104
operand 29
operator 29–35
arithmetic 30
addition 30
decrement 30
division 30
increment 30
modulus 30
multiplication 30
subtraction 30
assignment 32
%= *(operator)* 32
*= *(operator)* 32
/= *(operator)* 32
+= *(operator)* 32
= *(operator)* 32
-= *(operator)* 32
binary 29
comparision 31
equal 31
greater than 31
greater than or equal 31
less than 31
less than or equal 31
not equal 31
strictly equal 31
strictly no equal 31
conditional expression ternary 33

logical 31
and...31
not 31
or 31
new...41, 101
precedence 34
string 31
string concatenation 32
unary 29
or *(operator)* 31
OTHER *(error template)* 172

- P -

parameter ....*See* argument
parse *(method)*...120
parseFloat *(function)* 26
parseInt *(function)*..26
PI *(property)* 134
pop *(method)*...163
positive 0...21
positive infinity...21
pow *(method)* 141
PRECISION *(error template)* 174
PRECISION_ERRORS *(error template)*....174
PRECISION_ERRORS_AND *(error template)* 174
PrecisionError *(function)* 169, 174
precode masks................................11, 108–18
property..98, 101
push *(method)* 163

- Q -

qf *(function)*..81
question id 20
QuestionErrors *(function)*............................169

- R -

RaiseError *(function)*....................................14
random *(method)* 136
RANGE_ERRORS *(error template)*...........175
RANGE_ERRORS_AND *(error template)* 175
RANGE_MAX *(error template)* 175
RANGE_MIN *(error template)*..................175
RangeError *(function)*.........................169, 175
RANK_MAX *(error template)* 173
RANK_MIN *(error template)* 173
RankError *(function)* 169, 173
Redirect *(function)*..91
RegExp *(object)*....................................149–62
Constructors 149, 159
exec *(method)* 159
methods 159–60
syntax 149–59
test *(method)* 159
regular expressions..........*See* RegExp *(object)*
remove *(method)* 116
replace *(method)*...160
Request *(function)* 80
RequestIP *(function)*......................................80
response piping...13
return *(statement)* 95

reverse *(method)* 163

round *(method)* 135

- S -

SCALE *(error template)* 175

scale masks 11, 108–18

SCALE_ERRORS *(error template)* 175

SCALE_ERRORS_AND *(error template)* . 175

ScaleError *(function)* 169, 174

script nodes 17

search *(method)* 161

SEL_EXCL *(error template)* 171

SEL_EXCL_AND *(error template)* 171

SEL_EXCL_OR *(error template)* 171

SEL_NEXCL *(error template)* 171

SEL_NEXCL_AND *(error template)* 171

SEL_NEXCL_OR *(error template)* 171

SendMail *(function)* 88

set *(function)* 108

set *(method)* 44

Set *(object)*

  add *(method)* 116

  diff *(method)* 111

  isect *(method)* 111

  members *(method)* 115

  remove *(method)* 116

  size *(method)* 110

  union *(method)* 110

Set *(object)* 108–18

  a *(function)* 108

  Constructor 108

  Filter *(function)* 109

  inc *(method)* 109

  methods 109–17

  nnset *(function)* 108

  nset *(function)* 108

  set *(function)* 108

setDate *(method)* 124

SetErrorMessage *(function)* 15

setFullYear *(method)* 123

setHours *(method)* 125

setMilliseconds *(method)* 126

setMinutes *(method)* 125

setMonth *(method)* 123

SetQuestionErrorMessage *(function)* 15

setSeconds *(method)* 126

SetStatus *(function)* 79

setTime*(method)* 127

setUTCDate *(method)* 124

setUTCFullYear *(method)* 123

setUTCHours *(method)* 125

setUTCMilliseconds *(method)* 126

setUTCMinutes *(method)* 125

setUTCMonth *(method)* 123

setUTCSeconds *(method)* 126

setYear *(method)* 123

shift *(method)* 163

sin *(method)* 135

size *(method)* 110

SizeError *(function)* 169, 172

slice *(method)* 145, 167

small *(method)* 148
sort *(method)* 164
SPEC *(error template)* 172
splice *(method)* 167
split *(method)* 147
sqrt *(method)* 141
SQRT1_2 *(property)* 135
SQRT2 *(property)* 135
statement 36
  assignment 20, 36
  break 38, 64
  continue 65
  declaration 36
  declare array 41
  do while 59
  for 60
  function call 71, 93
  function definition 92
  if 36
  label 66
  loops 55–70
  object instantiation 101
  return 95
  switch 38
  while 55
strictly equal *(operator)* 31
strictly not equal *(operator)* 31
strike *(method)* 148
string 22
  formatting characters 22
String *(object)* 141–48, 160–62
  anchor *(method)* 148
  big *(method)* 148
  blink *(method)* 148
  bold *(method)* 148
  charAt *(method)* 142
  charCodeAt *(method)* 142
  concat *(method)* 147
  constructors 141
  fixed *(method)* 148
  fontcolor *(method)* 148
  fontsize *(method)* 148
  fromCharCode *(method)* 142
  index 142
  indexOf *(method)* 144
  italics *(method)* 148
  lastIndexOf *(method)* 144
  length *(property)* 141
  link *(method)* 148
  match *(method)* 160
  methods 142–48, 160–62
  properties 141
  replace *(method)* 160
  search *(method)* 161
  slice *(method)* 145
  small *(method)* 148
  split *(method)* 147
  strike *(method)* 148
  sub *(method)* 148
  substr *(method)* 145

substring *(method)* 146

sup *(method)* 148

toLowerCase *(method)* 143

toString *(method)* 148

toUpperCase *(method)* 143

valueOf *(method)* 148

string concatenation *(operator)* 32

sub *(method)* 148

substr *(method)* 145

substring *(method)* 146

subtraction *(operator)* 30

Sum *(function)* 71

sup *(method)* 148

switch *(statement)* 38

- T -

tan *(method)* 135

test *(method)* 159

text substitution 13

toBoolean *(method)* 27

toGMTString *(method)* 128

toLocaleString *(method)* 127

toLowerCase *(method)* 143

toNumber *(method)* 26

TOO_LONG *(error template)* 173

TOO_LONG_AND *(error template)* 173

toString *(method)* 26, 127, 148, 163

toUpperCase *(method)* 143

toUTCString *(method)* 127

true 21

types .... *See* data types

- U -

unary operator 29

undefined 25

union *(method)* 110

unshift *(method)* 163

UTC *(method)* 121

UTC (Universal Coordinated Time) 119

- V -

validation code 14, 168–75

value *(method)* 44

valueLabel *(method)* 44

valueOf *(method)* 127, 148, 163

values *(method)* 45

variable 20

global 96

local 96

name 20

naming conventions 176

scope 20

- W -

while *(statement)* 55

- Y -

year *(property)* 129

- Z -

zero 21