
Confirmit
Confirmit
Scripting Manual

This is document revision 3 of the Confirmit Horizons Scripting Manual published in October 2014. The information herein describes Confirmit Horizons Scripting and its features as of Build nr. 4373 New features may be introduced into the product after this date. Go to www.confirmit.com or check “News” on the Customer Extranet for the latest updates.

Table of Contents

What's New in this Revision?

Note: Only the latest changes to this documentation are listed here. Changes made to earlier revisions are listed in the "Changes to the User Documentation" document which can be downloaded from the Confirmit Extranet at https://extranet.confirmit.com.

The following changes have been made in revision 3 of the Confirmit Horizons v18 Scripting Manual:

- A correction is made to the Dynamic Questions section (see Dynamic Questions on page 9 for more information).
- The Flash and Silverlight section (old section 10.1.3.9.) is removed.
- An example is added to the GetRespondentValue... section (see GetRespondentValue and SetRespondentValue on page 80 for more information).
- A check code example is added to the GetAdditionalColumnValue section (see GetAdditionalColumnValue on page 92 for more information).
- The Writing the Custom Scheduling Script Code section is added to the CATI and CATI Specific Functions section (see Writing a Custom Scheduling Script Code on page 105 for more information).
- A note is added to the Presetting a Quote Question to Check Several Quotas example (see qf on page 109 for more information).

Note: The general layout and language in this document is continually being corrected, adjusted and improved to ensure the user has the best possible source of information. Only NEW information and details of functionality that has changed since the previous revision are listed here - minor corrections to the text and document layout are not listed.

Important
We need your feedback so we can improve this document and provide you with the information you require. If you have any comments or constructive criticism concerning the content or layout of this documentation, please send an email to documentation@confirmit.com. Please include in your email the section number and/or heading text of the section to which your comment applies.

1. Introduction

Most Confirmit questionnaires contain a quantity of script code. Scripts are used:

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

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

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

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

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

1.1. JScript .NET Fast Mode

To allow JScript.NET to execute with optimal performance, the JScript .NET in Confirmit surveys will be compiled in *fast* mode. Since fast mode places some restrictions on the type of code allowed, programs can be more efficient and execute faster. However, some features are not available in fast mode.

In fast mode, the following JScript behaviors are triggered:

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

Most of these changes should not affect scripts used in Confirmit. For example, for security reasons the `eval` method is not even allowed in Confirmit surveys. More details on those requirements that are most relevant when working within Confirmit scripts, follow.

1.1.1. All Variables Must Be Declared.

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

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

you must now declare the variables with the `var` keyword:

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

This will still not explicitly declare the type of the variable: It may change type later as a result of an assignment. In addition to this "loose typing", JScript .NET can now be a strongly typed language. JScript .NET provides more flexibility than previous versions of JScript by allowing variables to be type annotated. This binds a variable to a particular data type, and the variable can store only data of that type. Although type annotation is not required, using it helps prevent errors associated with accidentally storing the wrong data in a variable and can increase program execution speed (see Types, Variables and Constants on page 14 for more information).

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

1.1.2. Functions Become Constants

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

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

1.1.3. The arguments Object is not Available

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

In fast mode, the arguments object is not available. However, JScript .NET allows function declarations to specify a parameter array in the function parameter list. This allows the function to accept an arbitrary number of arguments, thus replacing part of the functionality of the arguments object (see Functions with a Variable Number of Arguments on page 136 for more information).

Note: It is not possible to directly access and reference the current function or calling function in fast mode.

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

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

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

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

2. Where is Scripting Used in Confirmit?

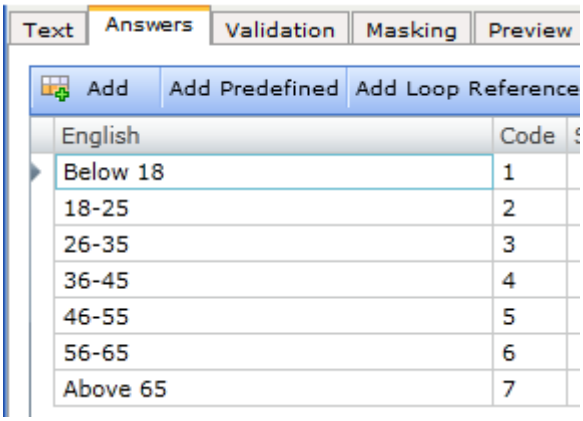
Scripting is used in several places in Confirmit. See the following sections for further details.

2.1. Conditions

In conditions you place a 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.

Screening Based on a Single Question

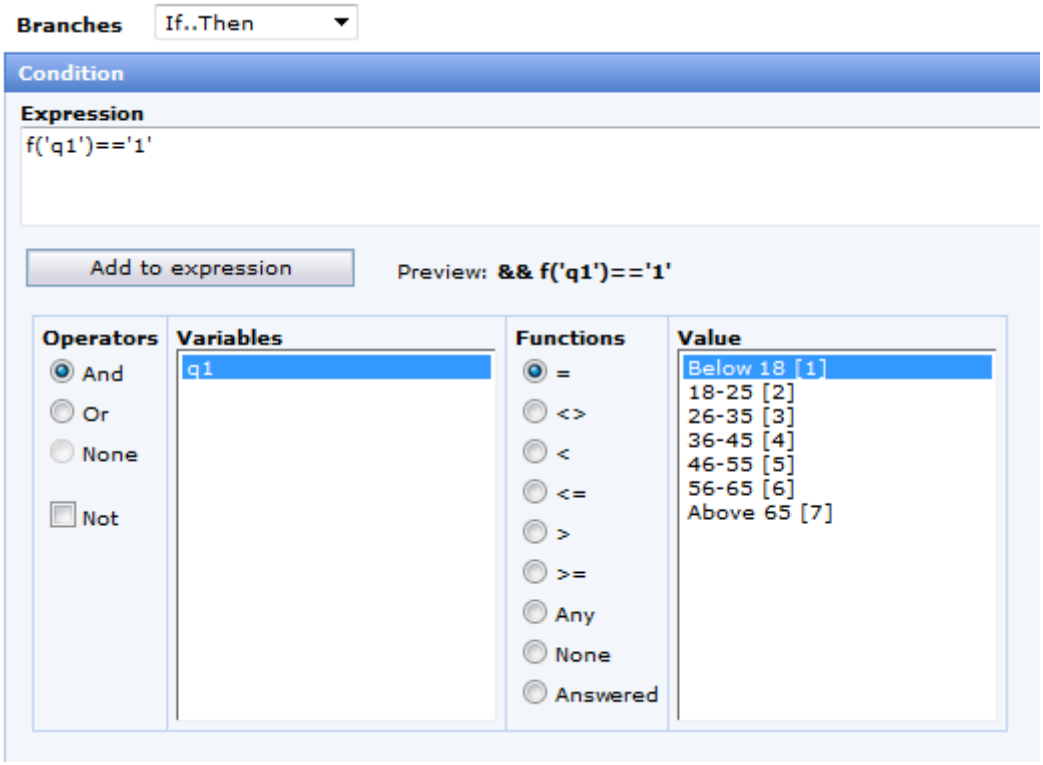
Assume the questionnaire has an age question (single), and you wish to screen respondents below the age of 18 from doing the rest of the interview. The age question has the following answer list:



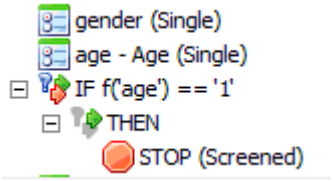
To screen respondents that answer "Below 18", insert a condition after the single question with the following syntax:A

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

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



The routing will look like this:



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

2.2. Filtering Answer Lists, Scales and Loops

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

- Masks based on sets of codes, which are used in "code masks" of single, multi, ranking, open text list, numerical list, grid, 3D grid questions and loops and "scale masks" of grids.
- "Column masks", which are used in 3D grids to filter the columns and are based on true/false expressions, just like conditions.

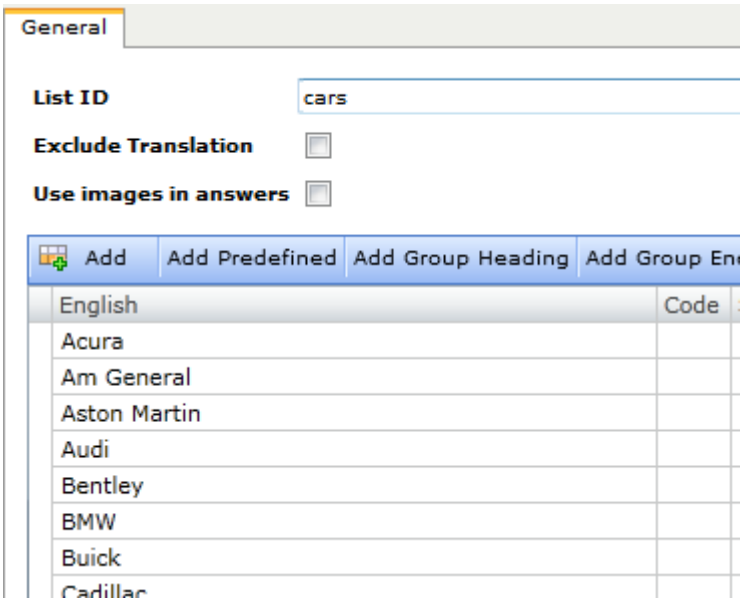
2.2.1. Code Masks and Scale Masks

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

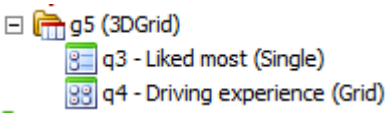
Scale masks are used to filter scale lists of grid questions. Use the scale mask field in the properties of a grid question to enter a JScript .NET expression that evaluates to a set of codes. The scale list will be filtered based on this set of codes.

Filtering a Single Question Based on Answers to a Multi

Assume you have a questionnaire that uses a list called "Cars".



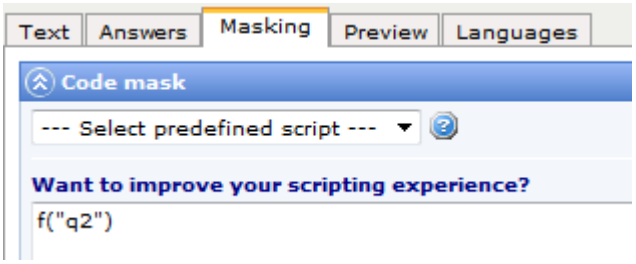
The list is used in two questions: First you have a multi question q2 which asks what cars you have tested, and then a 3D grid question g5 with a single question q3 which asks what car you did like most of the ones you tested, and a grid question q4 which asks you to rate the driving experience of the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



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

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

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



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

2.2.2. Column and Question Masks

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

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

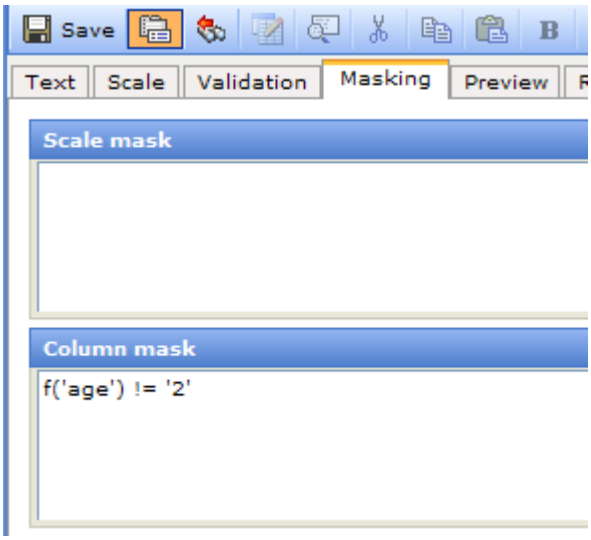
Excluding a Column (Question) in a 3D Grid

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

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

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

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



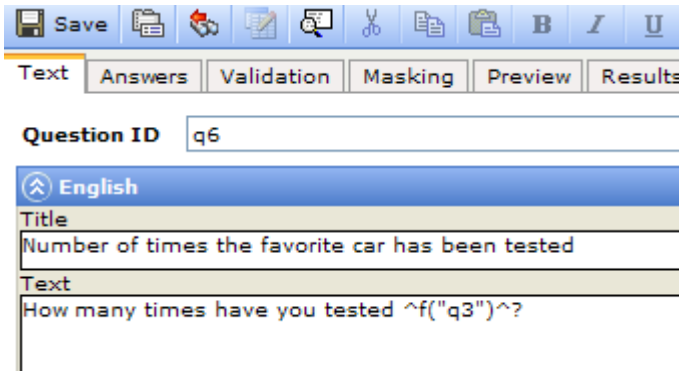
2.3. Text Substitution/Response Piping

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

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 as follows:

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



2.4. Validation Code

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

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

A validation code follows a pattern similar to this:

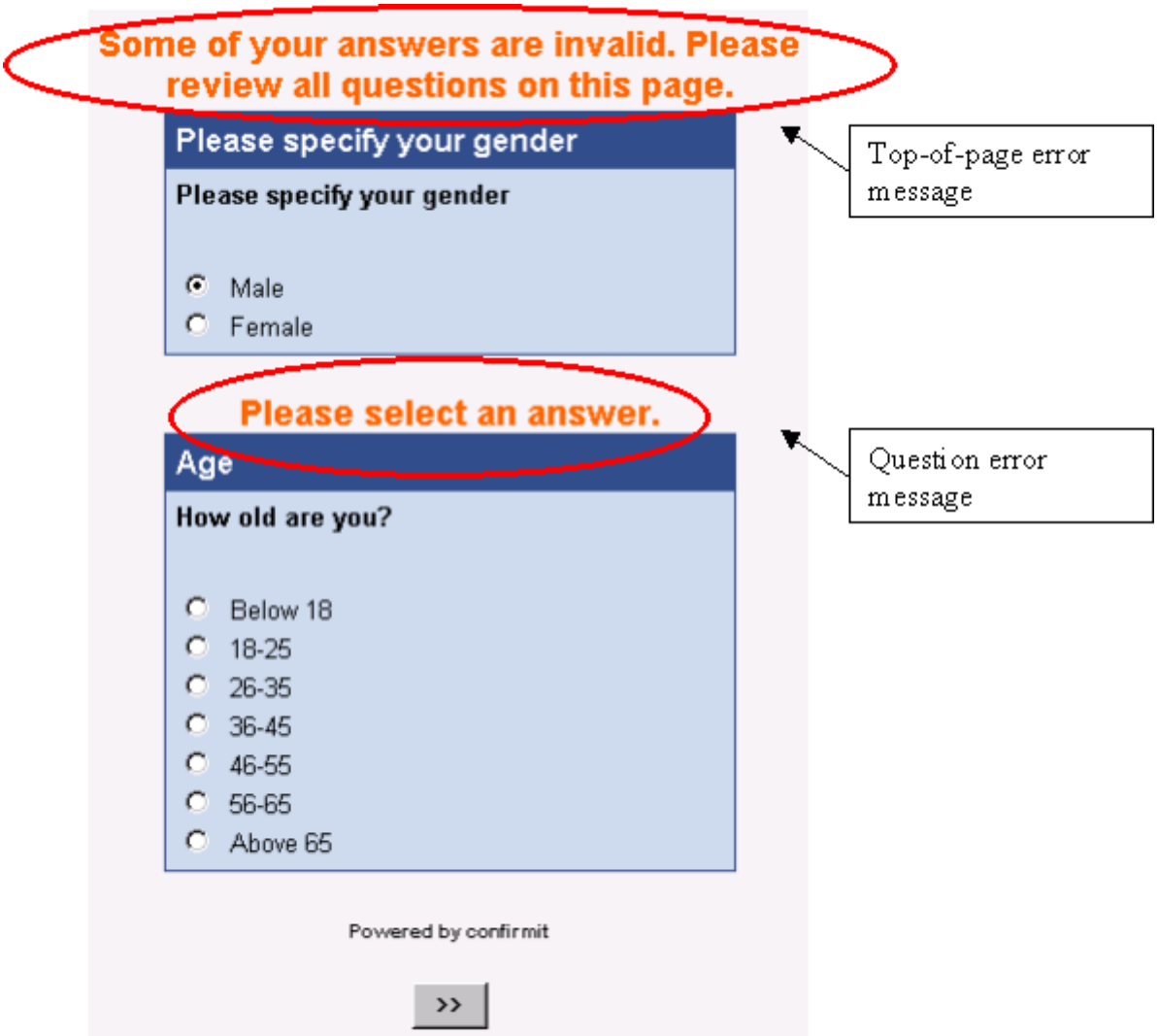
```
if(expression) {   RaiseError();   <some function(s) setting the text
of the error message(s)> }
```

We will get back to this later; so don't worry if you don't understand all of it now. The term's expression and function will be explained later (see Operators and Expressions on page 25 for more information) and (see Arrays on page 37 for more information).

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

```
LangIDs.en
```

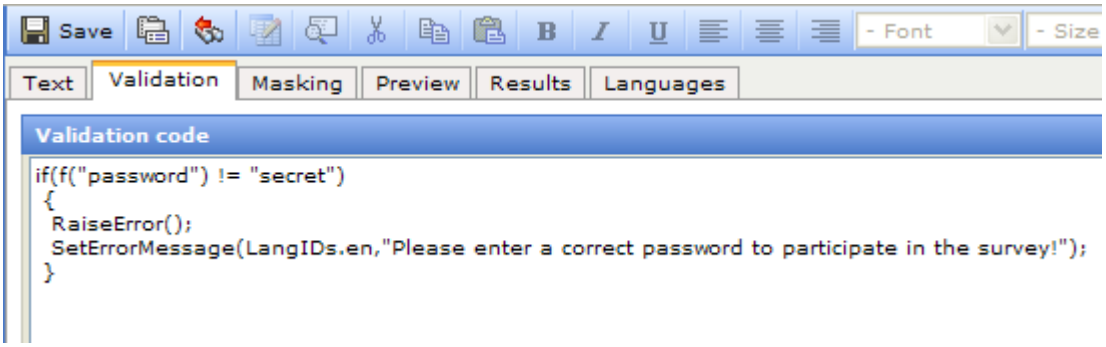
instructs the interview engine to use this error message when the language is English. These language codes, known as combidents, are listed in Appendix C.

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 that shown below in the validation code field of the open question (in this example the question ID is password):

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



2.5. Script Nodes

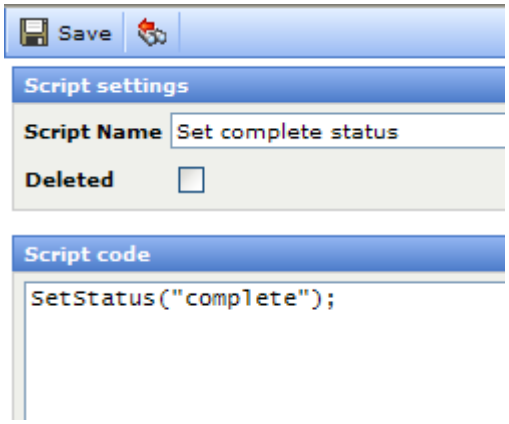
Script nodes typically contain code for:

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

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 using the SetStatus function (see GetStatus and SetStatus on page 81 for more information), in a script node:

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



2.6. Dynamic Questions

This functionality allows you to use logic within a page, and enables parts of a page to be updated based on responses to one or more questions on the same page. This can enable you to, for example, cause additional questions to appear on the page once the respondent has answered the first question. The functionality is based on the Ajax technology, and uses one (or more) questions as a trigger to activate logic in subsequent questions.

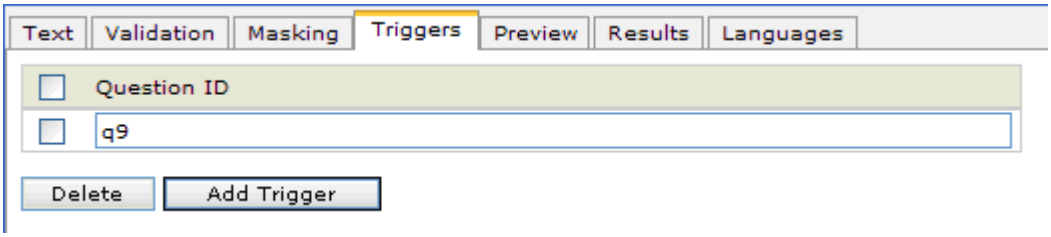
Note: Dynamic Questions only functions within page objects, and the trigger question(s) and the question(s) to be triggered, must be on the same page in the survey.

The following questionnaire logic can be used within pages and will update based on the triggers:

- Text substitution/response piping.
- Question masking.
- Column masking.
- Code masking.
- Scale masking.

Inside the question that you want to be dependent of responses to one or more questions on the same page, you can specify those other questions as “triggers”. This means that for every change to one of the trigger questions, the question will be updated with questionnaire logic like for example masking being reevaluated.

Note that this is only supported within the “Page” object, so the “Triggers” tab will only be available on questions within a page.



Note: The Dynamic Questions functionality is not supported in the CAPI/Kiosk console or debug station. Dynamic Questions is supported in IE8+ (Windows), Firefox 10+ Google Chrome 26+ and Safari 5.1+ (Mac only). The functionality is only available in surveys using Survey Layouts.
Unlike Advanced WI features, Dynamic Questions have no automatic alternative way of presenting the page if Dynamic Questions is not supported. A function DynamicQuestionsEnabled (see DynamicQuestionsEnabled on page 75 for more information) is provided to either screen or provide an alternative routing to respondents whose browser does not support Dynamic Questions.

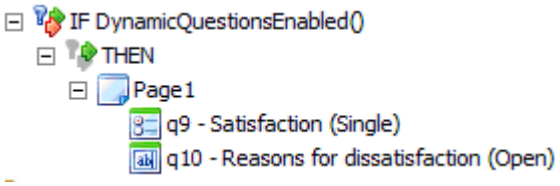
Displaying a Dynamic Follow-Up Question in the same Page

If for example you want a rating question to be followed by an open text question if a negative response has been given, you can use the Dynamic Questions functionality to have that appear on the same page when the negative response is given.

If you place a single question q9 asking people to respond on a 5-point scale, and you have an open text question q10 that should be displayed when q9 is answered with “1” or “2”, you can use a question mask as shown below on the open text question:

```
f("q9")=="1" || f("q9")=="2"
```

If you want these questions to appear on the same page, you can place them inside a “Page” object as shown below:



If you set “q9” as a trigger for q10, then q10 will be displayed automatically when a response is given on q9.

The DynamicQuestionsEnabled() function in the condition is used to ensure the page is only displayed to respondents who are answering in a browser that supports the functionality.

2.7. Script Execution when Respondent Moves Backwards

Script code will be executed both when the respondent moves forwards and backwards through the survey (except for validation code scripts - validation code is only executed when going forwards in the questionnaire).

This is important to bear in mind if your survey has a **Back** button. If for example you include a SendMail script, you might not want an email to be triggered both when the respondent goes forwards and backwards. If your scripts are used to set hidden variables based on certain conditions, you may also consider whether you need to remove the values that were set when going backwards, instead of resetting them. There is a function called "Forward()" (see Forward on page 82 for more information) which can be used for this purpose, to distinguish between actions taken when respondents move forwards and backwards.

2.8. The Syntax Highlighter

The Syntax Highlighter functionality means that while scripting you no longer need to remember the functions, or look up which properties belong to which classes and which parameters the various methods accept. Instead, all these are available at the touch of a button. The highlighter automatically color-codes key words, and provides lists of selectable options under specific conditions while scripting.

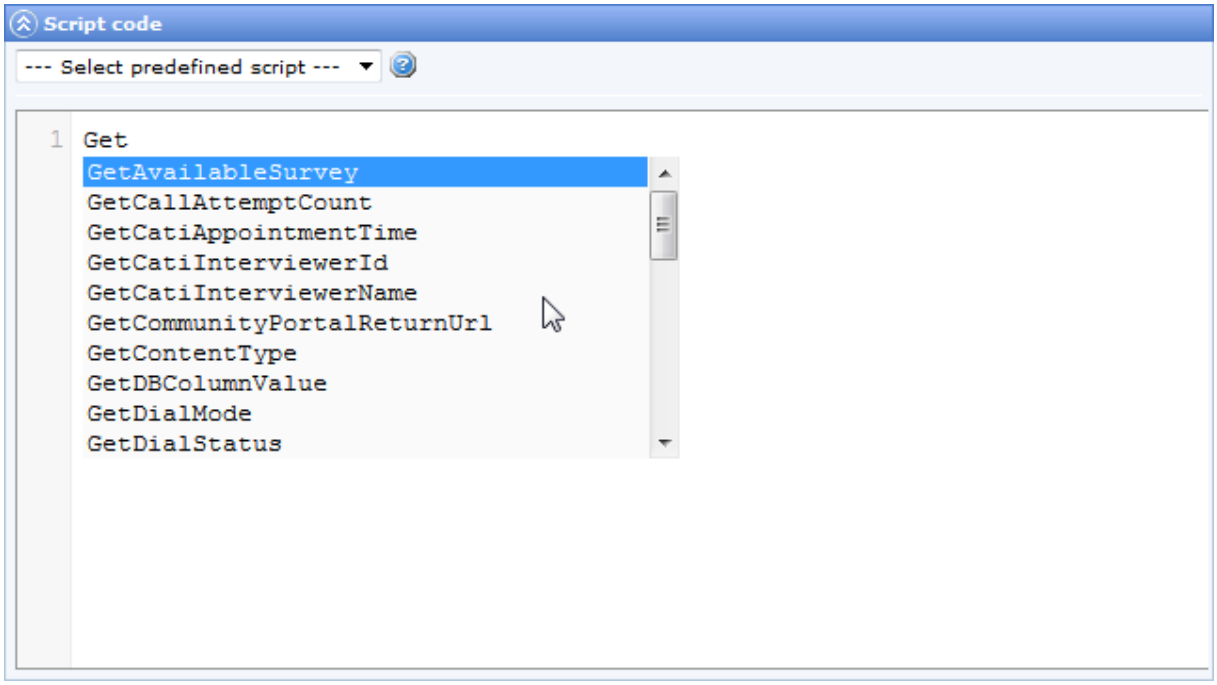
The Syntax Highlighter functionality is on by default for a survey, but you can switch it off in the User Settings.

The following script editing areas support the highlighter capabilities:

- Script nodes in authoring
- Validation and masking of questions
- Reportal scripting
- Data Processing scripting
- JavaScript editors

To use the highlighter, start typing into the scripting area the function you wish to use, then press the **Ctrl**+**Space** keys on your keyboard. A drop-down list of all the functions corresponding to the text string you have typed, opens. To open a complete list of all the available functions, press **Ctrl**+**Space** without first typing anything into the scripting area.

The figure below shows the Syntax Highlighter in action:



2.8.1. Using the Syntax Highlighter

When writing code, you can automatically get a list displayed with classes, functions, methods or properties relevant to the context you are in, covering both Confirmit-specific and general JScript.NET constructions. To display the list:

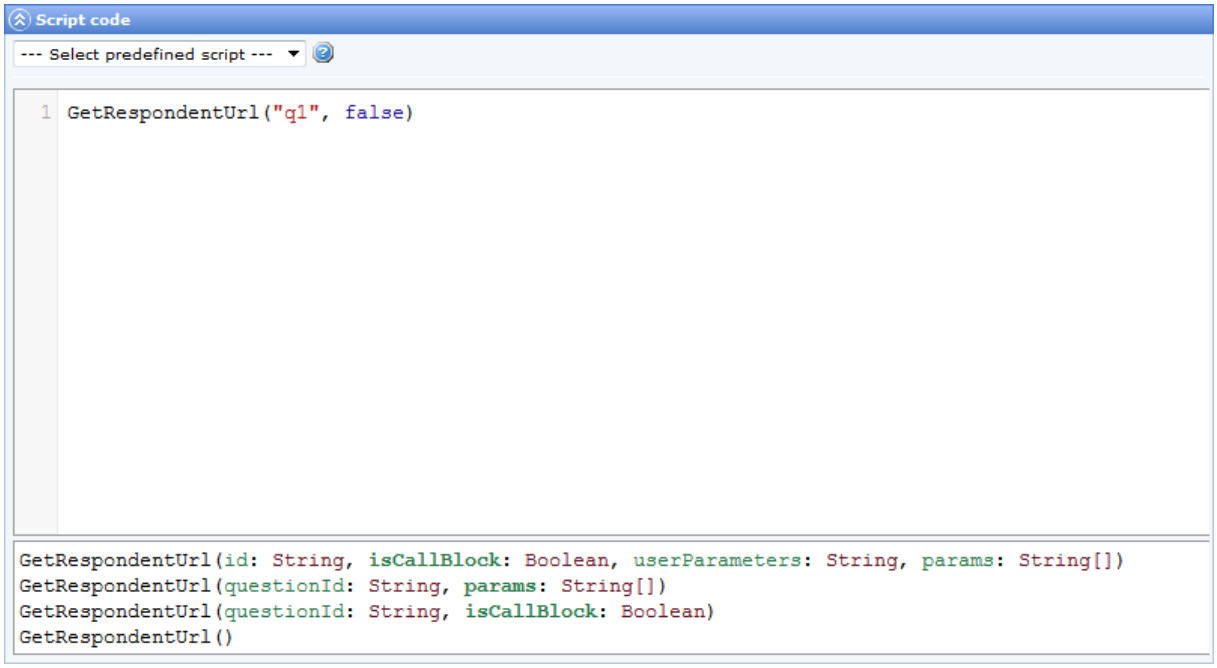
1. Press **CTRL+space** to display the list.

   If you have already started typing, the list will be filtered according to the typed string. After the name of a class or variable, the list will appear automatically when you type period (.).

2. Type more characters to reduce the number of items in the list and jump directly to a member in the list. Use the arrow buttons to move up and down in the list.

3. Press **Enter** to select an item.

On functions and methods, a parameters list will automatically pop up to give you information about the number, names and types of parameters required by a function, template, or attribute. The parameter in bold indicates the next parameter that is required as you type the function. Where there are different versions of the function with different parameter lists, you can select which parameter list you wish to view.



To view parameter information:

1. After the name of a function or method, type an open parenthesis (as you normally would) to open the Parameters list.

   The declaration for the function will pop up under the insertion point. The first parameter in the list appears in bold. To switch among functions, use the UP or DOWN arrow keys. As you type the function parameters, the bold changes to reflect the next parameter that you need to enter.

2. Press **ESC** at any time to close the list.

Use **Tab** to indent lines of code. Lines of code within curly brackets, { and }, will automatically be indented. To indent several lines of code in one operation, mark all the code you want to indent and press **Tab**. To un-indent, press **Shift+Tab**.

2.8.2. Syntax Highlighter Limitations

The Syntax Highlighter has some limitations. No autocompletions are available for implicitly typed objects (objects returned by functions etc.) so

Typing:

```
f()
```

will not result in an autocomplete suggestion.

Typing:

```
var d = new Date();
d.
```

will not result in an autocomplete suggestion.

Typing:

```
var myDate: Date;
myDate.
```

will result in an autocomplete suggestion.

3. Comments

Comments are text added in your scripts that are ignored when the script is run, but may be used to explain aspects of the code. In JScript .NET you can add comments in two ways:

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

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

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

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

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

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

It is recommended that you add a lot of comments in your scripts, to explain to yourself and to others what your script is supposed to do and what it can be used for. However, as you may later want to comment out large parts of your scripts, including comments, and it is not possible to nest comments, it is recommended that you always use the single line comments, as shown below:

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

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

4. Types, Variables and Constants

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

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

4.1. Naming

Variable and constant names in JScript .NET and **question IDs** in Confirmit begin with an upper- or lowercase letter (a-z, A-Z) or an underscore (_) and continue with letters (a-z, A-Z), digits (0-9) or underscore (_).

Examples of variable names:

```
countermakeMoreMoneycar123_tempiThinkThisIsReallyBoring
```

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

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

There are some reserved words that cannot be used as question IDs in Confirmit and some that cannot be used as variable or constant names in JScript .NET. See appendix C in the Confirmit User Manual and a JScript .NET reference manual. In addition, you cannot use names of functions (either the Confirmit provided functions described in this manual, or functions you define yourself in script nodes (see Functions on page 68 for more information).

4.2. Data Declaration

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

```
var counter : int;
```

This will declare a variable `counter` to be of type integer (see Null on page 15 for more information). Here it is not given an initial value, and will assume the default value for integers which is 0. You can also assign an initial value to it like this:

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

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

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

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

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

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

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

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

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

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

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

```
var counter;
```

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

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

```
var counter = 0;
```

Untyped constants are defined in the same way:

```
const maxSelected = 100;
```

4.3. Undefined Values

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

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

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

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

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

4.4. Null

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

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

Removing an Answer in a Single or Grid Question

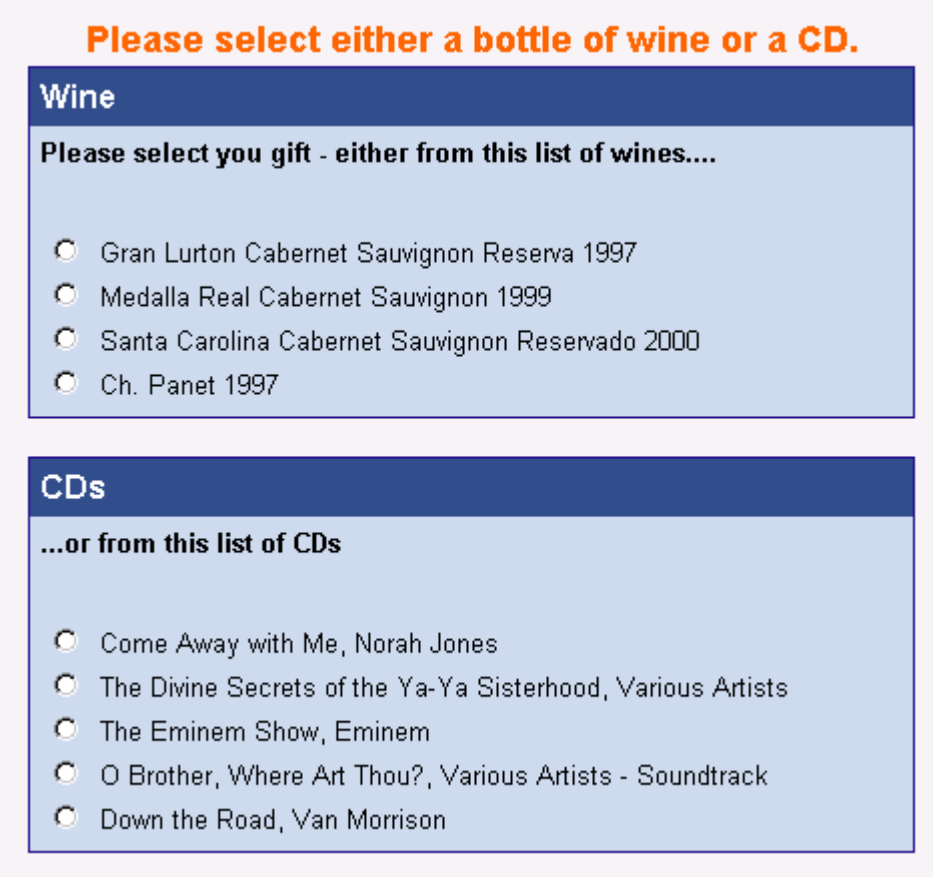
You can use `null` to "remove" an answer to a question with a radio button (i.e. single or grid questions). Say you have a page with two single questions, present1 and present2, at the end of the survey. The respondent should answer just one of the questions, where the respondent can choose between two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

You must 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 (as in the figure above), but the respondent cannot de-select the answers since this is not possible with radio-buttons. Then you must 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 (see toBoolean on page 23 for more information). `set` is used to set the value of a question (see get and set on page 40 for more information).

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



4.5. Types

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

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

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

4.5.1. Numeric

There are two main types of numeric data in JScript:

- Integers.
- Floating-point data.

4.5.1.1. Integers

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

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

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

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

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

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

4.5.1.2. Floating-point Data

Floating-point values are whole numbers with a decimal portion. They can either be represented with digits followed by a decimal point. (“period”/”dot”) and more digits (e.g. `1.29384`), or they can be expressed in scientific notation; that is, an uppercase or lowercase letter e is used to represent "times ten to the power of" (e.g. `7.64e3`). A number that begins with a single 0 and contains a decimal point is interpreted as a decimal floating-point literal and not an octal literal (see Integers on page 17 for more information).

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

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

JScript supports the following floating-point data types:

JScript value type	Range
`float`(single-precision floating-point)	The `float` type can represent numbers as large as 10^{38} (positive or negative) with an accuracy of about seven digits, and as small as 10^{-44}. The `float` type can also represent `NaN` (Not a Number), positive and negative infinity, and positive and negative zero. This type is useful for large numbers where you do not need precise accuracy. If you require very accurate numbers, consider using the `Decimal` data type.

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

4.5.2. Boolean

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

```
true
```

and

```
false
```

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

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

4.5.3. Characters and Strings

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

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

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

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

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

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

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

For apostrophe, use \':

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

Below is a table describing the special formatting characters:

Character	Meaning
\'	Single quote

\"	Double quote
\\	Backslash
\n	New line
\r	Carriage return
\t	Horizontal tab
\b	Backspace

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

There is a significant difference between the string "123" and the number 123. The number 2 will be a smaller value than the number 123. But the string "2" is larger than the string "123", because the first character in the string is compared first, and the character 2 is larger than the character 1 (the same system as is used in alphabetical listings such as dictionaries and telephone directories). All values are stored as strings in Confirmit questions, so you must convert them to numbers if you want them to be evaluated as numbers. See the next section in the manual.

4.5.3.1. Unicode

Jscript.NET supports Unicode characters in strings. They can be used like any other characters, but note that the encoding of the survey pages will be set according to the settings for the active language, so that for example for English, which by default will be set to have encoding “Western European (ISO)”, Unicode characters will not be displayed correctly on the survey page. Only Unicode languages will display Unicode characters correctly. You should also ensure you set the right codepage (Unicode - 65001) if using Unicode characters in SendMail and SendMailMultiPart scripts.

4.6. Conversion between Types/Conversion Functions

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

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

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

Explicit conversion is done by using the data type identifier (see Types on page 17 for more information). To explicitly convert an expression to a particular data type, use the data type identifier followed by the expression to convert in parentheses. Explicit conversions require more typing than implicit conversions, but you can be more certain of the result.

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

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

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

d = double(s);

Now the variable d holds the double 1234. This type of conversion is called *narrowing* since there are a lot of possible string values that cannot be converted to double (e.g. the string “Confirmit”). Explicit narrowing conversions will usually work, but with loss of information. The string “Confirmit” converted to a double will give `NaN`(not a number). But some types are incompatible and will throw an error, and for some values there is not sensible value to convert to.

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

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

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

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

```
d = s;
```

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

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

4.6.1. Conversion Methods in JScript .NET

4.6.1.1. parseInt

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

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

parses the string as an integer of base *radix*. *radix* is optional and is a value between 2 and 36 indicating the base of the number contained in *string*. If not supplied, strings with a prefix of '0x' are considered hexadecimal and strings with a prefix of '0' are considered octal. All other strings are considered decimal... Usually you want numbers to be treated as decimals, and since a leading zero would indicate that the number is an octal number (see Numeric on page 17 for more information), it is a good idea to always include the base 10 when using `parseInt`.

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

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

returns 123

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

returns `NaN` (not a number) since the first character is not a number.

4.6.1.2. parseFloat

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

```
parseFloat(numString)
```

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

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

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

4.6.1.3. isNaN

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

```
isNaN(num)
```

num is a numeric value to test.

4.6.1.4. isFinite

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

```
isFinite(num)
```

num is a numeric value to test.

4.6.1.5. toString

`toString` is a method of the Object object, which means it is available in any other JScript .NET object. It is used to convert a variable into a string. Sometimes this is needed on a variable before inserting it in a Confirmit question with the `set` method (see get and set on page 40 for more information).

4.6.1.6. valueOf

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

4.6.2. Conversion Methods in Confirmit

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

4.6.2.1. toNumber

`toNumber` is a method you can use to convert the value of a question from a string into a Number (double) (see Floating-point Data on page 18 for more information):

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

qID is the question ID.

Note: The toNumber method requires that the question is answered, and that the value stored for the question can be converted to a number. If not, it will return NaN (not a number).

4.6.2.2. toInt

`toInt` is a method you can use to convert the value of a question from a string into an integer (see Integers on page 17 for more information):

```
f(qID).toInt()
```

qID is the question ID.

Note: The toInt method requires that the question is answered, and that the value stored for the question can be converted to an integer. If not, it will fail, terminate the interview with an "Internal server error" and send an error report to the "Email address to receive emails triggered by scripting errors in interview". The method should either only be used when you know that there will always be a valid response to the question, or you should check that the response is numeric before using toInt (for example using IsInteger).

4.6.2.3. toDecimal

`toDecimal` is a method you can use to convert the value of a question from a string into a decimal (see Floating-point Data on page 18 for more information):

```
f(qID).toDecimal()
```

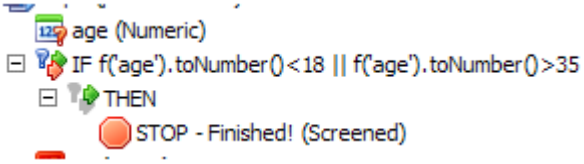
qID is the question ID.

Note: The toDecimal method requires that the question is answered, and that the value stored for the question can be converted to a Decimal. If not, it will fail, terminate the interview with an "Internal server error" and send an error report to the "Email address to receive emails triggered by scripting errors in interview". The method should either only be used when you know that there will always be a valid response to the question, or you should check that the response is numeric before using toDecimal (for example using IsNumeric).

Screening on a Numeric Question

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

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



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

4.6.2.4. toBoolean

`toBoolean` is a method that converts the variable into a Boolean (`true`/`false`). It can be used to check if a question has been answered, or for a single question with the Boolean property, if it has been answered with the true or false answer alternative.

Note: The Boolean question property mentioned in the last sentence above is only available in surveys using the Optimized Database format.

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

If the respondent has answered the question with question ID `qID`, the expression will yield `true`, if not, false. Or, if qID is a Boolean question, the expression will yield true if the respondent has chosen the answer corresponding to true (1), and false if the respondent has chosen the answer corresponding to false (0)

You have already seen `toBoolean` used in the Removing an Answer in a Single or Grid Question example.

4.6.2.5. toDate

toDate can be used on date questions to return a .NET structure representing the selected date value. Note that this is **not** the same as a JScript Date object. JScript supports the use of .NET structures, but not the declaration of new ones, so sometimes it may be easier to create a JScript Date object instead for scripting purposes. See The Date Object (chapter 14.1).

```
f(qID).toDate()
```

returns a strongly-typed Nullable DateTime object representing the date answered.

4.6.2.6. day

day can be used on date questions to get the day part of the specified date.

```
f(qID).day()
```

returns the integer value (1 - 31) representing the "day" part of the specified date. 0 is returned if no date is specified.

4.6.2.7. month

month can be used on date questions to get the month part of the specified date.

```
f(qID).month()
```

returns the integer value representing month part of the date specified, 0 if no date is specified. Conventional numbering of months is used (1 = January, 2 = February etc.).

4.6.2.8. year

year can be used on date questions to get the year part of the specified date.

```
f(qID).year()
```

returns the integer value representing year part of the date specified, 0 if no date is specified.

4.6.2.9. datestring

datestring can be used on date questions to get a textual representation of the specified date.

```
f(qID).datestring()
```

returns the textual representation of the chosen date in accordance to the current language setting (for example, English gives "Friday, April 25, 2010"). The same result will be achieved (a textual representation of the date in accordance with the current language setting) when performing response piping on a date question.

Note: The date question type is only available in surveys using the "Optimized database format".

5. Operators and Expressions

5.1. Terminology

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

For example, in the expression

```
2 + 3
```

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

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

```
x = 0;
```

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

```
2 + 3
```

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

```
2*3
```

is 6, the same is

```
3*2
```

Other operators give different results for different orderings:

```
3-2
```

is 1.

```
2-3
```

is -1.

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

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

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

5.2. Arithmetic Operators

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

Some of these may need some explanation:

Modulus gives the remainder of dividing two integers. Examples:

```
7%2
```

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

```
6%2
```

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

++ and --:

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

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

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

Result: x = 4 and y = 4

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

Result: x = 4 and y = 3

In the first example, x is increased by 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 by 1. We recommend that you be very careful when using these operators.

5.3. Logical Operators

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

5.4. Comparison Operators

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

>	Greater than
>=	Greater than or equal

Expressions with comparison operators evaluate to true or false.

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

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

5.5. String Operators

Operator	Description
+	String concatenation

Strings can be joined with the string concatenation operator +:

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

will return

```
"Confirmit"
```

5.6. Assignment Operators

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

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

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

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

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

will result in x being equal to "Confirmit".

5.7. new

`new` is used to create new objects. Its use is described in Arrays, Objects, Working with Sets and Predefined JScript .NET Objects.

5.8. The Conditional Expression Ternary Operator

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

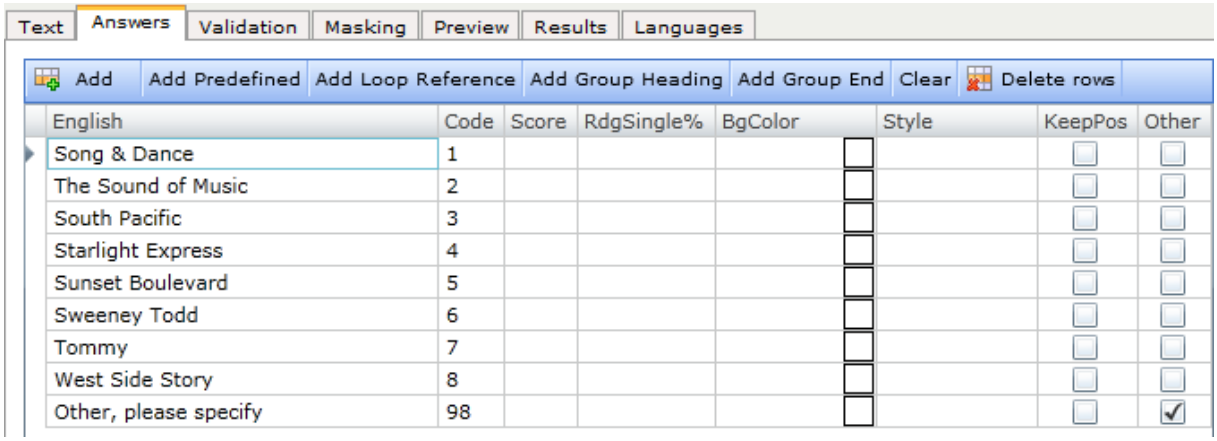
```
condition ? value1 : value2
```

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

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 code 98 that has the "Other" property set.

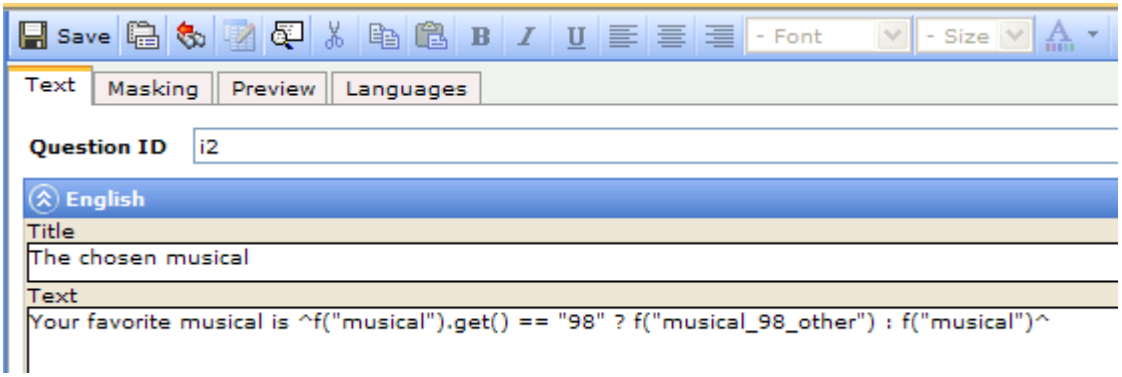


English	Code	Score	RdgSingle%	BgColor	Style	KeepPos	Other
Song & Dance	1					☐	☐
The Sound of Music	2					☐	☐
South Pacific	3					☐	☐
Starlight Express	4					☐	☐
Sunset Boulevard	5					☐	☐
Sweeney Todd	6					☐	☐
Tommy	7					☐	☐
West Side Story	8					☐	☐
Other, please specify	98					☐	☑

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

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

`f("musical").get()` will return the code of the answer to musical (see get and set on page 40 for more information). 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.



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:

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.

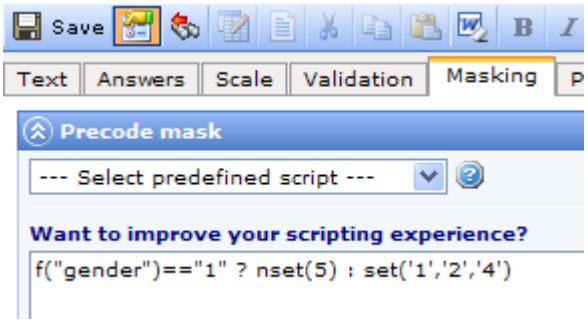
Note: Expressions inside ^ (carets) in question text fields, title fields and answer/scale lists in Confirmit must always return strings. You cannot run JScript .NET statements inside ^'s, however you can call a function that returns a string.

Conditional Code Masking

It is also possible to use this operator in a code mask so a different code mask can be used dependent on a condition. For example, if you want males (code 1) to be given different answer alternatives than women (code 2), you can base the code mask on the "gender" question like this:

```
f("gender")=="1" ? nset(5) : set('1','2','4')
```

If the condition is evaluated as true (=male) then a code mask of 1-5 would be used; if the condition is evaluated as false (=female) then a code mask of 1, 2 and 4 would be used.



5.9. Coercion

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

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

```
"Confirmit"+8.5
```

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

In JScript :NET you can also force what type a value should be coerced to by using the target type name (see Types on page 17 for more information).

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

row + column	string "12.34"	int 123	float .123	logical true	logical false	null
string "test"	test12.34	test123	test0.123	testtrue	testfalse	testnull
int 123	12312.34	246	123.123	124	123	123
float .123	0.12312.34	123.123	0.246	1.123	0.123	0.123
logical true	true12.34	124	1.123	2	1	1
logical false	false12.34	123	0.123	1	0	0
null	null12.34	123	0.123	1	0	0

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

Exercise 1:

Find the result of this expression.

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

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

See the answer in APPENDIX A Answers to Exercises.

5.10. Operator Precedence

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

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

	Operator
1	()
2	!, -- , ++, -, new
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 the answers in APPENDIX A Answers to Exercises.

5.11. Short Circuit Evaluation

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

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

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

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

This is particularly helpful in expressions involving function calls.

6. Simple Statements

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

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

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

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

A statement may be written over multiple lines of text. However, do not use line break inside string constants as this will give a script error. If you need a line break inside a string, you must use `\n` (see Characters and Strings on page 19 for more information). Also, postfix increment and decrement operators must appear on the same line as their argument (e.g. `i++` and `i--`, (see Arithmetic Operators on page 25 for more information), `continue` and `break` keywords must appear on the same line as their label (see The break Statement on page 63 for more information), and `return` must appear on the same line as its expression (see The return Statement on page 137 for more information).

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

6.1. Declarations

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

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

or

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

as for example in

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

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

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

```
var x = 12;
```

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

```
var x;
```

6.2. Assignment Statements

```
variableName  assignmentOperator  expression;
```

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

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

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

6.3. The if Statement

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

6.3.1. if

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

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

Note that the parentheses around the condition are required.

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

We have used if conditions several times already. See the examples in Validation Code, Null and toBoolean for more information.

6.3.2. if-else

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

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

Exercise 3:

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

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


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

…when x and y are declared as

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

…when x and y are declared as

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

See the answers in APPENDIX A Answers to Exercises.

6.3.3. Using Curly Brackets in if Statements

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

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

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

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

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

6.4. The switch Statement

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

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

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

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

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

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

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

```
if(expression == value1)
 {  statements1 }
...else if(expression == valuen)
 {
  statementsn
 }
else
 {
  statementsx
 }
```

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 numeric 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 first using if and the second using switch. As you see, switch gives more compact code that is easier to read:

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


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

7. Arrays

Arrays are a special type of JavaScript object. An object is referred to as a complex data type because it is built from primitive types (see The Array Object on page 186 for more information).

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

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

7.1. Typed Arrays

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

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

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

7.1.1. Declaring Typed Arrays

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

The data type of the elements in a typed array is defined with the name of the data type (see Types on page 17 for more information) followed by square brackets (`[]`).

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

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

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

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

This can also be done in one operation:

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

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

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

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

7.2. JScript Arrays

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

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

7.2.1. Declaring JScript Arrays

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

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

```
var weekday =

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

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

```
var weekday = new

Array("Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","S
unday");
```

You can also define an empty array:

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

or an array with size 7:

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

and then assign values to the elements:

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

The difference between declaring the array as a JScript array instead of as a typed array (see Declaring Typed Arrays on page 37 for more information) is that when it is defined as a typed array, all elements in the array must be of that type.

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

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

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

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

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

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

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

7.3. length

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

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

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

The array could have been declared without specifying length:

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

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

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

7.4. The length Property

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

We will get back to objects in general in Objects, and also the methods supported by the array object (see The Array Object on page 186 for more information).

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

```
objectName.propertyName
```

The length of an array is determined as follows:

```
arrayName.length
```

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

```
weekday.length
```

would return 7.

8. Methods of the Form Objects

The `f` function used 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, modify them etc. For each of the methods described, examples are given in Applying the Methods on Different Types of Questions.

See also Confirmit's f Function for further details.

8.1. get and set

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

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

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

will return the value stored in the database for that object. For an **open text** question this will be the answer the respondent has typed in the text box, for a **single** question this will be the code 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 code value that is set, in this case the answer with code "1". An open text question will be set to the value "1" (string) with this statement.

For **multi** questions and other question types with several answers (grid, ranking, open text list and numeric list questions), *get* is not supported. You can use get on individual elements of these question types though (see Referencing the Elements of a Multi, Ranking, Open Text List, Numeric List or Grid on page 53 for more information). *set* can be used to set multi, grid, ranking, open text list and numeric list questions. For multi questions the method will take the codes of the answers that should be set as input as a set, an array or as a dictionary.

```
f("q1").set(["1","3","4"]);
```

would set answers with code 1, 3 and 4 in a multi q1 as "selected", and all other answers as "not selected".

It is also possible to set a multi based on the responses to another multi:

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

For grid, ranking, numeric list and open text list, the method will take a dictionary as input, with a collection of key/value pairs (the code of the answer and the value you want to set).

For **date** questions, get will return the chosen date formatted as YYYY-MM-DD, or an empty string if no date is set. set can either take a string as parameter (format: YYYY-MM-DD) or a JScript Date object (for example f("q1").set(new Date()) to set current date).

Note: The Date question type is only available when using the Optimized Database format.

8.2. label

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

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

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

will return the title of the question q1.

8.3. text

```
f(qID).text()
```

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

```
f("q1").text()
```

will return the question text of the question q1.

8.4. instruction

```
f(qID).instruction()
```

This method is used to access the question instruction of a question. This method is available on **open text**, **single, multi, numeric, date, ranking, open text list, numeric list**and **grid** questions.

```
f("q1").instruction()
```

will return the question instruction of the question q1.

8.5. value

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

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

8.6. valueLabel

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

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

8.7. domainValues

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

This will return an array with all codes from the answer list of a **single**, **multi, ranking, open text list, numeric list** or **grid** question, headers excluded.

This method is subject to masking, so if a code mask is used to filter the answer list then this will only return the codes of the answers that are displayed to the respondent.

The method is also subject to randomization. That is, if the answer list is randomized, then the array will be returned randomized.

For hierarchy and table lookup questions, only the first 500 (server setting: WIMaxExternalLookupEntries) hits are returned. If the limit is exceeded, an error is returned.

8.8. domainLabels

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

This method will give an array with all the labels (answer texts) on the question, headers excluded. This is the corresponding answer texts to the codes 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, ranking, open text list, numeric list** or **grid** question. This is also subject to masking. For hierarchy and table lookup questions, only the first 500 (server setting: WIMaxExternalLookupEntries) hits are returned. If the limit is exceeded, an error is returned.

The method is also subject to randomization. That is, if the answer list is randomized, then the array will be returned randomized.

8.9. categories

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

`categories` will return an array with the codes of the items that have been answered on a **multi, ranking, open text list, numeric list**or **grid** question. For a multi question, this will be the codes of the answers that have been selected, for a grid this will be the codes 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.

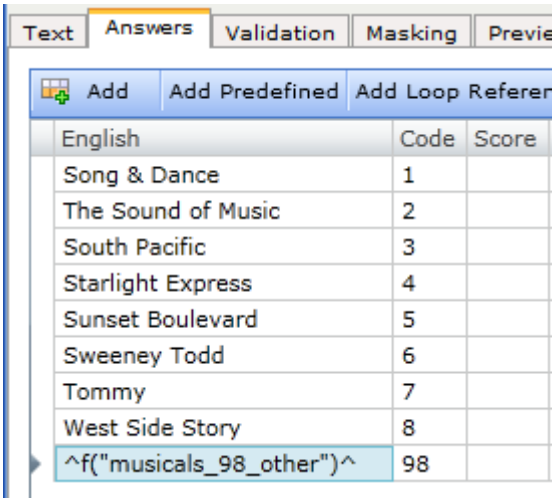
Copying a Multi to do Response Piping with "Other, specify"

In the example in The Conditional Expression Ternary Operator, we saw how you could pipe in the answer in the text box of an "other, specify" option on a single question. However, you cannot use the same approach on a multi question because there "other, specify" can be answered in combination with any of the other alternatives.

A way of solving this for a multi question is to copy the answer of 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 code 98 and the "Other" property set to yes, you can set up a second multi question *musicals_hidden* with the "hidden" property, where the answer from the "other, specify" text box is piped in with the syntax

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

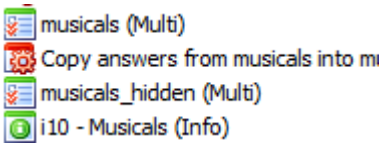


Then you need a script node to copy the response from *musicals* into *musicals_hidden*. Copying answers from one question to another hidden question can also be something you need to do for example to have different, shorter answer texts in response piping in a later question, or to have different texts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

```
f("musicals_hidden").set(f("musicals").categories());
```

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 codes).

The script is to be placed after the musicals question:

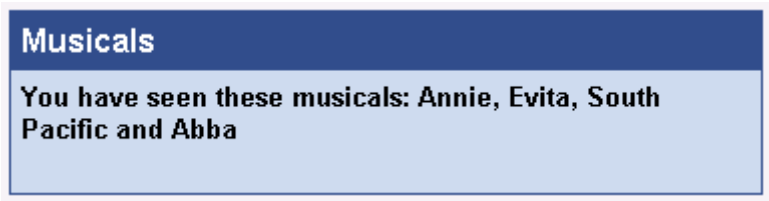


In the info node you may now refer to the *musicals_hidden* question in your response piping:

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

If the respondent for example answers like this:

the result of the piping will be:



8.10. categoryLabels

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

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

8.11. values

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

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

8.12. getType

```
f(qID).getType()
```

getType will return the type of the question. For an open text question it will return "OPEN", for a numeric question it will return "NUMERIC", for single question "CODED", for multi, ranking, open text list, numeric list and grid questions "COMPOUND", for date "DATE", for Boolean (property on single question) "BOOL" and "EXTERNAL" for a question referring to an external answer list in database designer.

The getType method can be used to determine the type of question in a general purpose script for example for validation.

8.13. any

```
f(qID).any(code1,…,coden)
```

`any` will return true if any of the codes listed is answered on a **single** or **multi** question, or if any of the items for these codes have a response on a **grid, multi, ranking, open text list** or **numeric list** question, or false if none of these items has a response. The method is for example useful in conditions.

8.14. all

```
f(qID).all(code1,…,coden)
```

`all` will return true if all of the codes listed is answered on a **multi** question, or if all of the items for these codes has a response on a **grid, multi, ranking, open text list** or **numeric list** question, or false if none of these items has a response. The method is for example useful in conditions.

8.15. none

```
f(qID).none(code1,…,coden)
```

`none` will return true if none of the codes listed is answered on a **single** or **multi** question, or if none of the items for these codes has a response on a **grid**, **multi**, **ranking**, **open text list** or **numeric list** question, or false if either of these items has a response. The method is for example useful in conditions.

8.16. between

```
f(qID).between(from, to)
```

`between` will return true if the answer to a **numeric** question or an item of a **numeric list** question is between the from value and the to value, or false if not. This means the value answered is higher than or equal to the from value and lower than or equal to the to value. The method is for example useful in conditions.

8.17. isNearBy

```
f(qID).isNearBy(latitude, longitude, distance)
```

isNearBy can be used on Geolocation questions, and will return true if the position recorded in the question is within a given distance in meters from the position defined by the latitude and longitude. Latitude, longitude and distance are all numeric (double).

8.18. latitude and logitude

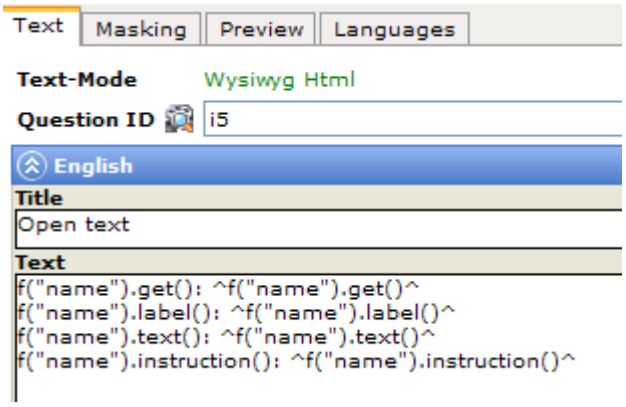
```
f(qID).latitude()

f(qID).longitude()
```

Latitude and longitude can be used on Geolocation questions, and will return the latitude and longitude of the position recorded in the question.

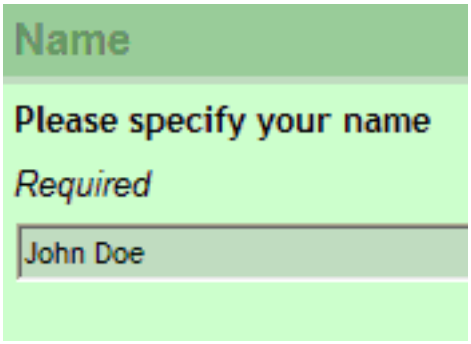
8.19. Applying the Methods on Different Types of Questions

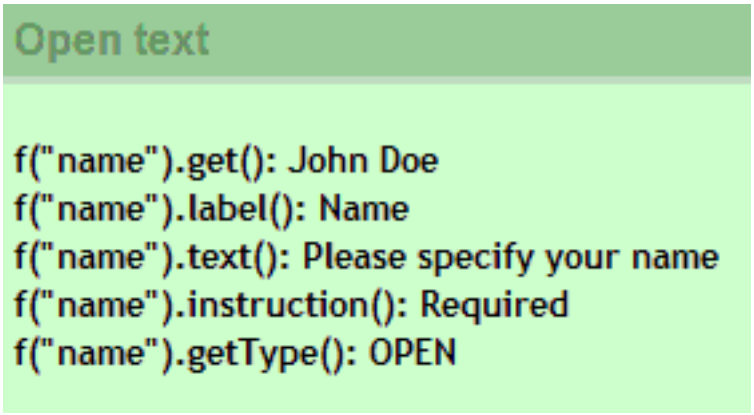
The examples below show these methods applied on an open text, date, single, single with Boolean property set, multi, open text list, and grid question. Each question is followed by an info node that is set up with response piping using the different methods.



8.19.1. Open Text Question

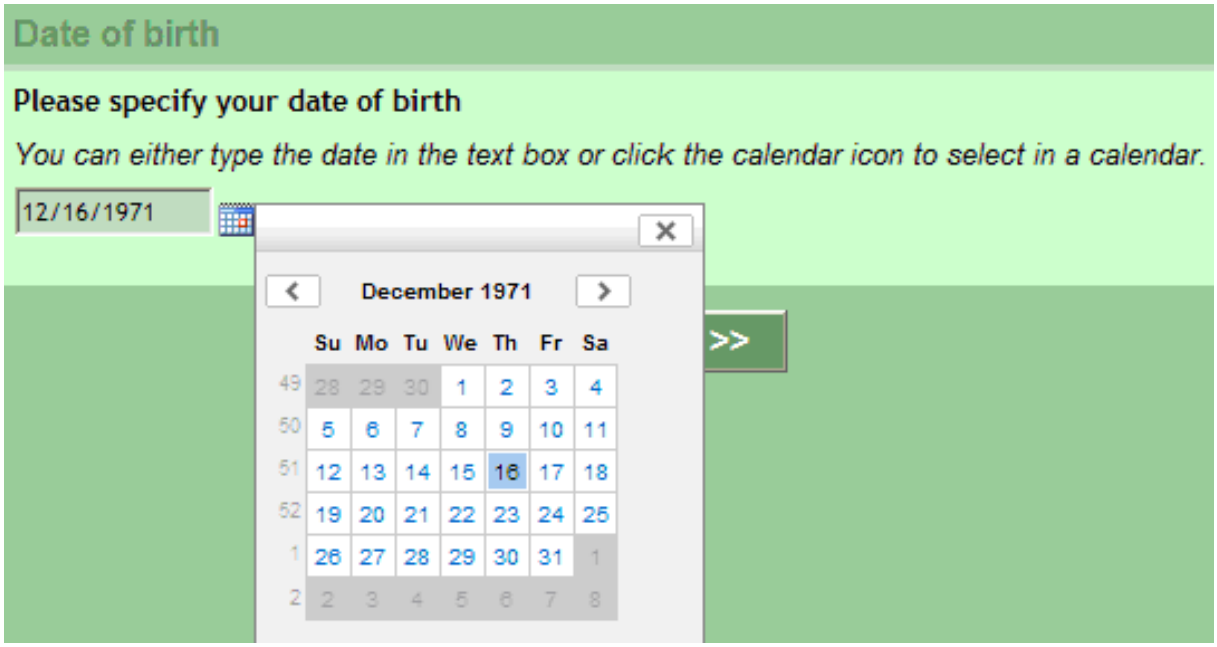
Question ID: "name".

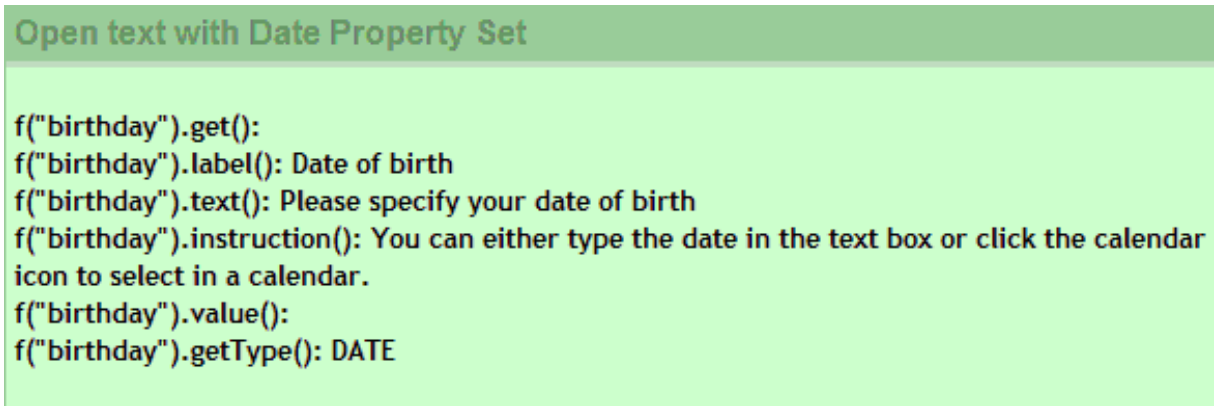




8.19.2. Date

Question ID: "birthday".



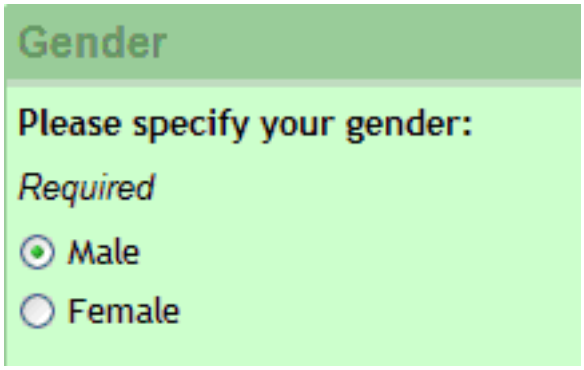


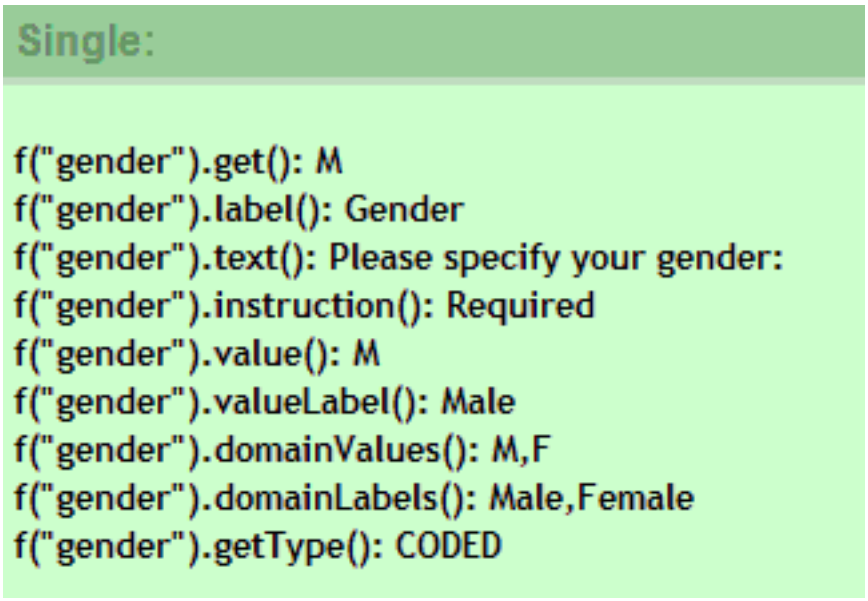
The date format will depend on survey language.

Note: The Date question property is only available when using the Optimized Database format.

8.19.3. Single Question

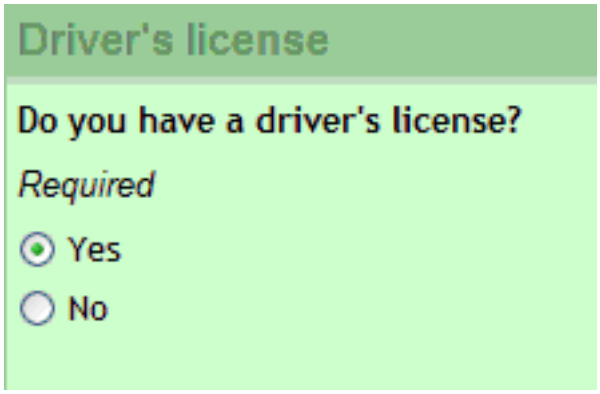
Question ID: "gender". Codes: M (Male) and F (Female).

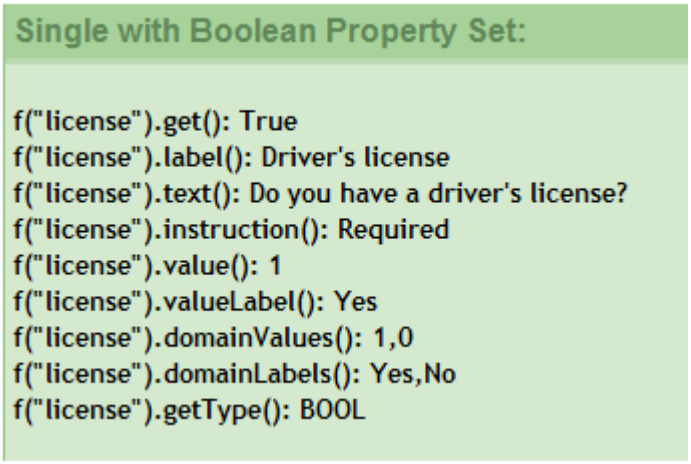




8.19.4. Single Question with Boolean Property Set

Question ID: "license". Single questions with the Boolean property set will be fixed to having two answers in the answerlists with code 1 (true) and 0 (false).

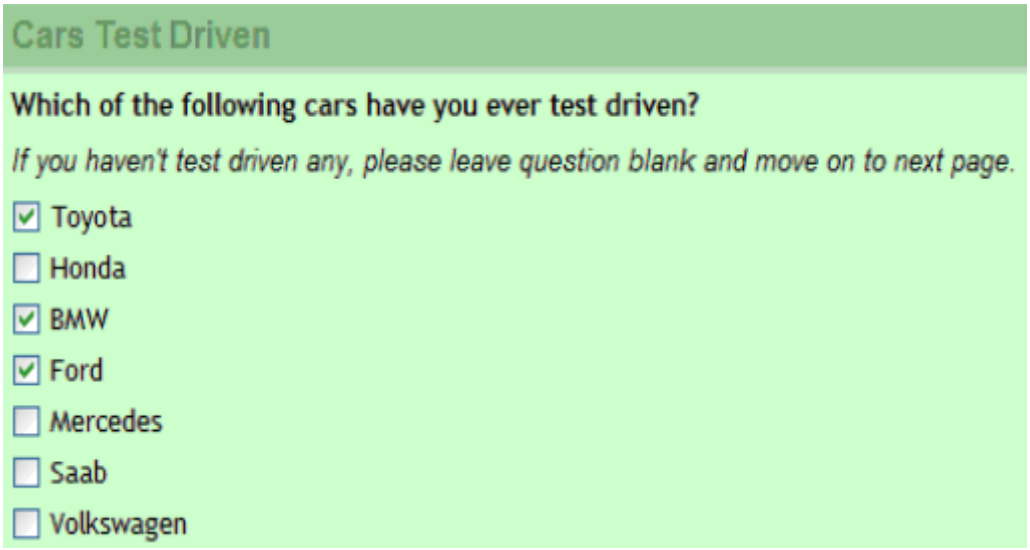




Note: The Boolean question property is only available when using the Optimized Database format.

8.19.5. Multi Question

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



Multi

f("cars").label(): Cars Test Driven
f("cars").text(): Which of the following cars have you ever test driven?
f("cars").instruction(): If you haven't test driven any, please leave question blank and move on to next page.
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
f("cars").getType(): COMPOUND

8.19.6. Open Text List

Question ID: "carscharacteristics". Codes: "1","2","3","4","5","6","7".

Characteristics

What are the top of mind characteristics of these different car brands.

Please type in the first characteristics that you can think of for each car. You may leave the text field blank for cars you do not want to comment.

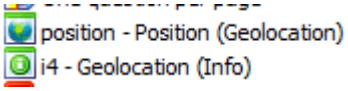
Toyota	Environmental friendly
Honda	Reliable
BMW	Innovative
Ford	Smart
Mercedes	Quality
Saab	Safety
Volkswagen	Value for money

Multi open text

f("carscharacteristics").label(): Characteristics
f("carscharacteristics").text(): What are the top of mind characteristics of these different car brands.
f("carscharacteristics").instruction(): Please type in the first characteristics that you can think of for each car. You may leave the text field blank for cars you do not want to comment.
f("carscharacteristics").values(): Environmental friendly,Reliable,Innovative,Smart,Quality,Safety,Value for money
f("carscharacteristics").categories(): 1,2,3,4,5,6,7
f("carscharacteristics").categoryLabels(): Toyota,Honda,BMW,Ford,Mercedes,Saab,Volkswagen
f("carscharacteristics").domainValues(): 1,2,3,4,5,6,7
f("carscharacteristics").domainLabels(): Toyota,Honda,BMW,Ford,Mercedes,Saab,Volkswagen
f("carscharacteristics").getType(): COMPOUND

8.19.7. Geolocation Question

Question ID: "position"



Geolocation

f("position").toBoolean: True
f("position").get(): 59.9213025 10.6767571
f("position").label(): Position
f("position").text(): The respondent's position
f("position").value(): 59.9213025 10.6767571
f("position").getType(): GEO
f("position").isNearBy(57.6301, 39.865631, 2000000): True
f("position").latitude(): 59.9213025
f("position").longitude(): 10.6767571

8.19.8. Grid Question

Question ID: "importance". Codes in answer list: "1","2","3","4","5","6","7". Code 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.

Required. Please rate the importantce on a scale from not important to very important.

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

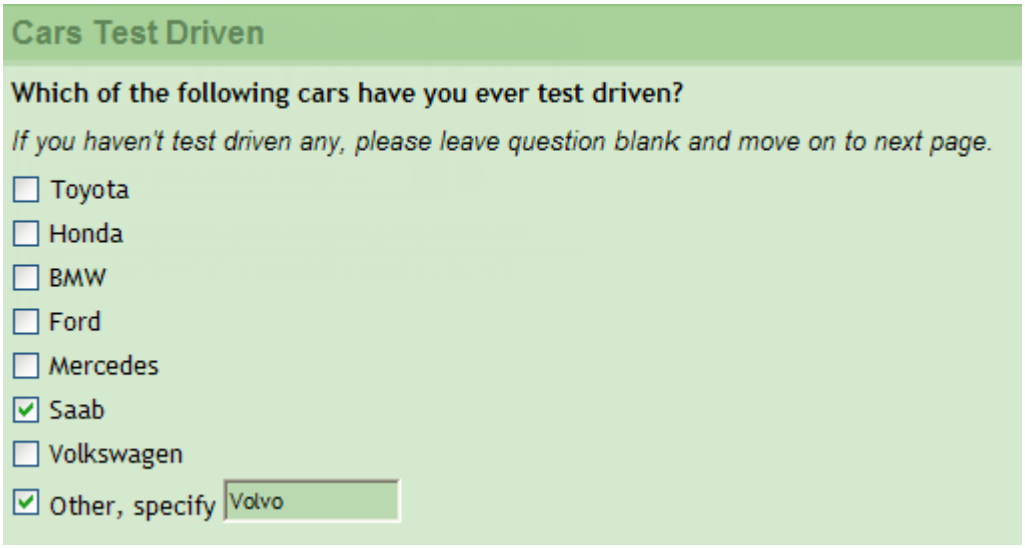
Grid

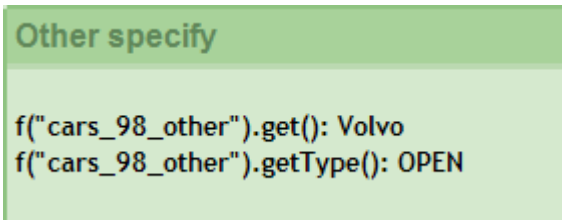
f("importance").label(): Areas of Importance
f("importance").text(): Please indicate how important the following areas are to you when you are considering purchasing a new car.
f("importance").instruction(): Required. Please rate the importantce on a scale from not important to very important.
f("importance").values(): 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
f("importance").getType(): COMPOUND

8.19.9. Other Specify Items

Single, Multi, Numeric Lists, Open Text Lists, Ranking, Grid and 3D Grid questions can all have elements in the answer list with the "Other" property set. This means the answer alternative will be presented with a text box such that the respondent can specify an "other" answer alternative.

You refer to this response with the id qID_code_other, for example q1_98_other for the other text box for answer alternative with code 98 on question q1. Here, for the other, specify alternative on a multi question cars:





8.19.10. Referencing the Elements of a Multi, Ranking, Open Text List, Numeric List or Grid

Each item in the answer list of a multi, ranking, open text list, numeric list or 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 code of the element. The code is a string. For example

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

will be the item with code "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`, `label` and `getType` for the elements of a **multi, ranking, open text list** or **numeric list** question. `label` will then give the label of the answer in the current language. getType will return "DICHOTOMY" if it is a multi, "NUMERIC" if it is a numeric list, and "OPEN" if it is any of the other types.
- `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 code 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 codes from the scale and the corresponding labels (answers) from the scale in the current language. getType will return "CODED".

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

8.19.10.1. 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
f("importance")["3"].getType(): CODED
```

8.19.10.2. Element of a Multi Question

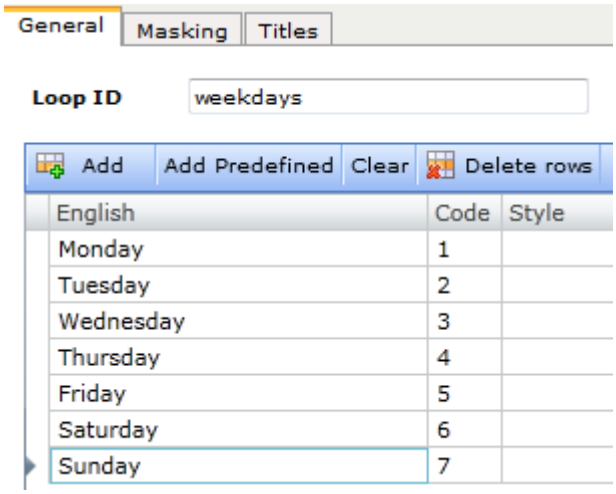
```
Multi:
f("cars")["3"].get(): 1
f("cars")["3"].label(): BMW
f("cars")["3"].getType(): DICHOTOMY
```

8.19.10.3. Element of an Open Text List Question

```
Multi open text:
f("carscharacteristics")["3"].get(): Innovative
f("carscharacteristics")["3"].label(): BMW
f("carscharacteristics")["3"].getType(): OPEN
```

8.19.11. 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).



Below 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
f("weekdays").getType(): CODED
```

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 also be used outside of the loop.

8.19.12. 3D Grid

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

An other specify item on a 3D grid can be accessed in a similar way to other specify on other questions:

```
f("qid_code_other").get()
```

qid would then be the question id of the 3D grid, and code would be the code of the answer with other specify.

8.19.13. Implicit Conversion of Arrays to Strings

The arrays in the previous examples 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 codes ("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

1. "Important"
2. "Design"

The answers are given in APPENDIX A Answers to Exercises.

8.20. Overview – Methods of Basic Variable Objects in Confirmit

		Open text	Date	Numeric	Single	Multi	Open Text List	Numeric List	Ranking	Grid	Geolocation	Element of multi, open text list, numeric list, ranking	Element of grid	Loop
String	.get ()	X	X	X	X						X	X	X	X
	.set ()	X	X	X	X	X	X	X	X	X	X	X	X	X
	.label ()	X	X	X	X	X	X	X	X	X	X	X	X	X
	.text ()	X	X	X	X	X	X	X	X	X	X			
	.instruction ()	X	X	X	X	X	X	X	X	X				
	.value ()		X	X	X						X		X	X
	.valueLabel ()				X								X	X
	.getType	X	X	X	X	X	X	X	X	X	X	X	X	X
Array	.domainValues ()				X	X	X	X	X	X			X	X
	.domainLabels ()				X	X	X	X	X	X			X	X
	.categories ()					X	X	X	X	X				
	.categoryLabels ()					X	X	X	X	X				
	.values ()					X	X	X	X	X				
Boolean	.any()				X	X	X	X	X	X				
	.all()					X	X	X	X	X				
	.none()				X	X	X	X	X	X				
	.between()			X								X[1]		
	.isNearBy()										X			
Double	.latitude()										X			

	.longitude()										X			

[1] numeric list item only.

9. Loop Statements

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

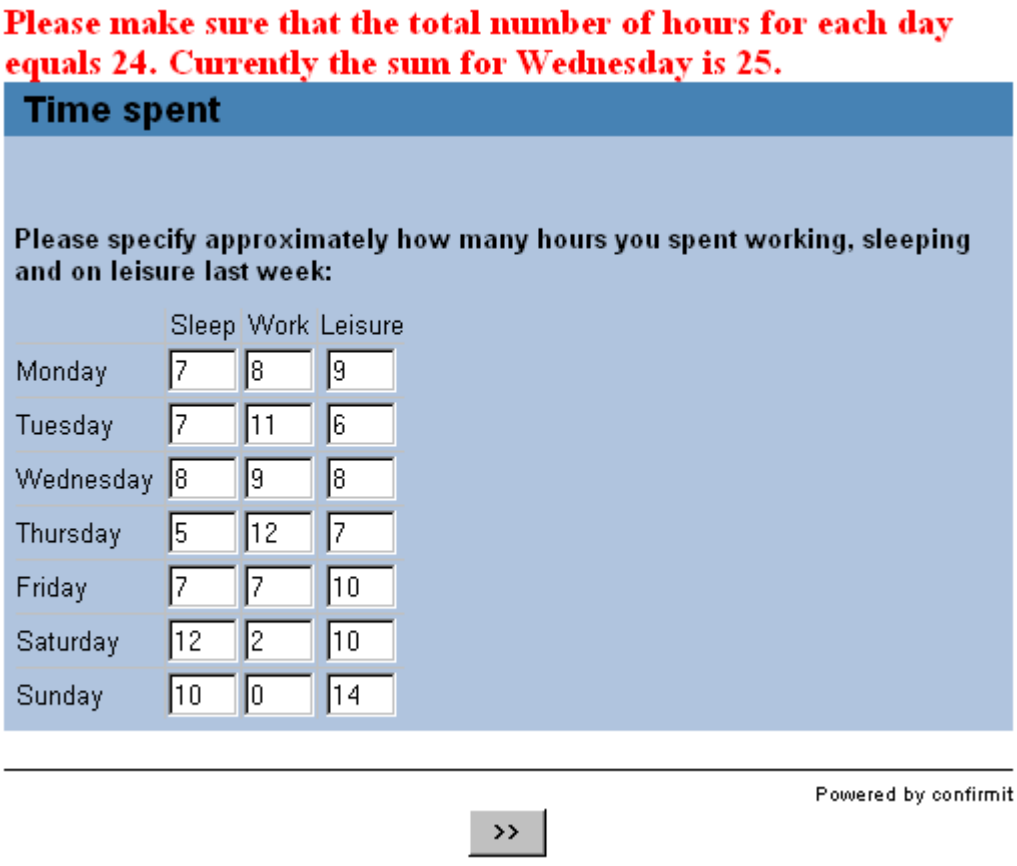
9.1. The while Statement

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

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

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:



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

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

```
var codes = f("q2").domainValues(); //array with all codes
var i : int = 0;
var correctSum : Boolean = true; //Boolean variable. Will be set to
false when a sum is not correct
while(i<codes.length && correctSum)
 {
  var code = codes[i]; //current code
  //calculate the sum for one row:
  var sum : int =

f("q2")[code].toNumber()+f("q3")[code].toNumber()+f("q4")[code].toNumb
er();
  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 above.

The first statement,

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

declares an array called `codes`, which contains all the codes 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 codes ("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:

```
codes[0] = "1"
codes[1] = "2"
codes[2] = "3"
codes[3] = "4"
codes[4] = "5"
codes[5] = "6"
codes[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 codes) without having to change the script.

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

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

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

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

With `i` equal to `0`, the first statement

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

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

A variable `sum` is then set in the statement

```
var sum : int =

f("q2")[code].toNumber()+f("q3")[code].toNumber()+f("q4")[code].toNumb
er();
```

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

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

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

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

```
  i++;
```

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

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

With `i` equal to 1, the first statement

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

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

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

```
var sum : int =

f("q2")[code].toNumber()+f("q3")[code].toNumber()+f("q4")[code].toNumb
er();
```

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<codes.length && correctSum)
```

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

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

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

```
var sum : int =

f("q2")[code].toNumber()+f("q3")[code].toNumber()+f("q4")[code].toNumb
er();
```

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<codes.length && correctSum)
```

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

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

9.2. The do while Statement

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

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

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

Validating Sums in a 3D Grid Using a do while Loop

Using the same example as in Validating Sums in a 3D Grid Using a while Loop, this code would give the same validation of the 3D grid:

```
var codes = f("q2a").domainValues(); //array with all codes
var i : int = 0;
do
 {
  var code = codes[i]; //current code
  //calculate the sum for one row:
  var sum : int =

f("q2a")[code].toNumber()+f("q3a")[code].toNumber()+f("q4a")[code].toN
umber();  var correctSum : Boolean = (sum == 24); //false if sum not
24, true if sum is 24
  i++;
 }
while(i<codes.length && correctSum)

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

9.3. The for Statement

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

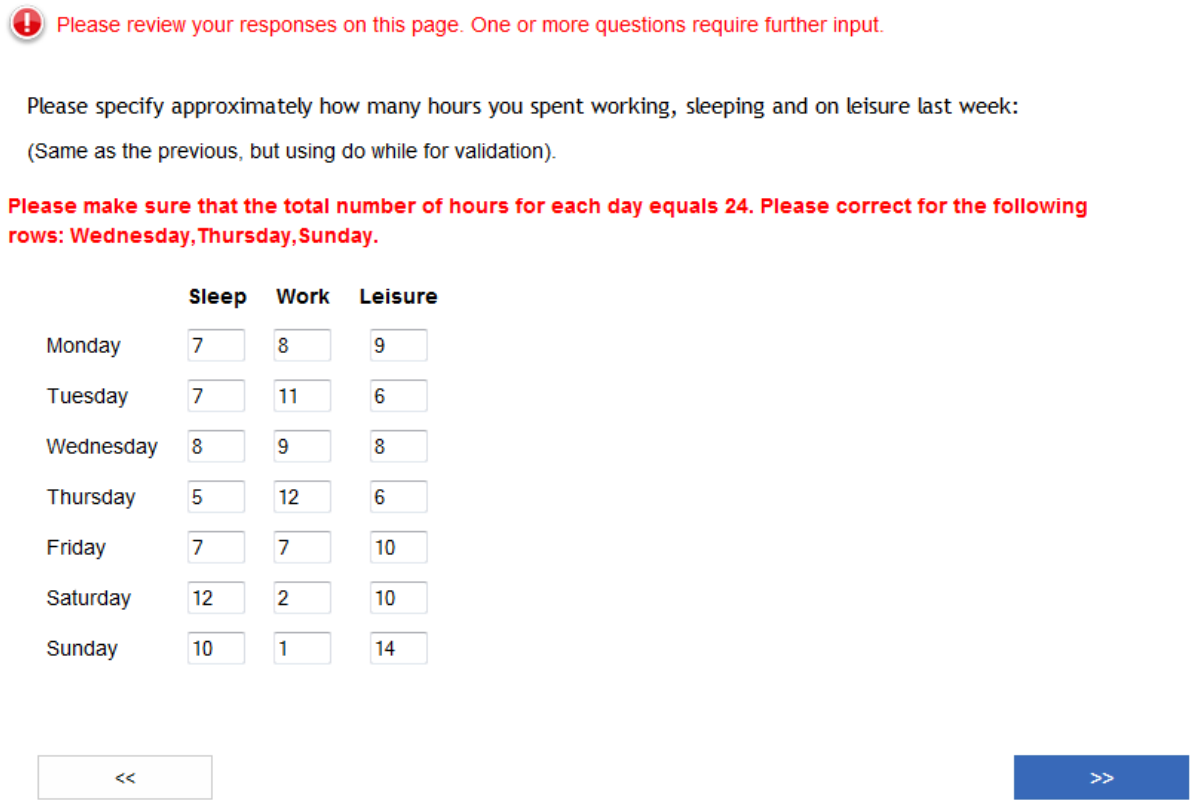
```
for (initializationStatement; condition; updateStatement)
 {
  statements }
```

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

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

Validating Sums in a 3D Grid Using a for Loop

In the previous examples, we stopped validating sums as soon as we found the first error. Here is an example using the *for* statement where we validate all rows and provide an error message listing all rows with errors.



The 3D grid contains 3 numeric list questions: q2 (sleep), q3 (work) and q4 (leisure). In addition we have a hidden multi question *error_rows* which is used to set which rows have incorrect sums.

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

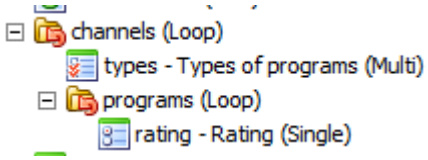
```
var precodes = f("q2").domainValues(); //array with all precodes
var error_rows = new Array();
for(var i = 0;i<precodes.length;i++)
{
var code = precodes[i]; //current precode
//calculate the sum for the row:
var sum : int =
f("q2")[code].toNumber()+f("q3")[code].toNumber()+f("q4")[code].toNumb
er();
if (sum != 24)
error_rows.push(precodes[i]);
}
if(error_rows.length>0)
{
f("error_rows").set(error_rows);
RaiseError();
SetQuestionErrorMessage(LangIDs.en,"Please make sure that the total
number of hours for each day equals 24. Please correct for the
following rows: "+f("error_rows").categoryLabels()+".");
}
```

(The push method is used to add elements to an array (see Removing and Adding Elements on page 187 for more information)).

9.4. Loop Nodes in Confirmit

You can build loops in Confirmit if you want to ask the same question(s) for different elements. The loops can be nested. For example, we can have a loop that iterates through some TV channels asking a question for each TV channel about what kind of programs the respondent watches on the different channels. Then, for those program types there is another loop inside the first one in which they are asked to rate those programs for each channel.

The inner loop is called *programs* and is placed inside the loop called *channels*. In the inner loop we have a single question called *rating* in which we want to ask the respondent about the program types he/she has specified in *types* (the multi question in the outer loop). So the *programs* loop uses the same list of program types as the *types* question, and the loop is filtered with a code 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 code 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 code "3" of the inner loop *programs*. For the outer loop *channels* it is the iteration with code "1".

9.5. The break Statement

```
break;
```

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

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

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

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

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

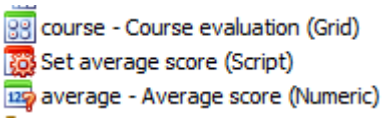
9.6. The continue Statement

```
continue;
```

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

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




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

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

9.7. The label Statement/Nested Loops

Sometimes you need to specify exactly where execution is to continue after a `break` or a `continue` is used. Especially when you have nested loops (loops in loops). To specify which loop you want the `break` or `continue` to apply to, you can specify a label and refer to that in your `break` or `continue` statement. The names of labels follow the same rules as variable names (see Naming on page 14 for more information).

Specifying a label:

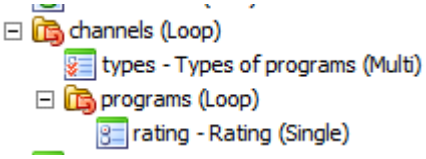
```
label:
```

Referring to a label:

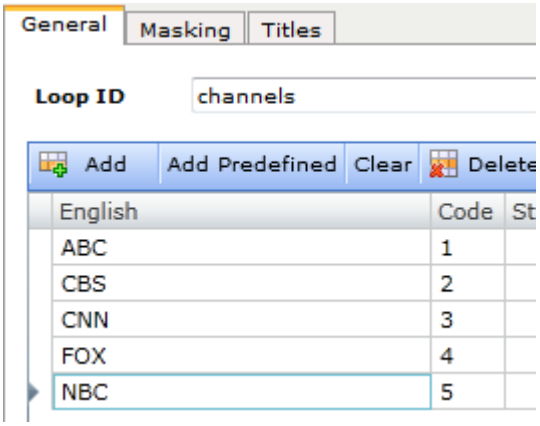
```
break label;continue label;
```

Calculating Averages on a Single Question in a Loop

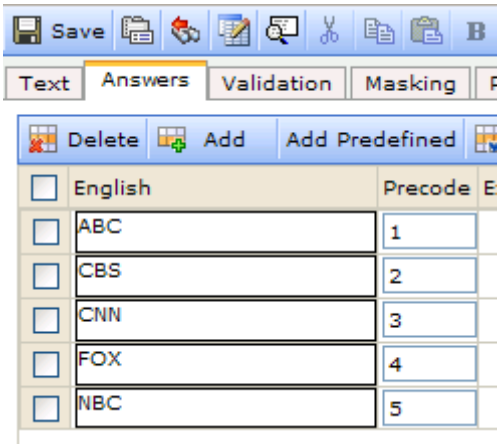
Referring back to the example with two nested loops in Confirmit (see Loop Nodes in Confirmit on page 63 for more information), One iterating through some TV channels, and the other iterating through some types of programs:



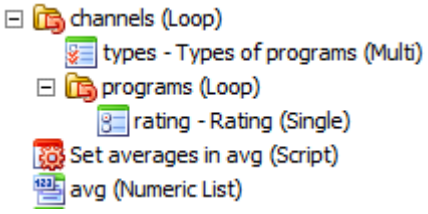
For each channel, the respondent answer what type of programs they watch on that channel, and then give a rating for each program type for that channel in the *rating* question. Now we want to write a script that calculates an average of these ratings for each channel. Here is the list of iterations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



We set up a hidden numeric list question *avg* with the same list:



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



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

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

  //the codes of the program types for this channel:
  var pcodes = f("types",ccode).categories();
   Inner:
  for(var j : int = 0;j<pcodes.length;j++) //run through the codes in
the programs loop
   {
    var pcode = pcodes[j]; //code 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.

The answer is given in APPENDIX A Answers to Exercises.

10. Functions

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

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

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

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

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

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

```
FunctionName()
```

A function may or may not return a value.

10.1. Built-in Functions in Confirmit

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

10.1.1. Arithmetic Functions

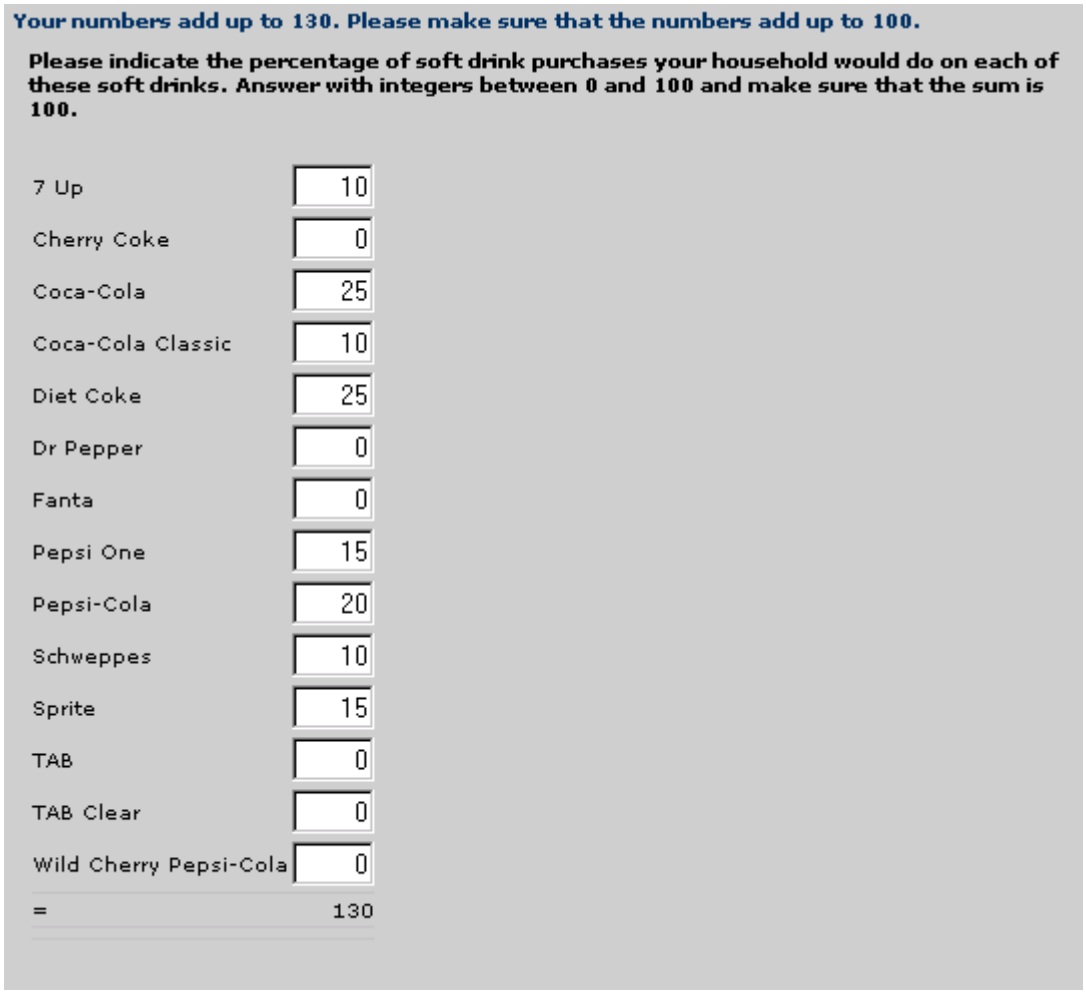
10.1.1.1. Sum

```
Sum( arguments )
```

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

Validating Sum on a Numeric List Question

Here is an example of a numeric list question, in which the respondents are asked to apply percentages that should sum up to 100. You can use the "Auto sum" property to have the sum calculated and displayed directly below the respondent's answers.



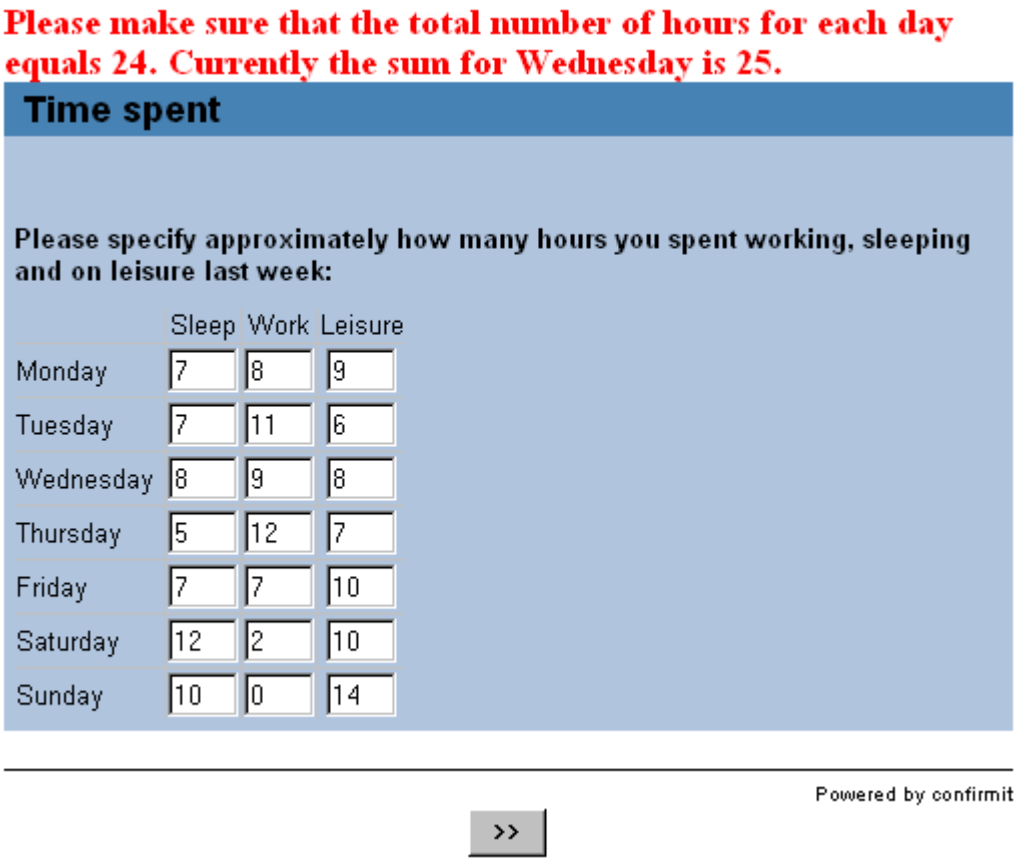
The question id of the numeric list question is *percentage*. In properties total digits is set to 3, scale to 0, and lower and upper limit to 0 and 100, so that the system provided validation makes sure that only integers between 0 and 100 is allowed. To validate that the answer is equal to 100, you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

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

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

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 the examples in The While Statement, The Do While Statement, and The Break Statement for details).



In the code of that example we used the expression

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

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

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

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

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

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

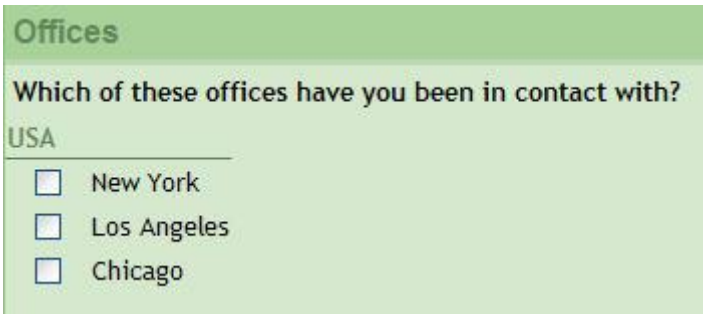
10.1.1.2. Count

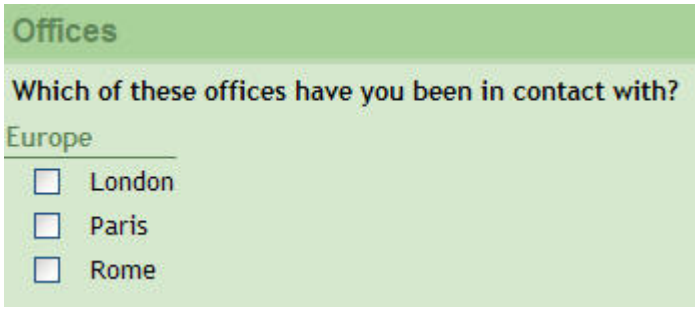
```
Count( arguments )
```

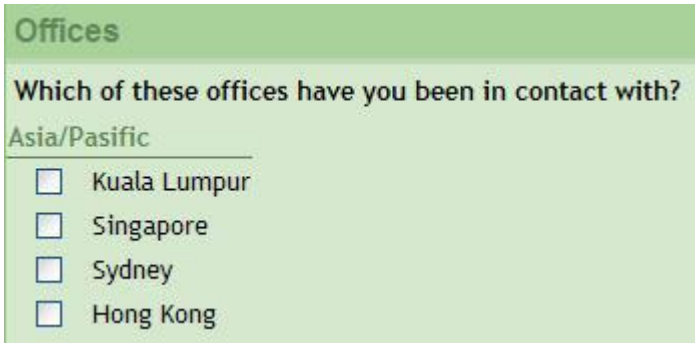
Count returns the number of arguments. Valid argument types are numbers, strings, arrays or any kind of object. Use comma to separate the arguments. An array will be split into its elements, so used on an array `Count` will return the number of elements in the array.

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 on three different questions on three different pages.







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("offices
c").categories()) > 1
```

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

10.1.1.3. Average

```
Average( arguments )
```

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

Calculating Averages on 3 Numeric List 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 three numeric list questions, *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 Sum on page 68 for more information).

10.1.1.4. Max and Min

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

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

Finding Maximum and Minimum Values on Numeric List Questions

Once again, referring to the "hours and weekdays" (see Sum on page 68 for more information), 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 the example (see Sum on page 68 for more information), *maxwork* will get the value 12 and *minsleep* the value 5.

10.1.2. Range

Range has two alternatives:

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

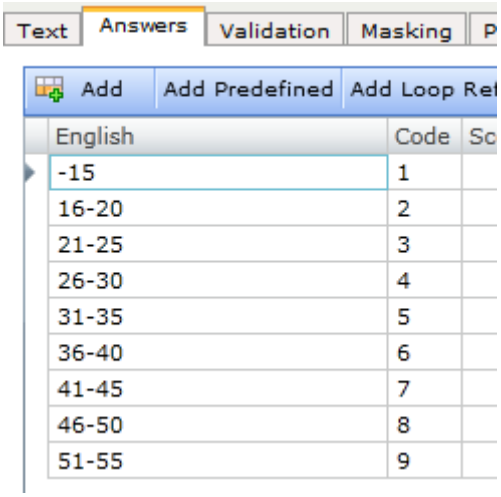
`InRange` returns true if *arg* is within the range (*min*, *max)* (inclusive).

```
InRangeExcl(arg,min,max)
```

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

Building a Condition on a Range of Codes

You may need your conditions to work for several codes within a range. For example, you may have an *age* question where this is a part of the answer list:



If you wanted some questions to be sent only to people between the ages of 26 and 50, you could use a condition with an expression as follows:

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

but in this case it will be easier to use the `InRange` function:

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

10.1.3. Context Information

10.1.3.1. GetSurveyChannel

```
GetSurveyChannel()
```

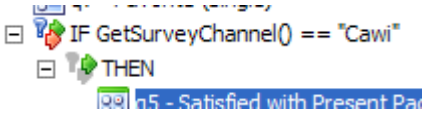
`GetSurveyChannel` is used in surveys that are running both on web and on Confirmit CAPI, and returns which channel the interview was performed in as a string.

String returned	Description
Cawi	Web interview
Capi	CAPI interview
Cati	CATI interview
RandomDataGeneration	Response is generated using Random Data Generator

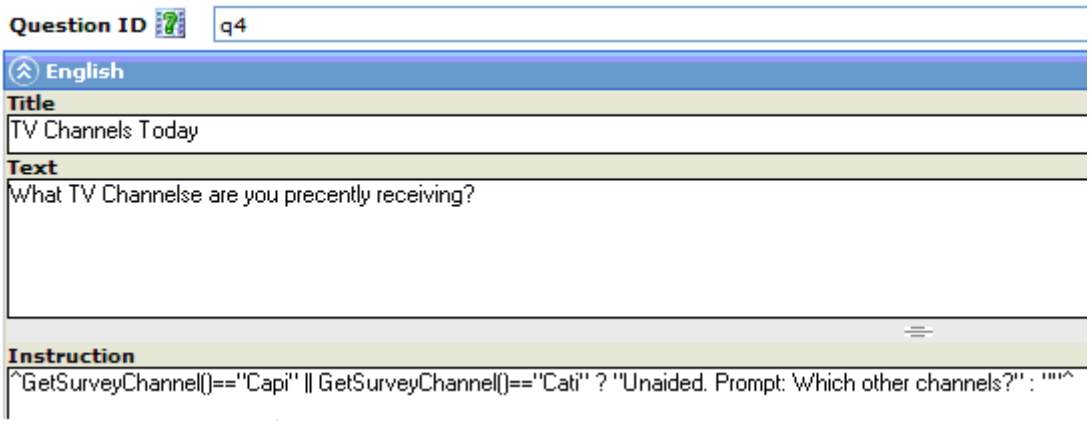
This function can for example be used to store the channel in a hidden single question `channel` by setting it in a script node:

```
f("channel").set(GetSurveyChannel())
```

Another example is to use it inside a condition to ask certain questions only to web respondents:



A third example is to use it to display certain texts only when the interview is performed by an interviewer (CAPI or CATI):



10.1.3.2. IsInProductionMode

```
IsInProductionMode()
```

IsInProductionMode() can be used to distinguish between running surveys in production mode or in quick test/external quick test/test mode. It will return one of the following values:

true	Production mode
false	Quick Test/External Quick Test/Test mode

10.1.3.3. GetRenderingMode

```
GetRenderingMode()
```

GetRenderingMode() can be used to determine whether it is desktop rendering or one of the mobile rendering modes (touch or generic) that is currently active in the survey. This can be used for example in a condition to have some questions only appear if the survey is taken from a PC/Mac (desktop rendering).

To get the mobile rendering modes (touch and generic), it requires that those modes are active on the survey.

desktop	Desktop rendering (PC/Mac/tablet)
touch	iPhone/iPod touch/Android phones
generic	All other mobile phones

If you for example want to provide some questions only to users accessing a survey through the desktop rendering, you can use the following expression in a condition, and then place those questions inside the Then branch:

```
GetRenderingMode() == "desktop"
```

10.1.3.4. GetContentType

```
GetContentType()
```

GetContentType()can be used to distinguish between surveys run as regular Web surveys or through a Flex extension using the Survey Front End extension point, such as iPhone, Android and SMS surveys. It will return one of the following values:

Html	Run as a regular web survey
Json	Run using the Flex Survey Front End extension point using Json

Xml	Run using the Flex Survey Front End extension point using Xml

10.1.3.5. AdvancedWIFeaturesEnabled

```
AdvancedWIFeaturesEnabled()
```

AdvancedWIFeaturesEnabled() is used to determine whether the respondent's browser supports Advanced WI Features that require client side scripts, such as sliders, drag-n-drop ranking, images instead of radio-buttons/check-boxes etc. It will return true if the respondent's browser supports the advanced WI features, or false if not. It can for example be used to display a different instruction text depending on whether or not respondents will have a question displayed with the advanced WI interface.

This function is optimistic with regards to checking whether JavaScript is turned on or off in the browser. In other words it will return true unless it is determined that JavaScript is turned off. Note that it is only possible to determine this state after the first page of the survey, therefore this function will give more precise results if it is executed after the first page of the survey.

Displaying different instruction if Advanced WI feature is being used

If you have a question using the drag-n-drop ranking feature, you may present two different instruction to respondents: One for respondents that have a browser supporting the drag-n-drop interface, and another one to respondents that do not have a browser supporting these. The latter will then get the question with ordinary input boxes where they can rank with numbers.

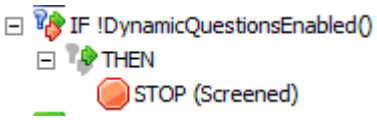
This can be done by using a ternary conditional expression with the AdvancedWIFeaturesEnabled function in the instruction text field of the questionnaire.

```
^AdvancedWIFeaturesEnabled() ? "Please rank the cars by dragging them
over to the right hand side, the first indicating the one your are
most likely to buy and so on" : "Please rank the cars by entering
numbers in the text boxes. 1 for the car your are most likely to buy,
2 for the second most likely and so on"^
```

10.1.3.6. DynamicQuestionsEnabled

```
DynamicQuestionsEnabled()
```

DynamicQuestionsEnabled() is used to determine whether the respondent's browser supports AJAX so that the "Dynamic Questions" functionality can be used (see Dynamic Questions on page 9 for more information). It will return true if the respondent's browser supports AJAX, or false if not. It can for example be used for screening respondents from a survey that have extensive usage of .this, or in a condition to provide an alternative path for respondents, if they have a browser that does not support it. Unlike "Advanced WI Features", no fallback (i.e. alternative interface) exists for this functionality.



10.1.3.7. IsDynamicQuestionCallback

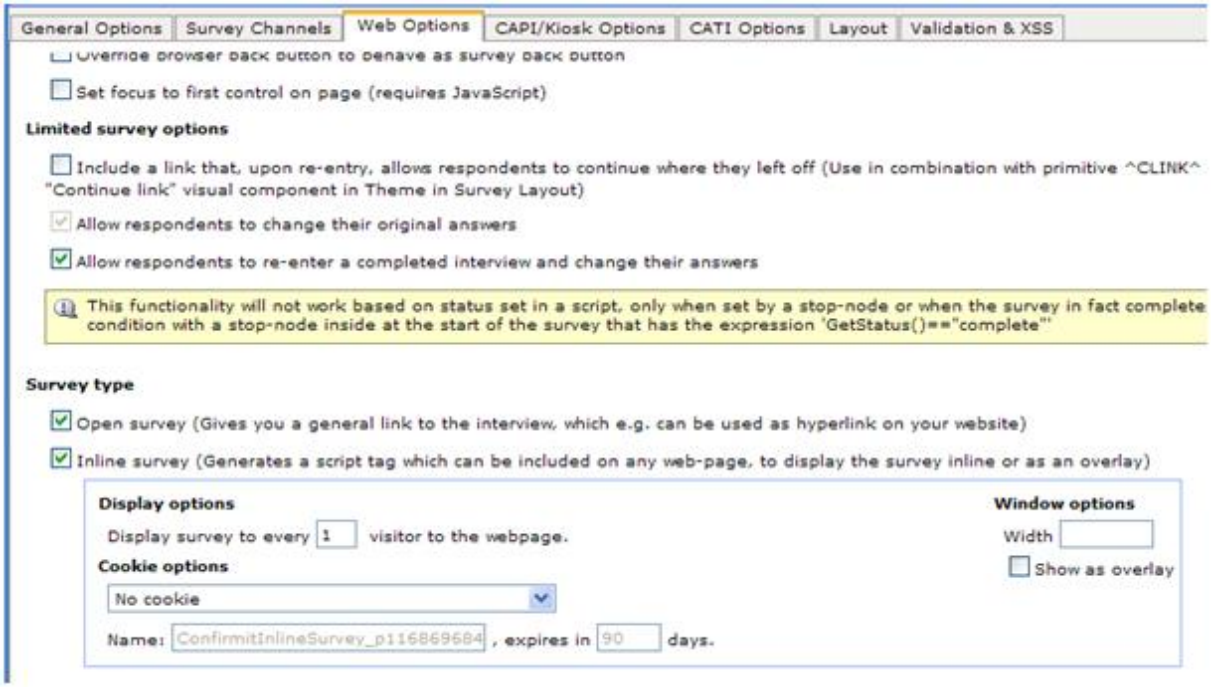
```
IsDynamicQuestionCallback()
```

IsDynamicQuestionCallback is used to determine whether the code is currently executed during a Dynamic Question Callback, i.e. an Ajax update of one or more Dynamic Questions. This will be true if used within a question that has a trigger, when one or more of the trigger questions is updated.

10.1.3.8. IsInlineSurveyCallback

```
IsInlineSurveyCallback()
```

IsInlineSurveyCallback is used to determine whether the survey is currently run as an inline survey.



For "regular" surveys, the response record is created when the respondent opens the first page of the survey (this is the default). However for inline surveys, the response record is created when the respondent submits the first page of the survey. This is done to reduce the amount of empty data records, since an inline survey may be placed within high volume sites where only a small percentage of the visitors will submit a response.

It is therefore not possible to use the .set() method in a script within the first page to store responses when the survey is running inline, because the data cannot be stored in the database since the database record does not yet exist. Such scripts can only be run from the second page onwards.

If you need to retrieve values from the query string using Request (see Request on page 199 for more information) and store in the database, you will have to do this on the second page. In inline mode the query string is preserved when moving from the first to the second page.

If the survey is accessed both in inline mode and as a regular survey link, you do have to treat these modes differently, and the IsInlineSurveyCallback function can be used for this. Let's say you send in a value site=1. At the start of the survey you can add this script to fetch the value when the regular survey link is opened:

```
if(Request("site")!=null &&!IsInlineSurveyCallback())
{
f("site").set(Request("site"));
}
```

Before the second page, you can insert the following script to fetch the value when the survey is inline:

```
if(Request("site")!=null &&IsInlineSurveyCallback())
{
f("site").set(Request("site"));
}
```

10.1.3.9. CurrentForm

```
CurrentForm()
```

`CurrentForm` is used in validation code, masking and in response piping in a question's text fields (or in functions called from validation code, masking and response piping) and returns the question ID (string) of the current question. Used inside the validation code field of one of the questions inside a 3D grid, it will give the question id of that question instead of the id of the 3D grid itself.

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

10.1.3.10. GetRespondentUrl

```
GetRespondentUrl()
GetRespondentUrl(string qid)
GetRespondentUrl(string qid, params string[] loopQual)
GetRespondentUrl(string id, bool isCallBlock)
GetRespondentUrl(string id, bool isCallBlock, string parameters)
GetRespondentUrl(string id, bool isCallBlock, params string[]
loopQual)
GetRespondentUrl(string id, bool isCallBlock, string parameters,
params string[]loopQual)
```

Where:

qid is a question id.

id is either a question id or a callbock id.

isCallBlock denotes whether the id is a question id or a callblock id.

parameters is a string to be passed into the survey in the url; the string can contain one or more parameters; where more than one parameter is being used a semi-colon should be used as a separator.

loopQual is only applicable when the question id exists within one or more loops or the callblock contains a loop, and the url is being used to direct the respondent to a particular iteration.

GetRespondentUrl returns a URL that will allow the current interview to be entered at a specific point.

All of the parameters are optional; if they are not used, the URL will take the user to the beginning of the interview. But when they are used, they can provide the following capabilities:

1. The respondent can be directed to the page of a particular question, and, if the question is inside one or more loops, a particular loop iteration or iterations. The interview will then continue from that point.
2. The respondent can be directed to a particular callblock which, upon completion of the callblock, will finish the interview.
3. The function can securely pass a text value or several text values into the start of the interview, that will then be accessible via the UserParameters function.

GetRespondentUrl can be used for example to send an email to the respondent to allow him/her to re-access the survey, or it can be presented inside the survey so that the respondent can copy it to return to the survey later.

To follow are some examples of different combinations of the optional parameters, and cases where they may be used.

In this example,

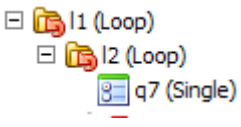
```
GetRespondentUrl("q9")
```

will give a link that when opened, will open the survey for the current respondent on the page with q9, and then continue through the interview. This could also be written as:

```
GetRespondentUrl("q9", false)
```

which will have the exact same effect.

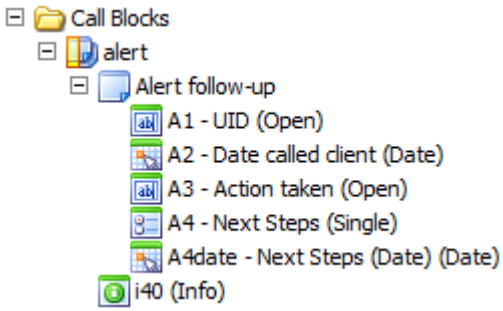
In this example,




```
GetRespondentUrl("q7","3","2")
```

will give a link that when opened, will open the survey for the current respondent on the page with q7 for the iteration with code "3" of l2 and the iteration with code "2" of l1.

In this example,




```
GetRespondentUrl("alert", true)
```

will give a link that when opened, will open the survey for the current respondent at the start of the call block named “alert”. When the respondent finishes the callblock, after submitting the i40 info node, the interview will finish.

In this example,

```
GetRespondentUrl("q12", false, "alert=true")
```

will give a link that when opened, will open the survey for the current respondent on the page with q12, passing the encrypted parameter named alert with a value of “true” (see the UserParameters function for details regarding accessing the parameter).

In this example,

```
GetRespondentUrl("", false, "alert=true;state=red")
```

will give a link that when opened, will start from the beginning of the survey, passing the encrypted parameters named “alert” and “state” with the values of “true” and “red” (see the UserParameters function for details regarding accessing the parameter).

Note: When using GetRespondentUrl to pass several text values into the start of the interview, a semicolon ; is used as a separator. If a ; is required as part of the parameter, it must be encoded. The plus sign + must also be encoded if used as a parameter.

In this example,

```
GetRespondentUrl("cb1", true, "alert=true")
```

will give a link that when opened, will open the survey for the current respondent at the start of the call block named “cb1”, passing the encrypted parameter named alert with a value of “true” (see the UserParameters function for details regarding accessing the parameter). When the respondent finishes the callblock, the interview will finish.

Note: When GetRespondentUrl is used to access a callblock, the interview_start, interview_end and interview status values are not updated during the callblock interviewing session. However, if the callblock contains a STOP node then interview_end and interview status will be updated.

Important
GetRespondentUrl should not be used to start the interview at a question contained within a callblock as the interview will fail when the respondent reaches the end of the callblock.

GetRespondentUrl access to a callblock will cause the interview to finish when the callblock is completed. It is not designed to act as a scripted way to access a callblock.

GetRespondentUrl must not be used with CATI; use GetCatiRespondentUrl **(see GetCatiRespondentUrl on page 102 for more information).**

Building a Cryptic URL to be displayed in an info node

Even for the respondents who answer open surveys (for example pop-ups), a unique URL can be used to reenter the survey.

So if you want respondents to an open survey such as for example a pop-up survey to be able to quit in the middle of the survey, close their browser and then continue later, they just need to have the correct link with the respondent-specific parameter. They will then be able to enter their own survey with previous answers intact. Without this link, they will have to start on a new survey.

To achieve this, one solution is to display the respondent-specific URL in the interview, instructing the respondent to copy that exact URL if they want to pause and continue later, for example:

In the example above, the survey setting “Encrypt System Request Parameters” is set. If this is disabled, the link will have r=respid&s=sid instead of __sid__. GetRespondentUrl supports either option for respondent-specific URLs.

You can insert the respondent-specific URL in the text area of an info node as follows:

```
^GetRespondentUrl()^
```

The same functions can be used to email a personal link to a respondent (see SendMail on page 128 for more information).

10.1.3.11. CurrentID

```
CurrentID()
```

`CurrentID` returns the respid of the current respondent. Note that this does not necessarily apply to CAPI interviews; CurrentID on the CAPI Console returns the CAPI local respondent ID.

Using modulus to route respondents to different parts of the questionnaire

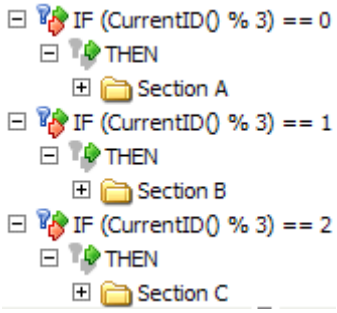
You can use the remainder of respid divided by a number x to route respondents to x different parts of the questionnaire in rotation. For example, if you have three sections A, B and C, you can create a pattern as:

respondent 1 would complete section C,
respondent 2 would complete section B,
respondent 3 would complete section A,
respondent 4 would complete section C,
respondent 5 would complete section B,
respondent 6 would complete section A,
etc.

This can be done by using the CurrentID() function and modulus / remainder. Depending on the result of

```
CurrentID()%3
```

the respondent is routed to the appropriate part of the questionnaire.



10.1.3.12. CurrentLang

```
CurrentLang()
```

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

10.1.3.13. 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.

10.1.3.14. GetRespondentValue and SetRespondentValue

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

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

```
GetRespondentValue(fieldname)
```

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

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

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

Calculating the number of days elapsed since a record was uploaded

The following script works out the number of days passed since a record was uploaded in the respondents database.

```
f('UploadDate').set(GetRespondentValue("CreatedDate"))
//where UploadDate is a date type question

var start: DateTime =GetRespondentValue("CreatedDate").toDate();
var end = new Date();
var span : TimeSpan = end - start;
var days = span.TotalDays;

f('nodays').set(days)
```

10.1.3.15. InterviewStart, InterviewEnd, SetInterviewStart and SetInterviewEnd

When a respondent enters a Confirmit interview, an 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, an 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: If a respondent re-enters the interview, *interview_start* is re-set, and similarly, if she or he reaches a stop node or the end of the interview after re-entering a completed interview, the variable *interview_end* is also re-set.

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

```
InterviewStart()
```

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

```
SetInterviewStart(date)
```

`SetInterviewStart` sets the respondent's interview. The parameter *`date`* is optional. If it is not included, `interview_start` will be set to current server time and date. If *`date`* is included, it can either be a JScript.NET Date object or a .NET DateTime object.

```
InterviewEnd()
```

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

We will return to examples on how to use these functions later in the documentation (see InterviewStart and InterviewEnd on page 160 for more information) where they are used together with the Date object.

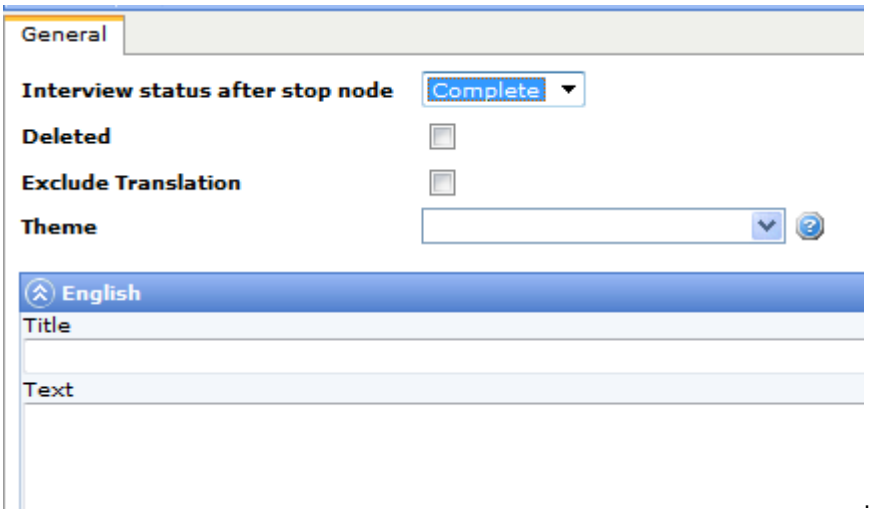
There are also two functions available to set these timestamps. `SetInterviewStart` can be used for example when you want interview start to be set after the screening questions of a survey instead of at the beginning of the survey, and `SetInterviewEnd` can be used to set the interview end timestamp when a stop node is never reached because of a redirect at the end of the survey (see Redirect on page 133 for more information).

`SetInterviewStart()` - sets the respondent's interview start time to current server time and date.

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

10.1.3.16. GetStatus and SetStatus

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



Currently Confirmit recognizes the following status settings:

Status	Code
Complete	complete
Screened	screened
Quota Full	quotafull
Error	error

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

There are two functions that operate on status:

```
GetStatus()
```

`GetStatus` returns the current interview status.

```
SetStatus(status)
```

`SetStatus` can be used in a script to set the interview status, e.g. if you want respondents who have answered the questionnaire up to a certain point to count as complete interviews, even though they do not answer all the remaining questions. *`status`* is a string with the status value you want to set, i.e. "complete", "screened" or "quotafull". `SetStatus` is very often used in combination with the Redirect function (see Redirect on page 133 for more information)

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

Note: The use 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 Confirmit.

Setting Interview Status Before End of Survey

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

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

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

10.1.3.17. Forward

```
Forward()
```

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

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

10.1.3.18. IsInRdgMode

```
IsInRdgMode()
```

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

10.1.3.19. SetRandomCategories

```
SetRandomCategories(number, qid)
```

SetRandomCategories can be used to randomly switch on the specified number of categories of the specified question ID. Typically this would be used to randomly turn on number categories of a specific hidden qid. In the example below, two categories of the hidden question q2 would be switched on.

```
SetRandomCategories(2, 'q2')
```

This function is only applicable for standard multi questions. It does not apply to multi questions with the Capture Order property set.

(see Random on page 167 for more information)

10.1.3.20. TerminateLoop

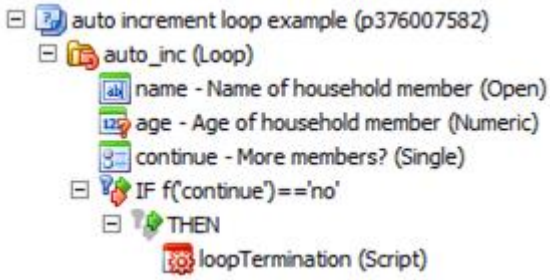
```
TerminateLoop()
```

TerminateLoop is used inside of auto-increment loops. Auto-increment loops are only available for optimized database projects. A TerminateLoop call is required to stop further iterations of the loop being created. Without this, the loop will continuously create new loop iterations, and the respondent will never be able to finish the loop.

A script node containing a TerminateLoop function call is a mandatory requirement for an auto-increment loop because an auto increment loop iteration requires some interactive content for the respondent (question, info node etc.). If the loop contains nothing interactive, the interview will be terminated with an error status and an error notification will be sent via email.

Note that the TerminateLoop function call must be placed at the end of the loop to have any effect.

In the following example the loopTermination script node contains the single function call TerminateLoop and is used to prevent further iterations of the auto-increment loop based on the answer to the "continue" question.



TerminateLoop calls are only applicable to auto-increment loops; they are not applicable to, and have no effect on loops based on normal answer lists or those based on table lookups.

10.1.3.21. IsAccessibleMode and SetAccessibleMode

```
IsAccessibleMode()
```

`IsAccessibleMode` returns true if the current Survey Layout Theme is a theme in Accessible mode. The function is useful if you some respondents are responding in accessible mode and some not, and you would like different content in the two modes.

```
SetAccessibleMode(bool)
```

`SetAccessibleMode` can be used to either set the survey in accessible mode (i.e. switch to a Survey Layout Theme that is set as Accessible):

```
SetAccessibleMode(true)
```

or to set the survey not to be in accessible mode (i.e. switch to a Survey Layout Theme that is not set as Accessible):

```
SetAccessibleMode(false)
```

10.1.3.22. UserParameters

```
UserParameters[paramName]
```

UserParameters can be used to access texts that are passed to the start of the interview using the GetRespondentURL() function.

In this example,

```
UserParameters["alert"]
```

will give the value that the parameter named “alert” was set to when GetRespondentUrl was used to access the survey.

Note: UserParameters can only be used on the first page that the respondent is directed to via GetRespondentUrl. UserParmeters will return empty values if used later in the survey.

To access all parameter information UserParameters.Keys can be used. The Keys function returns a NameValueCollection which can be iterated through.

In this example, a script function “ShowKeys()” has been created to display all parameter information:

```
function ShowKeys()
{
 var str = "";
 for(var x = new Enumerator(UserParameters.Keys); !x.atEnd();
x.moveNext())
 {
 var y = x.item();
 str += "<br />" + y + " = " + UserParameters(y);
 }
 return str;
}
```

When piping the result of this function, for example using ^ShowKeys()^ in an interview initiated via GetRespondentUrl("", false, "alert=true;state=red") the following the following will be displayed:

10.1.3.23. GetDeviceInfo

```
GetDeviceInfo()
```

GetDeviceInfo() is a scripting function that returns an object containing a variety of information regarding the type of device that is being used by the respondent. Note that the information returned is dependent on the rendering mode used for the survey.

The object returned contains the properties described in the following table:

Property Name	Type	Value	Meaning
IsGeneric	Boolean	true or false	Returns true if the respondent is using a generic mobile device.
IsDesktop	Boolean	true or false	Returns true if the respondent is using a desktop device.
IsTouch	Boolean	true or false	Returns true if the respondent is using a touch mobile device, for example an iPhone or Android phone.
IsTablet	Boolean	true or false	Returns true if the respondent is using a tablet device.
IsMobile	Boolean	true or false	Returns true if the respondent is using a mobile device.
ScreenResolution	String	“600|400” or “0|0”	The screen resolution of the device being used width|height or “0|0” if unknown.
DeviceType	Int	1, 2 or 3	Where 1 means desktop rendering, 2 means touch rendering and 3 means generic rendering.
DeviceType.toString()	String	"Desktop", "Touch", "Tablet" or "Generic"	The device type in use.

For example, an info node containing the following:

```
The results of the function calls are:
GetDeviceInfo().IsDesktop: ^GetDeviceInfo().IsDesktop^
GetDeviceInfo().IsTouch: ^GetDeviceInfo().IsTouch^
GetDeviceInfo().IsTablet: ^GetDeviceInfo().IsTablet^
GetDeviceInfo().IsGeneric: ^GetDeviceInfo().IsGeneric^
GetDeviceInfo().IsMobile: ^GetDeviceInfo().IsMobile^
GetDeviceInfo().ScreenResolution: ^GetDeviceInfo().ScreenResolution^
GetDeviceInfo().DeviceType: ^GetDeviceInfo().DeviceType^
GetDeviceInfo().DeviceType.toString():
^GetDeviceInfo().DeviceType.toString()^
```

Will return the following information on an iPhone 4:

Note: ScreenResolution will only be available if Smartphone rendering mode is enabled and the device is either a touch or tablet device.

Note: The definition of "Tablet" is a device with a user agent string conforming to the UserAgentRegExTouchDevice expression from System Configuration that also has at least one screen dimension exceeding 650px.

When only Desktop mode is enabled, the screen resolution check is not performed so IsTablet will always return false.

Important
Screen dimension is often considered to be the actual screen resolution that is advertised on a phone's specification sheet (i.e. the screen.width and screen.height * the device pixel ratio). If you are using and relying on the GetDeviceInfo().ScreenResolution information in your survey, be aware that Confirmit uses the viewport's width and height (that is window.innerWidth and window.innerHeight). Confirmit also uses the CSS Interpretation (the resolution / the device pixel ratio) divided again by the device pixel ratio. These are the numbers that are returned for GetDeviceInfo().ScreenResolution, and the numbers used to determine tablet.

Important
Be aware that although iPad is classified as a tablet device, iPad is included in the desktop regex (the UserAgentRegExDesktop expression from System Configuration) that is configured for Confirmit. Therefore if a respondent is taking a survey on an iPad device, IsTablet will never return true; it only gets as far as the first check (the user agent string) and returns false.

10.1.3.24. GetQuestionIds

GetQuestionIds is used to return a string array of question IDs based on specific filters provided.

```
GetQuestionIds(questionTypes: String[], variableTypes: String[],
loopId: String)

GetQuestionIds(includeInteractiveQuestions: Boolean,
includeNonInteractiveQuestions: Boolean, loopId: String)
```

Where:

`questionTypes` is a string array that filters and lists questions which are of a specific type only. The types are (case insensitive): "single", "multi", "grid", "ranking", "open text list", "numeric list", "open", "numeric", "date", "geolocation", "multigrid" and "3dgrid".

`variableTypes` is a string array which filters and lists questions which are of a variable type, the types are (case insensitive): "normal", "panelvisible", "hidden", "background", "recoded" and "recoding".

`loopId` filters and lists questions inside the specified loop only (not recursive).

`includeInteractiveQuestions` if set true will filter and list all questions that have normal or panel visible variable types.

`includeNonInteractiveQuestions` if set true will filter and list all questions that have hidden, background, recoded and recoding variable types.

Note that only root nodes of multigrids and 3dgrids are listed. It is possible to get title, text and instruction of a multigrid or a 3dgrid using f(g1).text() or .label() or .instruction().

In this example,

```
GetQuestionIds(true, false)
```

will return a string array of all interactive questions in the survey.

In this example,

```
var questionsTypesIncluded : String[] = ["open", "numeric"];
f("myQuestions").set(GetQuestionIds(questionsTypesIncluded).toString()
);
```

will set the ”myQuestions” answer to be a comma-separated list of all open text and numeric questions in the survey.

10.1.3.25. Get3DGridQuestionIds

Get3DGridQuestionIds is used to return a string array of the questions inside a specific 3DGrid.

```
Get3DGridQuestionIds(gridId: String)
```

Where:

`gridId` is the question ID of a 3Dgrid.

In this example,

```
Get3DGridQuestionIds("g1").toString()
```

will return a comma-separated list of all question IDs inside the 3DGrid “g1”.

10.1.4. Browser Information

10.1.4.1. BrowserType, BrowserVersion and trapBrowser

```
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:

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());
```

trapBrowser can be used to set browser type and browser version as a combined string into a question:

```
trapBrowser(qID)
```

qID is the question ID of the opentext question in which the browser type and browser version is to be stored.

10.1.4.2. RequestIP

```
RequestIP()
```

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

Note: Recording the respondents' IP addresses may be in conflict with privacy of the respondents. Responsibility for ensuring respondents' privacy resides with the Confirmit clients. Confirmit recommends following ESOMAR guidelines, see www.esomar.org.

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 repeat IP addresses. However, this probably will not give the desired effect, because:

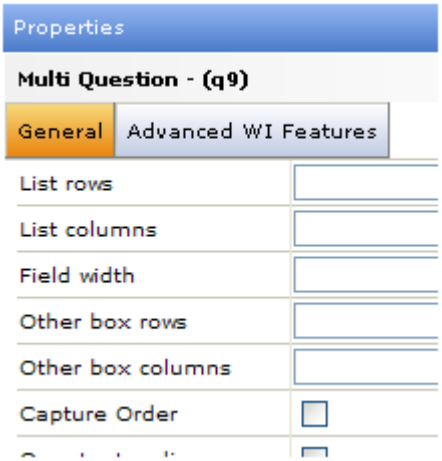
- respondents inside a firewall will have the same IP address
- internet providers may use dynamic IP addresses

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 IsNet on page 128 for more information).

10.1.5. Ranking Questions and Capture Order Multis

The functions described in this section can be used on ranking questions and multi questions with the "Capture Order" property set. These questions will, for each answer list item, store an integer which represents the rank it was given (for example ranked as number 1) or in which order it was selected (selected 1st). A few functions are provided to easily retrieve the element that was selected first etc.



Note: The Capture Order property is only available in surveys using the Optimized database.

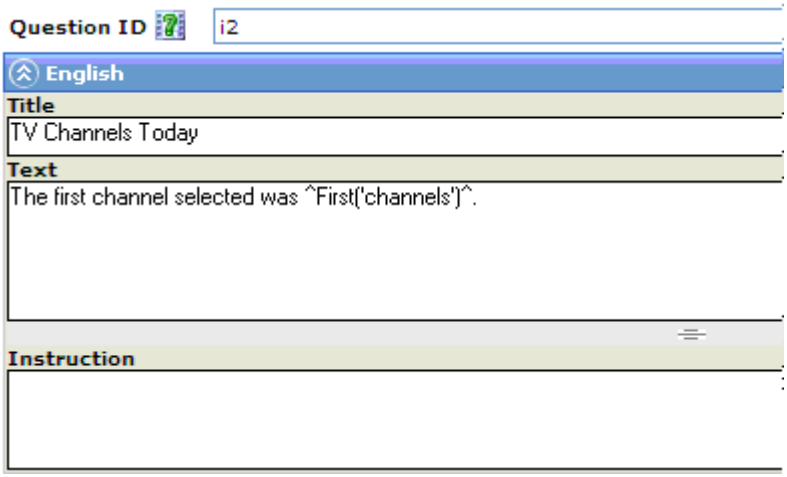
10.1.5.1. First

This function is used to return the first category selected for multi choice questions with the capture order property, or the category ranked as 1st for a ranking question.

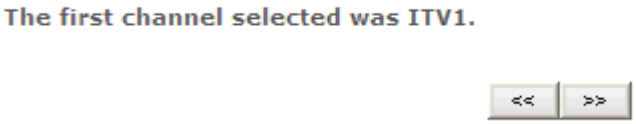
```
First(qid)
```

First returns the code or label, depending on the context, of the first selected or ranked as number 1 answer category for the specified question ID qid. Label will be returned in text piping mode (within ^s), code elsewhere.

This function can only be used on ranking questions and multi's that have the capture order property enabled. It is not applicable to other multis.



If these responses were given in the order ITV1, BBC1, Channel 5, we would get the following result from using First():



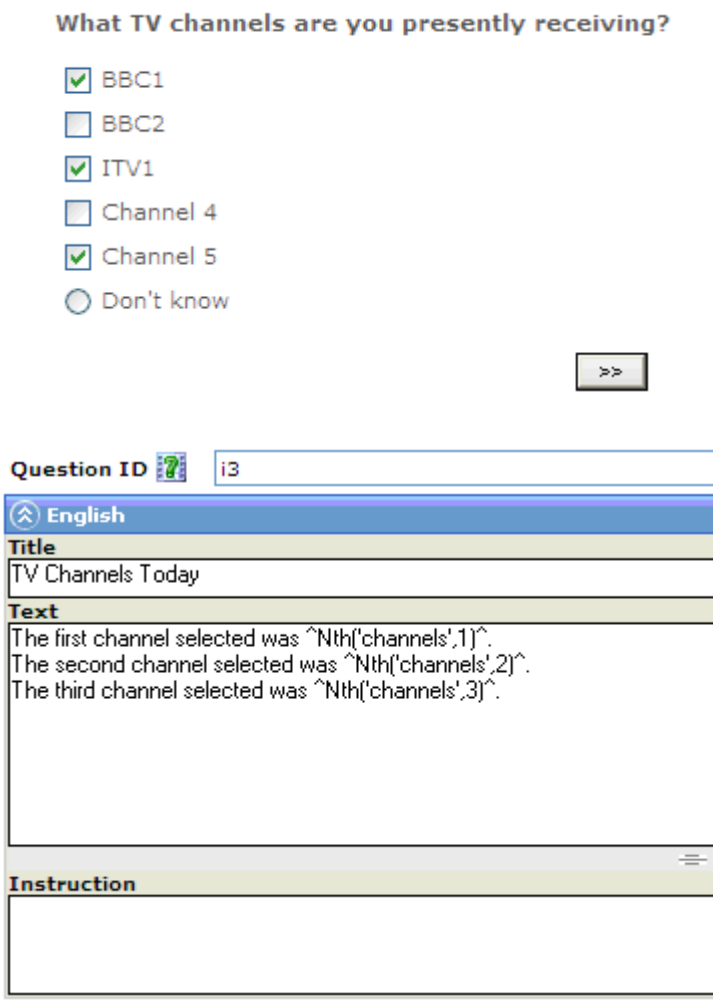
10.1.5.2. Nth

This function returns the nth category selected for a multi choice question with the capture order property, or the category ranked as the nth for a ranking question.

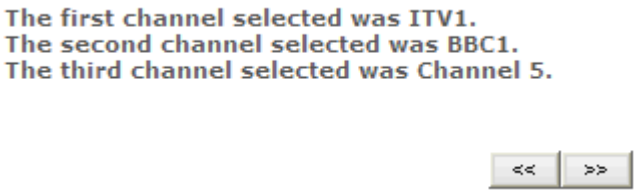
```
Nth(qid,position)
```

Nth returns the code or label, depending on the context, of the answer category in the position specified for the specified question ID. Label will be returned in text piping mode (within ^s), code elsewhere.

Nth(“q1”,1) will return the same value as First(“q1”). This function can only be used on ranking questions or multi's that have the capture order property enabled. It is not applicable to other multis.



If these responses were given in the order ITV1, BBC1, Channel 5, we would get the following result by using Nth():



10.1.5.3. AnswerOrder

This function is used to return an array of codes in the order selected for a multi choice question, or in the order ranked for a ranking question.

```
AnswerOrder(qid)
```

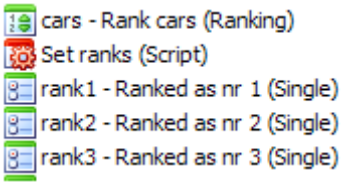
AnswerOrder returns the codes of the answers supplied in the order they were selected or ranked for the specified question ID. The primary purpose of this function is to make it easy for users to retrieve the selected answer codes without having to use the Nth() function and loop through the answer list multiple times, which would result in poor script and server performance in the event of a very long answer list.

This function can only be used on ranking questions or multi's that have the capture order property enabled, it is not applicable to other multis.

10.1.5.4. Setting Variables for Nth Mentioned / Ranked for Reporting Purposes

If you have a ranking question or a capture order multi, you will easily be able to report on which ranks/order of mention each answer got. However, if you want to report on answers per rank/order of mention, you will save some recoding work by creating a set of hidden single variables for each rank/order of mention, that have the same answer list as the multi.

In this example we have a question “cars”, in which the respondents are asked to rank up to three of their favorite cars. The three hidden single questions rank1, rank2 and rank3, are set up with the same answer list as the multi, and will be used to store the car selected as number 1, 2 and 3.



The following script will set the single questions based on the responses to the ranking question:

```
var ordered = AnswerOrder("cars");
for(var i=0;i<ordered.length;i++)
{
f("rank"+(i+1)).set(ordered[i]);
}
```

10.1.6. Table Lookup Specific Functions

10.1.6.1. GetDBColumnValue

This is a survey engine scripting function used to retrieve data from an additional column or columns associated with a specific table inside of a table lookup list.

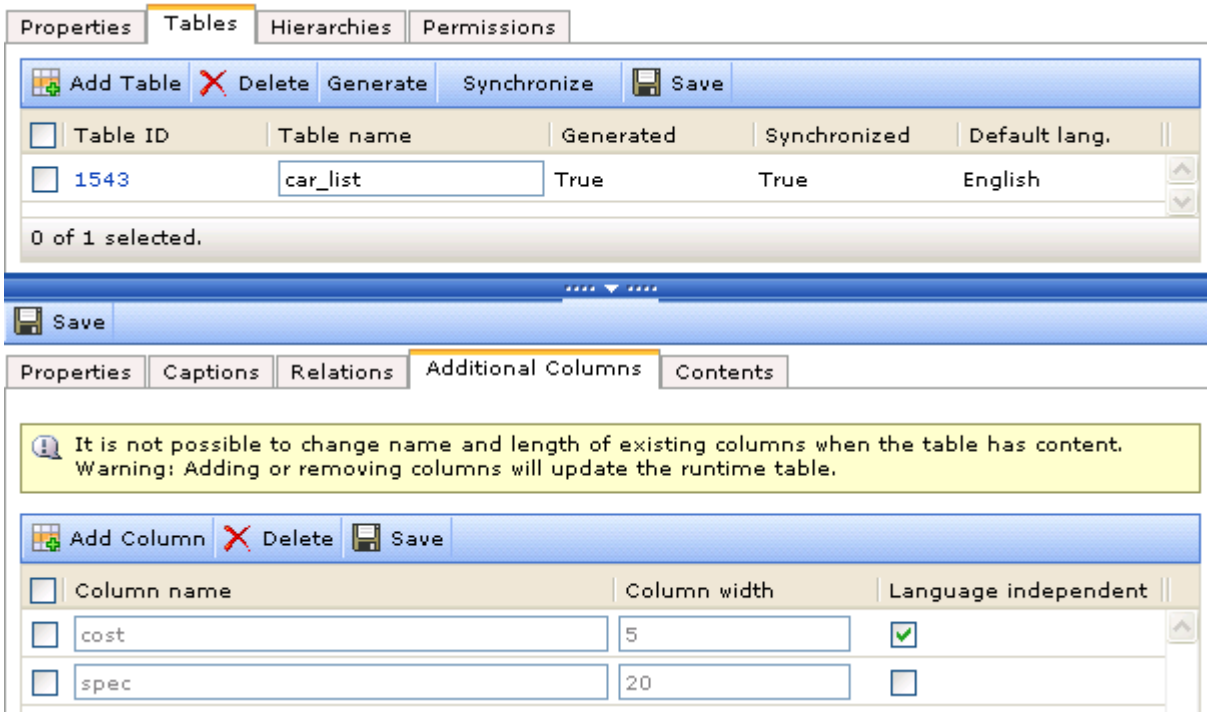
```
GetDBColumnValue(schemaID, tableID, keyID, additionalColumnName,
[languageID])
```

where:

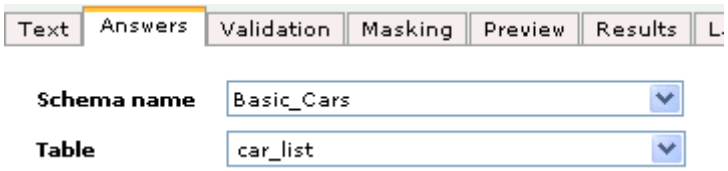
schemaID is the ID of the schema in database designer.
tableID is the ID of the table.
keyID (string) is the ID of the value being searched on (ID row from the contents of the table).
additionalColumnName is the name of the column that the data is to be pulled from.
[languageID] is the ID of the language to be retrieved from. This is optional; if not specified it uses the respondent's language for this survey.

This function can be used to programmatically retrieve values from additional columns that are associated with answers for a table lookup list defined within database designer. For example a table lookup list may be defined in the following way:

The database schema “Basic_Cars” (schema ID 805) contains table “car_list” (table ID 1543) with a list of cars, each containing their own unique string key value. Additionally there are 2 columns “cost” and “spec” containing background information relating to the specific cars.



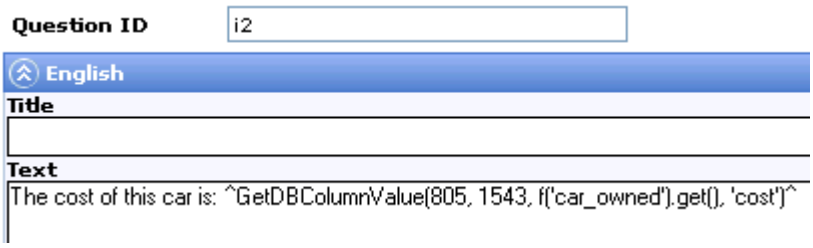
During the survey, the author would like to pipe in the answer to “cost” and store the value of “spec” in a hidden question. If table lookup question named car_owned is based on this table lookup as follows:



Piping in the value of the cost of the selected car can be achieved using the following syntax:

```
^GetDBColumnValue(805, 1543, f('car_owned').get(), 'cost')^
```

as shown here:



Similarly, if the value for spec for the English language (language ID is 9) is to be stored in the hidden question c_spec, the following script syntax can be used:

```
f('c_spec').set(GetDBColumnValue(805, 1543, f('car_owned').get(),
'spec', 9))
```

If the keyID value is supplied explicitly, this must be contained within quotes as this is a string value. For example, to store the value of keyed 7, the following syntax can be used:

```
f('c_spec').set(GetDBColumnValue(805, 1543, '7', 'spec', 9))
```

Note: The GetDBColumnValue() function is not executed during a Random Data Generator run. In this case the value returned will be empty.

10.1.6.2. GetAdditionalColumnValue

This is a survey engine scripting function used to retrieve data from an additional column or columns associated with a specific table inside of a table lookup list.

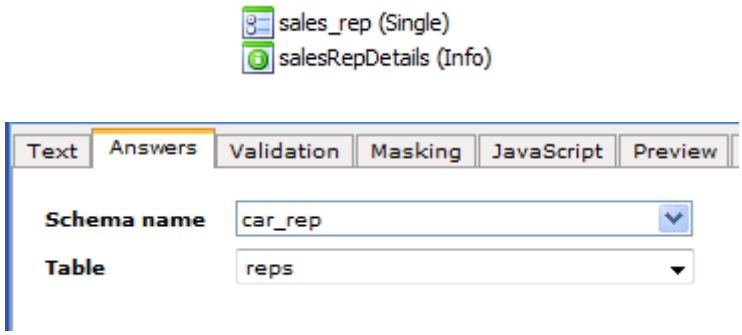
```
f('qid').GetAdditionalColumnValue(code, additionalColumnName)
```

where:

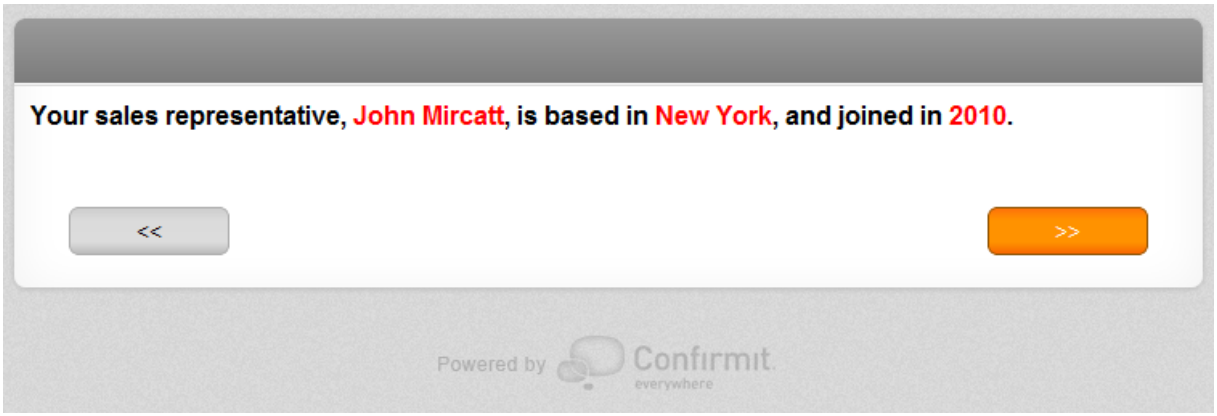
code is the id to query from the lookup table.
additionalColumn name is the name of the additional column from which data is returned.

This will return the data in the specified additional column name in the table lookup for the question “qid” for the code supplied, in the same language the interview is being completed in.

In this example where there is a single question “sales_rep” based on a database lookup “car_rep”:



The “salesRepDetails” info node contains the following piping of the additional columns “office” and “joined_year”:



This is achieved using the following syntax:

```
Your sales representative, ^f('sales_rep')^, is based in
^f('sales_rep').GetAdditionalColumnValue(f('sales_rep').get(),"office"
)^, and
joined in
^f('sales_rep').GetAdditionalColumnValue(f('sales_rep').get(),"joined_
year")^.
```

Note: In most cases the code value supplied in the function will refer to the f(‘qid’).get() for the question that the lookup is based on.

You can perform a test to check whether the code exists in the table used as qid’s answerlist. If the code doesn't exist in the table lookup then the method will return "undefined".

To test if the code exists in the table lookup, and display an error message if not, add the following:

```
if(f('qid').GetAdditionalColumnValue(code,additionalColumnName)==undef
ined)
{
 RaiseError();
 SetQuestionErrorMessage(LangIDs.en,"Error – that code doesn't exist
in the lookup table");
}
```

10.1.7. CAPI and CATI Specific Functions

CAPI (Computer-Assisted Personal Interviewing) and CATI (Computer-Assisted Telephone Interviewing) are Confirmit add-ons. The Redo function described below operates for surveys in CAPI and CATI modes. The rest of the functions described in this chapter operate for surveys running in CATI mode.

10.1.7.1. Redo

Redo() force a redo of a specific question:

```
Redo('qid')
```

where qid is the question ID of the question that is to be re-done.

Redo can be used together with SetErrorMessage to provide a redo-context sensitive error message (this can be defined per language). Example:

```
Redo('q1')
SetErrorMessage(LangIDs.en, 'Error Message')
```

Considerations regarding use of Redo:

- Redo can only be used for surveys that use the optimized database format.
- Redo can only be used in CATI or CAPI survey channels. If, however, a survey is running in the CATI channel then Redo is available for CAWI interviewing also. No error will be reported at launch time if the survey contains an 'unsupported' Redo , instead the survey will produce an internal error at runtime.
- The question id has to be a question that the interviewer has seen before (i.e. a question that has already been processed).
- The question id can be a question inside a callblock. If the callblock has been called multiple times, the the most recent call the callblock will be chosen.
- The question id can be a question inside a loop. The first iteration of the loop is chosen.
- The function is supported by quick test.

Invoking the Redo() will cause the engine to return to the specified question ID. Having then answered this question the engine will fast forward back to the script that invoked the Redo. If during fast forward the interview cannot progress due to a change in the interview path or due to some validation the interview will continue from that point.

Important
When fast forwarding through the project, the Redo() WILL be re-executed. Therefore an example such as:
q1 - Gender question
Script: Redo('q1')
will produce a loop, repeating q1 over and over again. This is intentional behavior. It is the survey author's responsibility to use the function in a way that takes this into account. The Redo is intended to be based upon a certain condition that must be resolved to bypass the Redo, so it will typically be placed within a condition or similar.

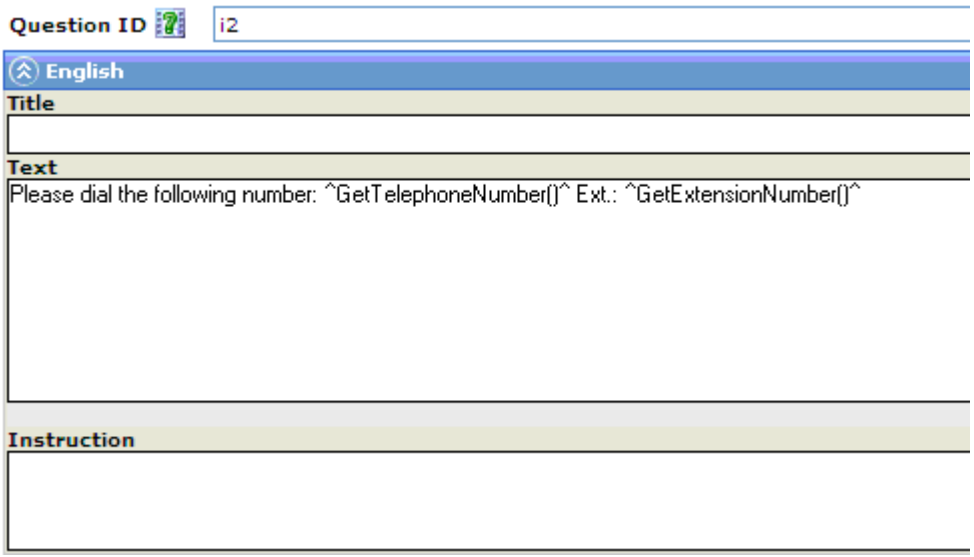
10.1.7.2. GetTelephoneNumber and SetTelephoneNumber

Two functions exist for dealing with the telephone number of the respondent in CATI surveys. This can either be to access the value of the respondent's telephone number, or to explicitly set the value of the respondent's telephone number. The telephone number is a special field that is displayed in the survey's call list in CATI supervisor as well as on the manual selection screen for interviewers.

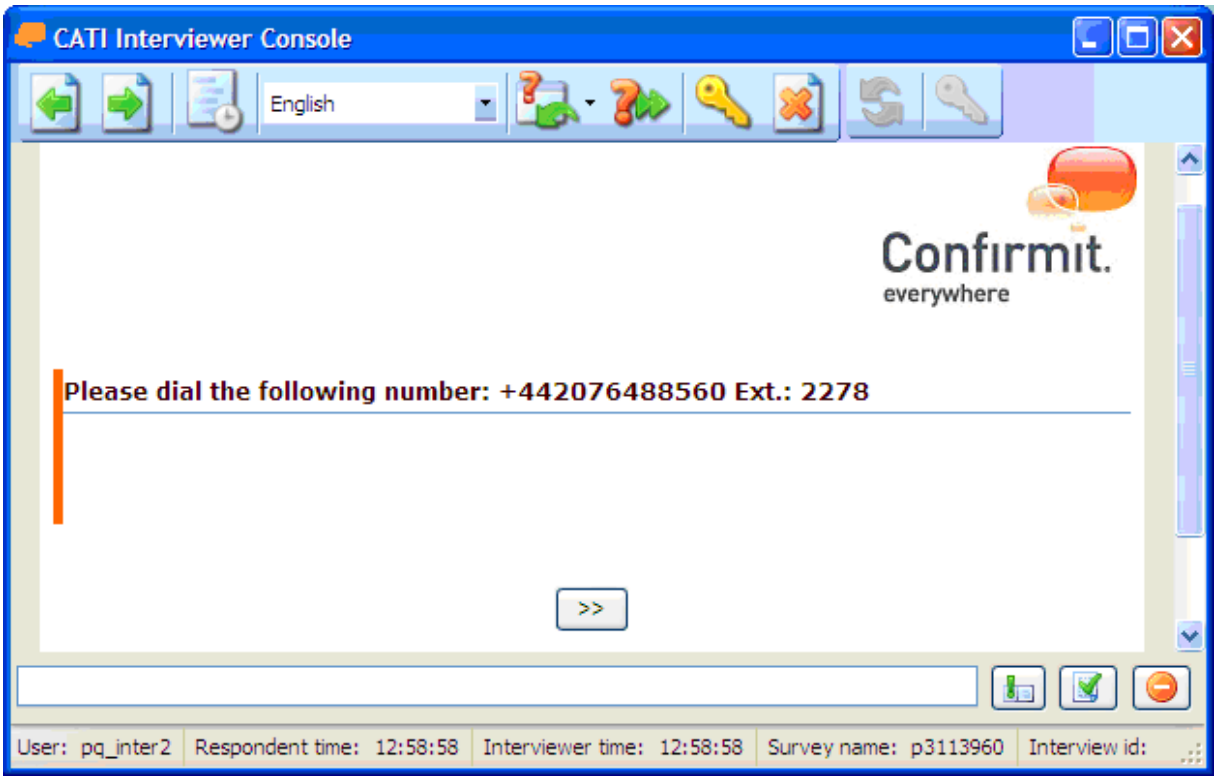
```
GetTelephoneNumber()
```

will return the value of the current respondent's telephone number (string).

An example of retrieving the value of the respondent's telephone number may be as part of an initial contact screen, such as the info node below:



This would be represented as follows in the CATI interviewing Console:



Similarly the telephone number can be set inside a script node in authoring as follows:

```
SetTelephoneNumber(value)
```

Where value is the (string) value that is to be assigned as the respondent's telephone number.

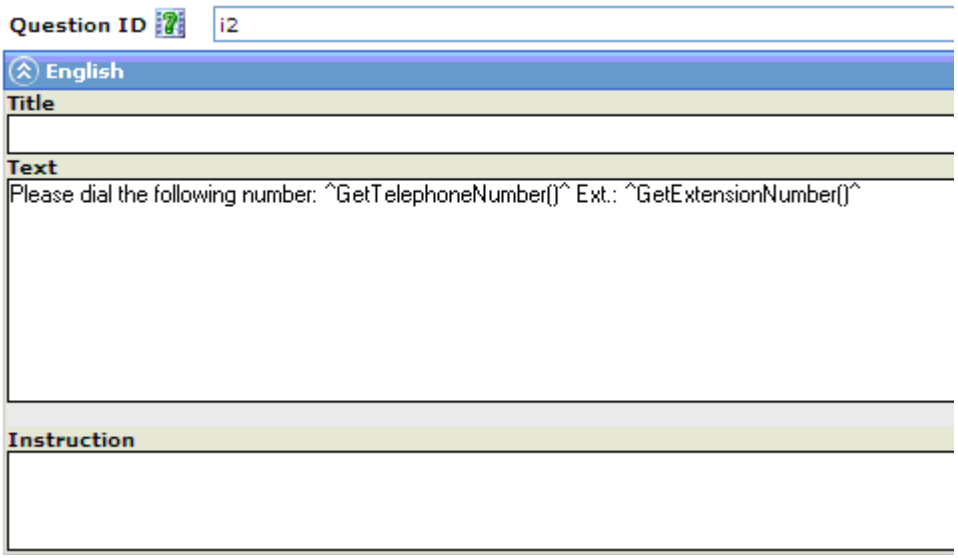
10.1.7.3. GetExtensionNumber and SetExtensionNumber

Two functions exist for dealing with the telephone number extension of the respondent in CATI surveys. This can either be to access the value of the respondent's telephone number extension, or to explicitly set the value of the respondent's telephone number extension.

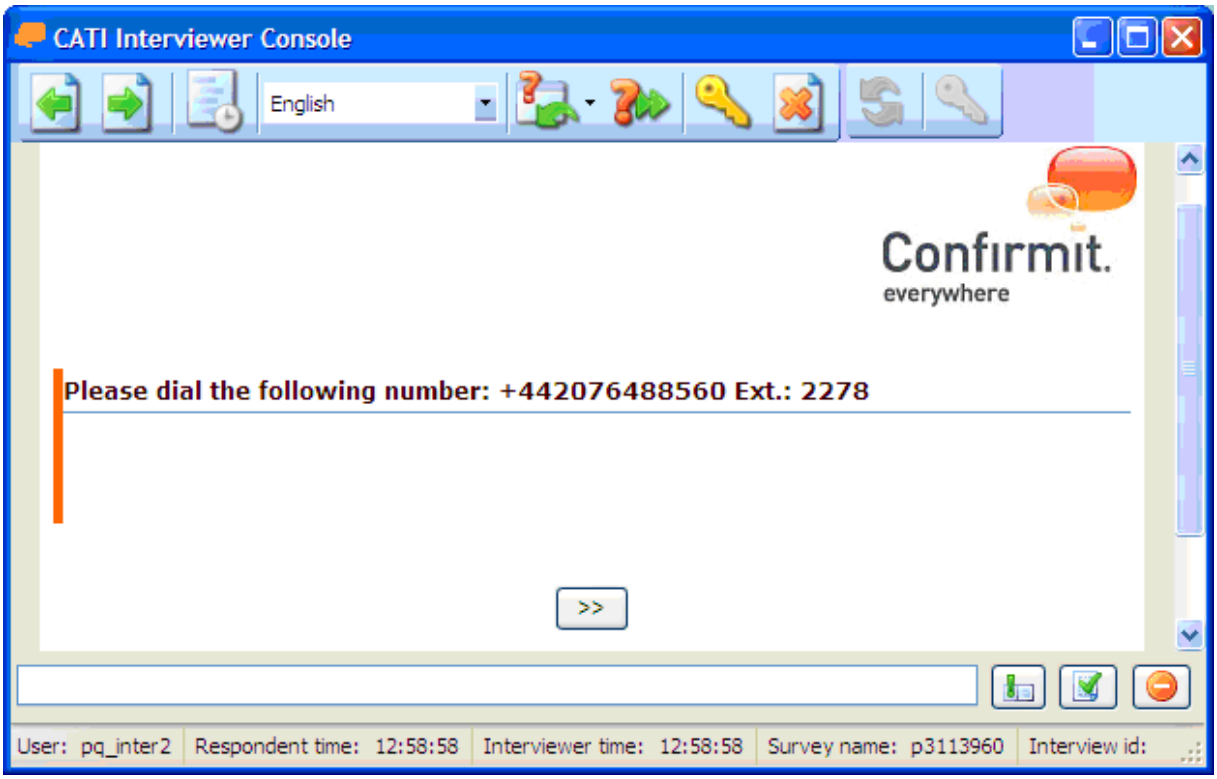
```
GetExtensionNumber()
```

will return the value of the current respondent's telephone number (string).

An example of retrieving the value of the respondent's telephone number extension may be as part of an initial contact screen, such as the info node below:



This would be represented as follows in the CATI interviewing Console:



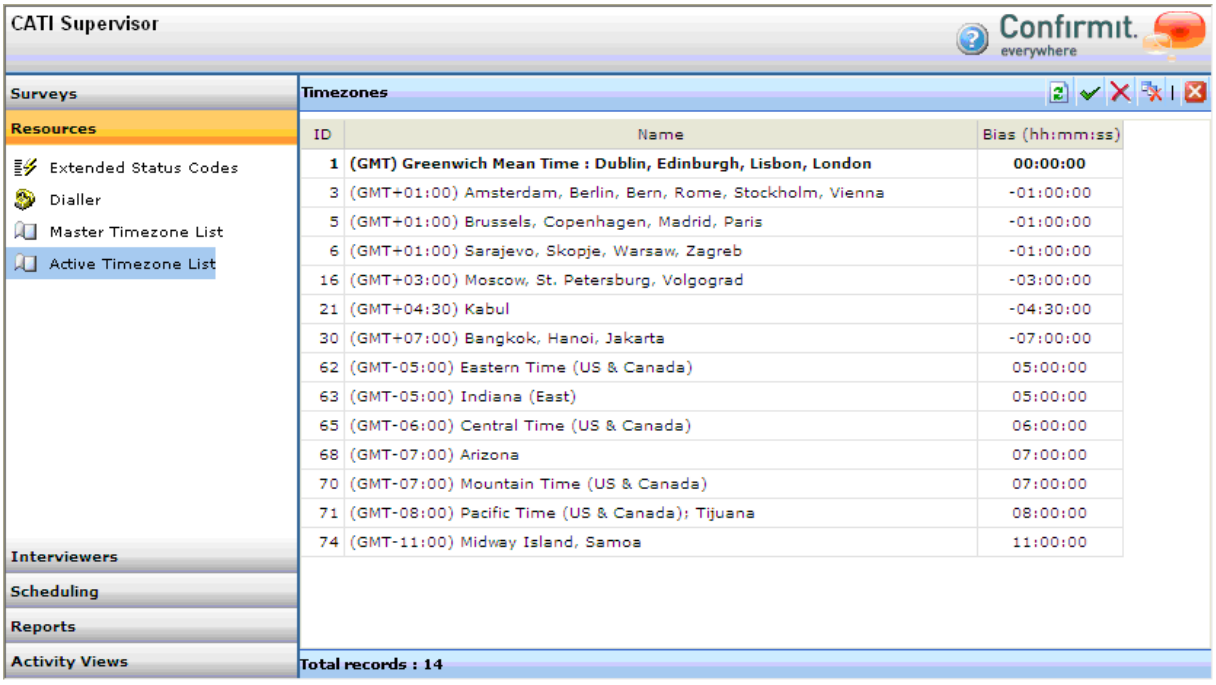
Similarly the telephone number extension can be set inside a script node in authoring as follows:

```
SetExtensionNumber(value)
```

Where value is the (string) value that is to be assigned as the respondent's telephone number extension.

10.1.7.4. GetTimeZoneId and SetTimeZoneId

Two functions exist for dealing with the time zone that the respondent resides in CATI surveys. It is possible to either retrieve or set a value for the respondent's time zone. The time zone associated with the respondent corresponds to the time zone list from the resources tab inside of the CATI supervisor. An example list is as follows:



ID	Name	Bias (hh:mm:ss)
1	**(GMT) Greenwich Mean Time : Dublin, Edinburgh, Lisbon, London**	**00:00:00**
3	(GMT+01:00) Amsterdam, Berlin, Bern, Rome, Stockholm, Vienna	-01:00:00
5	(GMT+01:00) Brussels, Copenhagen, Madrid, Paris	-01:00:00
6	(GMT+01:00) Sarajevo, Skopje, Warsaw, Zagreb	-01:00:00
16	(GMT+03:00) Moscow, St. Petersburg, Volgograd	-03:00:00
21	(GMT+04:30) Kabul	-04:30:00
30	(GMT+07:00) Bangkok, Hanoi, Jakarta	-07:00:00
62	(GMT-05:00) Eastern Time (US & Canada)	05:00:00
63	(GMT-05:00) Indiana (East)	05:00:00
65	(GMT-06:00) Central Time (US & Canada)	06:00:00
68	(GMT-07:00) Arizona	07:00:00
70	(GMT-07:00) Mountain Time (US & Canada)	07:00:00
71	(GMT-08:00) Pacific Time (US & Canada); Tijuana	08:00:00
74	(GMT-11:00) Midway Island, Samoa	11:00:00

The first 'ID' column corresponds to the respondent's time zone Id.

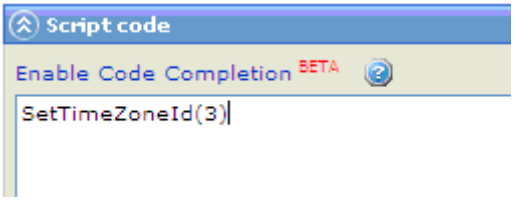
```
GetTimeZoneId()
```

will return the numeric Id stored for the current respondent.

It is also possible to define the time zone for the respondent; this can be achieved using a script node. Here

```
SetTimeZoneId(value)
```

can be used to set the value of the time zone, where value is an valid integer time zone Id value. In the following example the respondent's time zone is changed to time zone Id 3 (GMT+1).



10.1.7.5. GetExtendedStatus and SetExtendedStatus

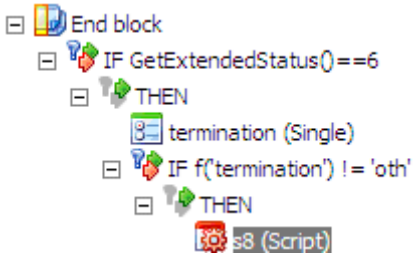
The Extended Status is the status primarily used for CATI interviewing that stores the call outcome of the interview record. The extended status is a numeric value corresponding to the extended status list within the CATI Supervisor. An example list is shown below:

ID	
1	Appointment
2	Busy
3	No reply
4	Quota failure
5	Refusal
6	Terminated
7	Answer phone
8	Modem
9	Fax
10	Congestion
11	Unobtainable

```
GetExtendedStatus()
```

will return the value of the current interview's extended status value (integer). If no value is set 0 will be returned.

A typical example of where this may be used is in the End block of a CATI interview. When an interviewer terminates the interview during an interview, the interview is immediately given a GetExtendedStatus() value of 6. The End block can then be used to react to this termination. In the example below, for interview terminations the type of termination is sub-classified by a termination question and then the interview is further dispositioned to a new extended status value depending on the termination type:

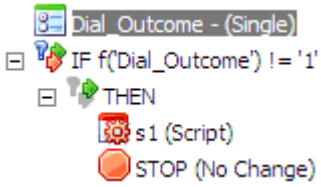


It is also possible to define the extended status value for an interview. This can be achieved using a script node. Here,

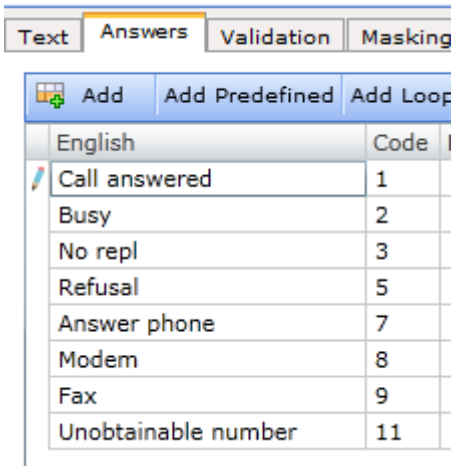
```
SetExtendedStatus(value)
```

can be used to set the value of the extended status, where 'value' is a valid integer extended status value; an integer between 1 and 120.

The initial contact screen that is used to record the call-outcome of the call demonstrates the use of this function. Here we can see the Dial_Outcome question where the call-outcome is recorded:



This contains the following answer list:



The codes here correspond to the extended status code list within the CATI supervisor. Therefore the script node s1 is used to disposition the interview with a SetExtendedStatus() value corresponding to the code for the selected category in the Dial_Outcome question.

10.1.7.6. GetLastInterviewStart

```
GetLastInterviewStart()
```

GetLastInterviewStart returns the date/time of the previous call attempt made to this respondent.

10.1.7.7. GetLastChannelId

```
GetLastChannelId()
```

GetLastChannelId returns the ID (integer) corresponding to the channel used during the last interview attempt

ID	Channel
0	None
1	Web
2	CAPI
4	Random Data Generator
8	CATI

10.1.7.8. GetCatiInterviewerId

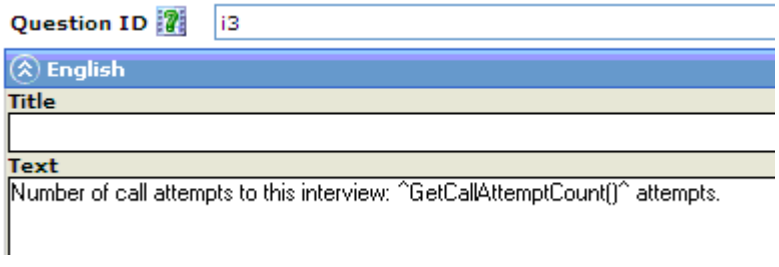
```
GetCatiInterviewerId()
```

GetCatiInterviewerId returns the ID corresponding to the CATI interviewer as defined by the ID column in the CATI interviewer list in the CATI supervisor.

10.1.7.9. GetCallAttemptCount

```
GetCallAttemptCount()
```

The GetCallAttemptCount function is used to return the number call attempts made to this respondent. For example the following info node would display the number of attempts made:



This function is for CATI only, not CAPI.

10.1.7.10. GetTotalAttempts

```
GetTotalAttempts()
```

The GetTotalAttempts function is used to return the number of attempts made to this respondent in either CATI or Web interview modes. For example,

```
^GetTotalAttempts()^
```

used in an info node would display the total number of attempts made to this respondent in either CATI or Web interviewing.

Note: This function is not available for CAPI interviewing and is only available for surveys that have the CATI channel enabled.

10.1.7.11. GetDialMode and SetDialMode

Two functions exist for dealing with the dial mode in use when a company has the CATI Telephony add on enabled. This can either be to access the value of the dial mode or to explicitly set the value of the dial mode.

```
GetDialMode()
```

will return a numeric value corresponding top the list below.

1 = Manual

2 = Preview

3 = Automatic

4 = Predictive (requires the additional predictive telephony add on)

Similarly the dial mode can be set inside a script node in authoring as follows:

```
SetDialMode(value)
```

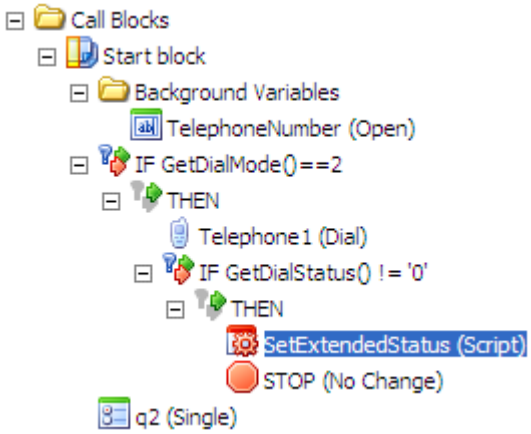
Where value is either 1, 2, 3 or 4 based on the list above.

10.1.7.12. GetDialStatus

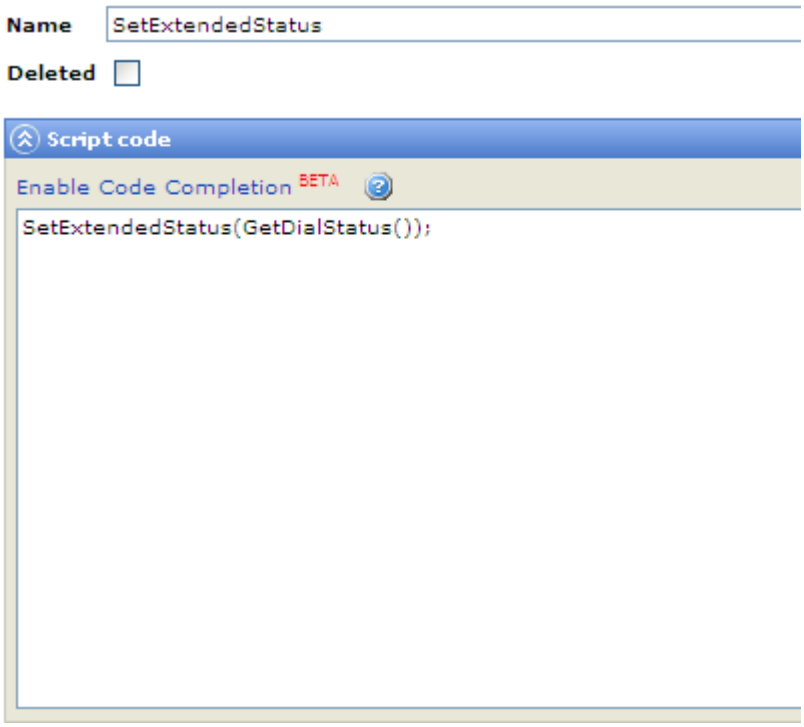
A function exists that returns the outcome of a dial attempt made by the dialer (when a company has the CATI telephony add on enabled). The GetDialStatus() function call will return a numeric value corresponding to the outcome list below.

Dial status value	Description
0	Connected call
2	Busy
3	No reply
7	Answer phone
8	Modem
9	Fax
10	Congestion
11	Unobtainable
12	Nuisance
15	Returned not dialed
25	Returned dialer expired
28	Stopped
29	Telephony failure
30	Error

Typically this would then be used to control either continuing the call or finishing the interview, as shown in the screens below:



With the script doing the following:

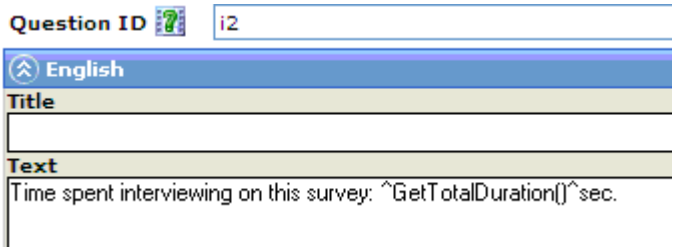


Note: The GetDialStatus() function should not be called before the dial command has been issued for this interview. If it is, it will return the last DialStatus value from the previous call, or a "0" if it is the first call. For surveys using more than 1 dial mode (for example a predictively dialed survey that has preview in predictive calls), ensure that appropriate logic surrounds the function such that it is only retrieved once the dial command has been issued.

10.1.7.13. GetTotalDuration

```
GetTotalDuration()
```

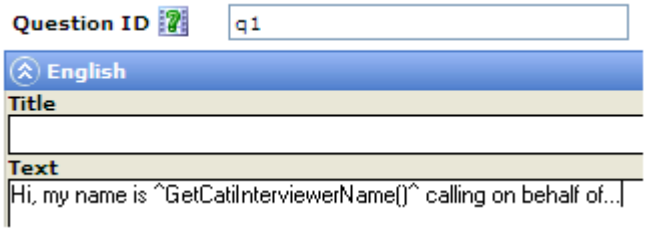
The GetTotalDuration function is used to return the number of seconds spent interviewing cumulatively across all interview attempts. For example the follow info node would display the amount of time spent on this interview across all interview attempts:



10.1.7.14. GetCatiInterviewerName

```
GetCatiInterviewerName()
```

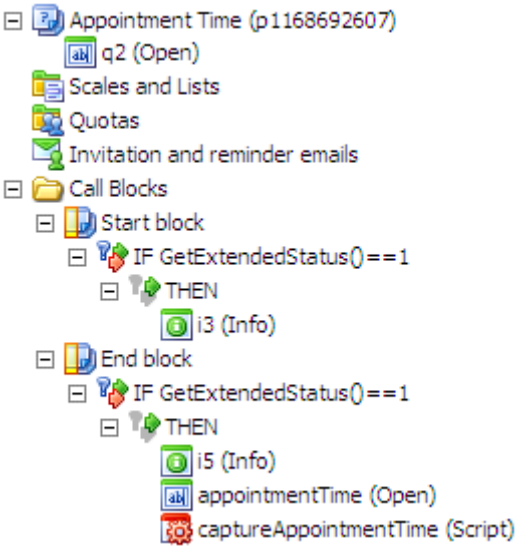
`GetCatiInterviewerName` returns the username of the currently logged on interviewer. This is the Login column in the CATI interviewer list in the CATI supervisor. This is typically used during the initial greeting with the respondent, for example as follows:



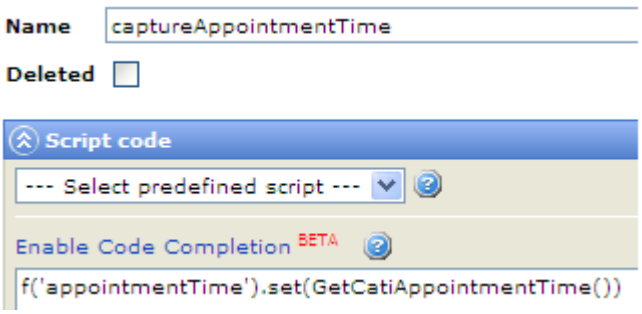
10.1.7.15. GetCatiAppointmentTime

```
GetCatiAppointmentTime()
```

`GetCatiAppointmentTime` returns the time (in local respondent time) for the CATI appointment that has just been made. This is designed to be used in the Exit block at the end of the current interview only. For example the following survey would display the time and also capture it to a hidden variable to be used at the start of a subsequent call. An example of such a survey is:



Where the `captureAppointmentTime` script contains the following:



This is then piped in during the Start block in the i3 info node.

10.1.7.16. GetCatiRespondentUrl

This is GetRespondentUrl for CATI.

Using a redirect inside a CATI survey that jumps to a specific question in the survey (for example *Redirect(GetRespondentUrl("q1")*) results in the interview continuing in Web mode with a number of negative implications because GetRespondentUrl is not supported in CATI. The GetCatiRespondentUrl() function has the same signatures as GetRespondentUrl() and is intended for use in CATI interviewing to safely jump to a specific question.

The same rules apply as for GetRespondentUrl() (see GetRespondentUrl on page 77 for more information).

Note: The GetCatiRespondentUrl() function must ONLY be used in conjunction with redirects to questions inside of the same CATI survey. The GetCatiRespondentUrl() function only applies in the CATI interviewing mode. If it is used in other modes it behaves like GetRespodnentUrl().

10.1.7.17. StartVoiceRecording and StopVoiceRecording

```
StartVoiceRecording('label')
```

`StartVoiceRecording` is only applicable when connected to a supported Confirmit Local Dialer. When issued inside of a script node, sectional voice recording will begin. This will continue to run until a StopVoiceRecording() function call is made of the interview finishes. It is not possible to have concurrent whole interview recording and sectional recording, if whole interview recording is enabled when a StartVoiceRecording() function call is made the whole interview recording will be stopped. The text label passed is incorporated into the filename of the .wav sound file created.

```
StopVoiceRecording(stopRecordingMode: String)
```

Where:

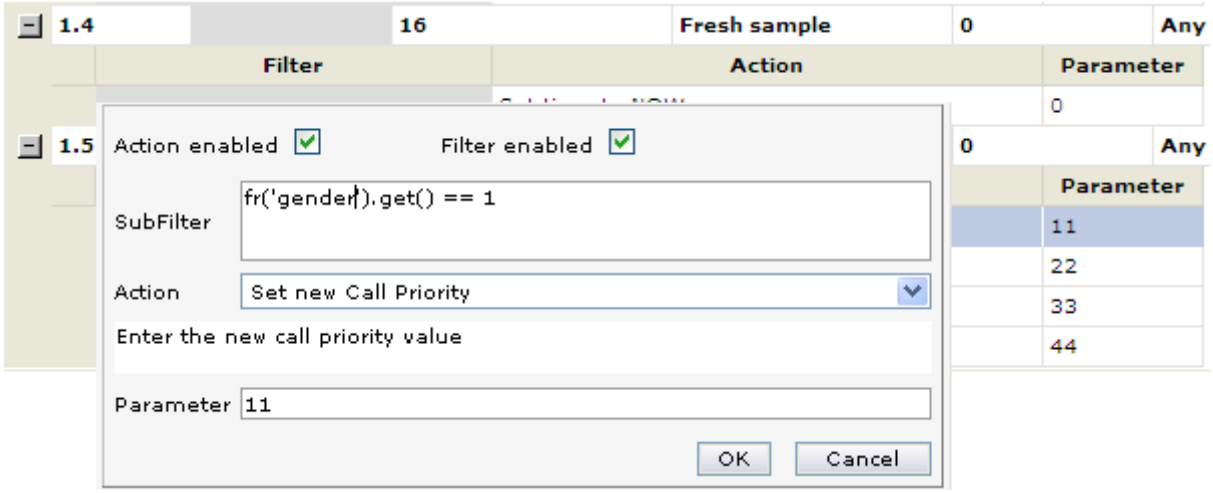
`stopRecordingMode` is an optional parameter that can either be "WholeInterview", "Sectional" or "Both".

`StopVoiceRecording` is only applicable when connected to a supported Confirmit Local Dialer. When issued inside of a script node, if voice recording is enabled for this interview then the specified voice recording type will be stopped on the dialer, if no value is supplied the default is “Both”.

10.1.7.18. fr

```
fr()
```

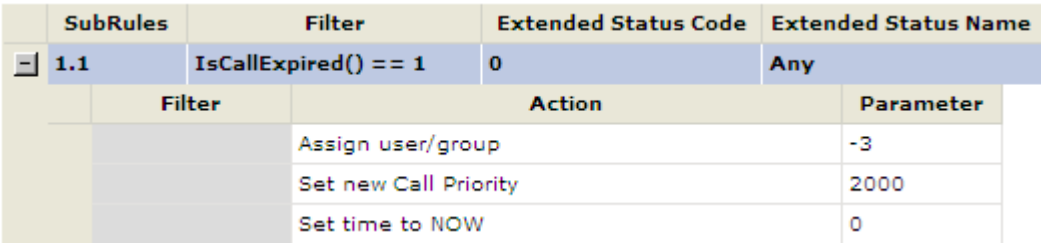
This function is only available inside of survey scheduling inside of the CATI supervisor. This can be used to reference the value of CATI replicated background variables inside of a scheduling script. The fr function is necessary if during sample loading you want to include filters that refer to background variables. Only variables with the Available as CATI filter question-level property can be referenced. The following demonstrates how a filter can be used when referring to the value of the CATI replicated variable named gender:



10.1.7.19. IsCallExpired

```
IsCallExpired()
```

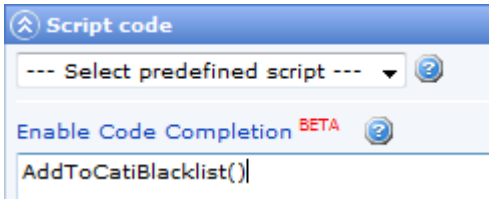
This function is only available inside of survey scheduling inside of the CATI supervisor. CATI scheduling is invoked when a call expired event occurs. Typically, this would be due to an appointment expiry timeout being reached. During scheduling execution the IsCallExpired function returns true if scheduling has been invoked due to a call expiry event. This allows a new call to be scheduled in this scenario, here the actions are to remove assignment locking, set the call priority to be 2000 and set the time to call to be now.



10.1.7.20. AddToCatiBlacklist

```
AddToCatiBlacklist()
```

This function is available in Authoring for CATI-enabled projects. When this function is called, the current respondent's telephone number (contained within the respondent data variable named TelephoneNumber) will be added to the company's telephone blacklist. The telephone blacklist can be managed from the CATI Supervisor, from the Resources tab > Telephone Blacklist item. When telephone blacklist support is enabled at the survey level in Authoring, any attempt to call one of the blacklisted numbers will not be processed and the respondent extended status will automatically be changed to 'Blacklist'. Duplicate numbers or numbers containing spaces or special characters such as ( ) + # - will not be added to the list. Telephone Blacklist checking is based on exact matches only. The example below adds the current respondent's telephone number to the company telephone Blacklist.



10.1.7.21. GetParamValue

```
GetParamValue(value)
```

This function is only available in survey scheduling in the CATI supervisor. If the scheduling script contains some defined parameters then these parameters can be referenced in filters in scheduling. In this example a scheduling parameter named TimeFrame exists, where ID is 1.

ID	Name	Type	Default Value	Description
1	TimeFrame	Numeric	5	Period of time to recall after

To refer to these parameters, use the GetParamValue(value) function where you can either pass the GetParamValue("ParameterName") or GetParamValue(ParameterId) to reference by name or ID. For example, GetParamValue("TimeFrame") or GetParamValue(1). The example below shows this being used in the filter for the Busy call outcomes.

1.4	GetParamValue(1) == 5	2	Busy	0	Any

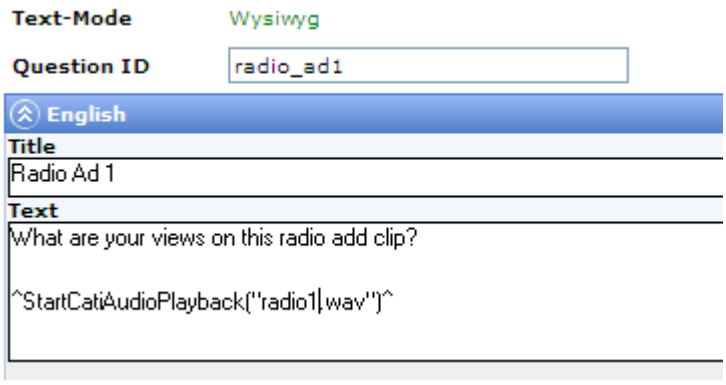
Filter	Action	Parameter
	Recall after number of minutes	5

10.1.7.22. StartCATIAudioPlayback

```
StartCatiAudioPlayback ("filename",[auto_start])
```

This function is only applicable for CATI-enabled projects, in the CATI channel and when working with a supported Confirmit Dialer. When this function is called a command is sent to the dialer to play the specified sound file. The function can be used in any interactive text that will be displayed to the CATI interviewer; typically this would be a question relating to the sound file being played. This function should not be used inside of non-interactive nodes such as script nodes or hidden questions as this will result in nothing being played to the CATI interviewer.

The parameters are the filename (excluding path) of the .WAV file to be played and optionally a Boolean for automatically starting the sound file playback when the function is invoked. The auto_start parameter determines whether the sound file should immediately start playing when the function is invoked, if this is not specified the default value is false (or 0) meaning the file will not start playing immediately. Example of usage is as follows:



In this example, the CATI interviewer console will activate the sound playback buttons on this page allowing the CATI interviewer to play, pause or stop the sound clip, "radio1.wav", it will not start automatically. Using this syntax: ^StartCatiAudioPlayback("radio1.wav", 1)^ upon displaying this page the sound file would immediately be played.

A CATI interviewer can use the play, pause and stop toolbar buttons to control the playback of the audio file and whenever the audio file is playing both the interviewer and the respondent will hear it. These buttons are only active for questions that make use of the sound playback capabilities. Sound file playback will automatically be stopped if the CATI interviewer submits the current page.

Note: It is the responsibility of the survey author or the person who manages the local dialer to ensure that the sound file has been placed in the appropriate directory on the dialer(s). This task is not automated in any way by the Confirmit system. If the file cannot be located and played by the dialer the interview will continue as if the there was no request to play a sound file.

10.1.7.23. IsInOpenendReviewMode

```
IsInOpenendReviewMode()
```

This function is only applicable for CATI-enabled projects, being interviewed in the CATI channel. IsInOpenendReviewMode returns true if the function is executed during CATI openend reviewing, false otherwise. You may use this function to prevent questions from appearing during openend reviewing either as part of a question mask or from condition logic. This would typically be used for open text questions that do not require review at the end of the CATI interview.

Note: If you use this masking, remember to check the "No cleaning on question masking" property (refer to the Authoring User Guide for more information). If this property is not checked, then the data will be cleared if the page is empty.

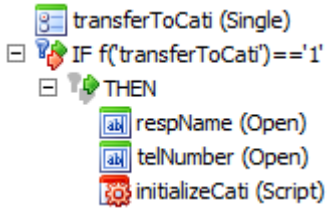
10.1.7.24. AddRespondentToCati

```
AddRespondentToCati(extendedStatus)
```

This function is only applicable for CATI-enabled projects, and applies to interviews being conducted in the Web interviewing mode. It is designed for open surveys where respondent data has not been loaded into the system and the Web respondent needs to continue the survey in the CATI interviewing channel. When this function is called the current respondent will be initialized in the CATI system and be available to be interviewed on in the CATI interviewing channel. If a record already exists for that respondent ID in the CATI interviewing channel the existing respondent will be updated and have the specified extended status value allocated to it.

The extendedStatus parameter that is supplied is the numeric value of the extended status (between 1 and 120) that will be assigned to the respondent in the CATI system; call scheduling will be executed on this call using this extended status value.

The following example demonstrates how this could be used in an open Web survey:



If the answer to the question in “transferToCati” is “Yes” (code 1) then the respondent name and telephone number is collected and initialized via the script node “initializeCati”. The call is then initialized in the CATI interviewing channel with an extended status value of 21 (“Transfer to CATI”). The script contains the following code:

```
SetRespondentValue("respondentname", f("respName"));

SetTelephoneNumber(f("telNumber"));

AddRespondentToCati(21);
```

Note: You must ensure that a scheduling rule is defined to handle calls with the specified extended status to ensure that an appropriate CATI call is created.

10.1.7.25. CreateCatiAppointment

```
CreateCatiAppointment()
```

This function is available for CATI-enabled projects in Authoring. When this function is called, the appointment dialog will appear on the CATI interviewing console. The interviewer will be able to make an appointment and on clicking **OK** the survey will finish with an appointment status. Clicking **Cancel** will allow the interviewer to continue the interview without making an appointment.

Note: This function is only effective if there is an interactive question located after it in the survey. If there are no interactive questions in the survey after this function is called then the function will be ignored, the appointment dialog will not appear and the interview will merely continue until it is finished.

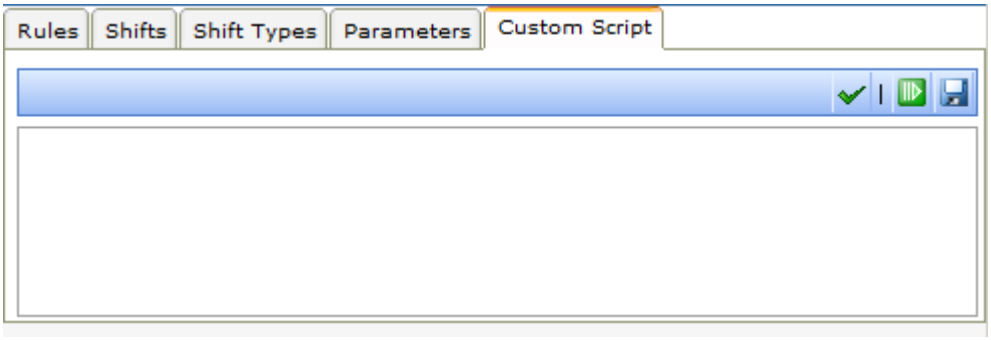
10.1.7.26. Writing a Custom Scheduling Script Code

In the event you wish the scheduling script to perform operations which cannot be configured using the regular CATI Supervisor functions available through the user interface, you can write the script code manually. This requires a thorough knowledge of the JavaScript.NET language.

The script code you enter using this tab is executed only when the “Run specified script” action is enabled.

To create a custom scheduling script manually:

1. With the scheduling script opened in the View mode, go to the Custom Script tab in the lower-right frame.



   This tab contains a single text field, which you use to enter your script code.

2. Enter the script code (JavaScript.NET language is used) in the text field. You can also paste the clipboard contents into this field. Click the **Parse** button in the lower-right frame's toolbar to check the code is correct.

(see Accessing the Call Object in Custom Scripting on page 106 for more information).

10.1.7.26.1. Accessing the Call Object in Custom Scripting

This topic describes how the call object can be accessed in custom scripting, inside of CATI scheduling. This functionality is available through the "Scheduling" object which is available in custom scripts. The "Scheduling" object has the following properties:

Type	Name	Description
BvSurveyEntity	Survey	This object provides data for the survey which contains the current interview. ReadOnly.
BvInterviewEntity	Interview	This object provides data for interview which is scheduled. Read/Write.
BvCallEntity	LastCall	If the scheduling script is run for interview which previously had a call, then this object contains info about the call, otherwise null. ReadOnly.
BvCallEntity	NewCall	If the scheduling script creates a call, this object provides info about the new call. If the object is null, call will not be created upon scheduling completion. Read/Write.
DateTime	Time	Scheduling time.
ShiftService	Shifts	This object provides for shift functionality.

A function exists to initialize a new call inside of the scheduling object when used via a custom scheduling script: `CallShouldBeCreated()`

Once initialized Scheduling.NewCall will be initialized and available.

To cancel the creation of the new call, set Scheduling.NewCall to Null.

Object breakdown

BvSurveyEntity object provides access to survey data as follows:

Type	Name	Description
Int	SID	Internal object ID
String	Name	Project ID
String	Description	Project name
Int	ScheduleID	Scheduling script ID

BvInterviewEntity object provides access to interview data as follows:

Type	Name	Description
Int	ID	ID of interview
String	TelephoneNumber	Respondent telephone number
String	RespondentName	Respondent name
Int	TimezoneID	ID of respondent timezone
Int	TransientState	Extendend status
DateTime	LastCallTime	Last call time
Int	LastCallPersonSID	User ID of last interview
byte	DialingMode	Dialing mode

BvCallEntity object provides access to call data as follows:

Type	Name
int	CallID
int	SurveySID
int	InterviewID
int	Phase
int	RoleID
int	ShiftID
DateTime	TimeInShift
DateTime	TimeToExpire
int	Priority
int	Resource
int	ApptID
int	ResourceType
Guid	RuleNumber

Shift functionality is available through the ShiftService object supporting methods when working with shifts:

- MatchingShift GetExactShift(DateTime utcNowTime, int tzID)
- MatchingShift GetMatchingShift(DateTime utcTime, int tzID)
- DateTime GetMatchingTime(DateTime utcNowTime, int tzID)
- MatchingShift GetNextShift(MatchingShift currentShift, int tzID)
- MatchingShift GetNextShift(MatchingShift currentShift, int tzID, out int countSkipShifts)
- MatchingShift GetNextShiftByID(DateTime utcTime, int tzID, int scriptShiftID)
- MatchingShift GetNextShiftOfSpecifiedType(DateTime utcTime, int tzID, int scriptShiftTypeID)
- MatchingShift GetShiftAfterNumberOfMinutes(DateTime utcNowTime, int tzID, int countMinutes)
- MatchingShift GetShiftAfterNumberOfShifts(DateTime utcNowTime, int tzID, int numberOfShifts)
- MatchingShift GetShiftAfterNumberOfShifts(MatchingShift curentShift, int tzID, int numberOfShifts, bool isTakingExclusionIntoAccount)

Custom code examples

1. Custom script creates new call with priority 10

```
function ScriptFunction()
{
 CallShouldBeCreated();
 Scheduling.NewCall.Priority = 10;
}
```

2. Custom script creates new call with priority which is taken from number variable with 'num_prior' name

```
function ScriptFunction()
{
 CallShouldBeCreated();
 Scheduling.NewCall.Priority = f("num_prior").get();
}
```

3. Custom script creates new call with time to call on next shift

```
function ScriptFunction()
{
 CallShouldBeCreated();
 var shift = Scheduling.Shifts.GetMatchingShift( Scheduling.Time,
1/*Timezone*/ );
 shift = Scheduling.Shifts.GetNextShift( shift, 1/*Timezone*/ );
 Scheduling.NewCall.ShiftID = shift.ShiftTypeID;
 Scheduling.NewCall.TimeInShift = shift.StartDate;
}
```

4. Custom script creates new call and assigns the interviewer with the ID from variable ‘inter’

```
function ScriptFunction()
{
 CallShouldBeCreated();
 var name = f("inter").get();
 Scheduling.NewCall.Resource = PersonRepository.GetByName( name ).SID;
}
```

5. Custom script writes the interviewer's SID to variable 'history' (for all interviewers who have conducted an interview)

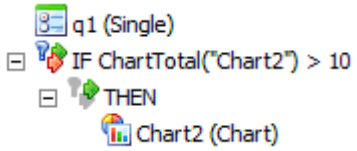
```
function ScriptFunction()
{
 var history= f("history").get();
 var name : String = "";
 if( Scheduling.Interview.LastCallPersonSID != 0 )
 {
 var person =
PersonRepository.GetByID(Scheduling.Interview.LastCallPersonSID);
 if( person != null )
 {
 if( String.InNullOrEmpty(history))
 history = person.name;
 else if( !String.Split( history, ',' ).Any( x => x == person.name ) )
 history = history + "," + person.name;
 else
 return;
 f("history").setValue(history);
 }
 }
}
```

10.1.8. Chart

10.1.8.1. ChartTotal

```
ChartTotal(chartID)
```

ChartTotal returns the current total number of responses in the question the chart with the id chartID is based on. The function is typically used in conditions to prevent showing the chart until a minimum number of responses is reached.



10.1.9. Quota

10.1.9.1. qf

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

```
qf(quotaName)
```

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

If the respondent qualifies for several quotas within *`quotaName`*, `qf` returns `true` if one of these is full, and `false` if none of them are full.

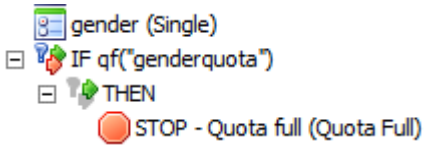
Note: The use of redirects, quotas, status-screened or other solutions with the principle purpose of avoiding the survey "complete" status being reached by a respondent having offered a reasonable amount of responses, is prohibited and will be regarded as an attempt to avoid transaction fee obligations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

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.



Presetting a Quota Question to Check Several Quotas

Note: This example has been superseded by the GetLeastFilledQuotaCodes() function (see GetLeastFilledQuotaCodes on page 111 for more information). You should consider using this function instead of the example given here as it is more efficient.

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

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

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

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

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

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

10.1.9.2. qc and qt

The function `qt` is used to retrieve the target set for a particular quota, and `qc` is used to retrieve the current count.

```
qt(quotaName)

qc(quotaName)
```

`qt` and `qc` both return an integer. They will return target and count for the quota cell in the quota *`quotaName`* corresponding with the current respondent's answers on the questions the quota is based on.

If the respondent qualifies for several quotas within *`quotaName`*, `qt` and `qc` will return `-1`.

Note: Quota functions require a database, so will not work in Quick Test / External Quick Test.

Allocate a Respondent to the Lowest Current Quota Cell

Assume for example that you are performing a survey to evaluate respondents' feelings about five different web page layouts, but you want each respondent to only evaluate one of the layouts. You also want the respondents to be evenly distributed across the five layouts. You could use the following script to select which layout a respondent is shown:

```
var form = f('q1')
var codes = form.domainValues();
var count = 2000; //a value higher than your highest quota target

var code;
var lowestCode;

/*Loops through all the alternatives in q1 and checks whether the
quota count is lowest with that answer*/

for(var i: int=0; i<codes.length; i++) {
  var code = codes[i];
  form.set(code);
  if( qc("quota1") < count ) {
    count = qc("quota1");
    lowestCode = code;
  }
}

form.set(lowestCode);
```

Here "q1" is the question id for the question with an answer per web page layout tested (should normally be a hidden question). "quota1" is the quota id for the quota used in the survey. Now you can create skip logic in the survey based on q1.

Note: It is important that the quota 'quota1' does not have any quota cells set to 'Any' for this example. If that is the case then the respondent will qualify for several quota cells, and the qc() function will then return -1 which will stop the script working as required.

10.1.9.3. GetLeastFilledQuotaCodes

The function GetLeastFilledQuotaCodes is used to retrieve an ordered array of codes for the least filled quota cells for a specific quota.

```
GetLeastFilledQuotaCodes(quotaName, N, [code mask])
```

where

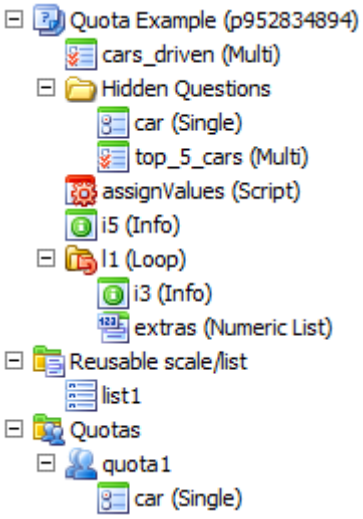
quotaName is the name of the quota to be evaluated.
N is the number of quota cell codes to be returned if available.
[code mask] is an optional code mask that can be applied when considering the quota cells.

The array returned is the codes of the quota cells ordered by least filled first, in percentage terms. If the percentage filled values are equal, then the equal cells are ordered randomly. The function can be used with a quota which is based on one question only, the question can be either a single or multi choice question, but there cannot be more than one question used in the quota. If the quota is based on more than 1 question an empty array is returned.

Note: In test mode, external test mode and through RDG, this function returns an empty array.

For a quota based on multi choice question, answers code(s) without the prefix questionName_ are returned. Quota cells that are already full or that have a target value of 0 are not returned. For a quota based on a single choice question, [Any] cells are not considered and are not retuned by the function. For a quota based on a multi choice question all answers should be [Any] except one which should be [Chosen]; only cells with this structure are considered by the function.

In this example:



There is an interactive multi choice question “cars_driven”, a hidden single choice question “car” and a hidden multi choice question “top_5_cars” has the same answer list. Of the cars selected in “cars_driven” the quota cell that is least filled is assigned to the question “car”, and the top five least filled quota cells are assigned to the question “top_5_cars”. This is achieved using the “assignValues” script node, the contents of this is:

```
//Find the least filled quota cell and assign it

f("car").set(GetLeastFilledQuotaCodes('quota1',
1,f("cars_driven").categories()).toString());

//Find the top 5 least filled quota cells and assign them

f("top_5_cars").set(new Array(GetLeastFilledQuotaCodes('quota1', 5,
f("cars_driven").categories())));
```

The loop “l1” is then masked based on the value assigned to the question “car”, meaning that only the chosen car (least filled quota) will have the loops questions asked.

Note that the ...(new Array()... is required because Confirmit returns a standard .NET array type but "Array" in JScript.NET is different; this then eliminates any possible incompatibility.

Other examples of using this function include:

1. Display the least filled code:

```
^GetLeastFilledQuotaCodes('quota1', 1).toString()^
```

2. Display the length of the array:

```
^GetLeastFilledQuotaCodes('quota1', 6).Length^
```

3. Display the top 3 filtered by a hard-coded array:

```
^GetLeastFilledQuotaCodes('quota1', 3, ['1', '2', '3', '4', '5',
'6']).toString()^
```

Note: The GetLeastFilledQuotaCodes() function is calculated every time the function is called, therefore if it is executed several times in the same interview, the values returned could be different due to either the state of the quotas changing or due to the random selection of cells that are equally full in percentage terms.

10.1.10. Credits in Panels

This section describes the functions that can be used to give points (credits) to panelists. Some functions can be used only in Standard Panels and Professional Panels, and some can also be used in Basic Panels. Basic Panels and Standard Panels and Professional Panels are Confirmit add-ons. If you do not have access to this functionality, please contact your Confirmit account manager for more information.

10.1.10.1. SetPanelistCredit

SetPanelistCredit is used for giving panelists points for registering in the panel and responding to surveys. SetPanelistCredit can be used in Basic Panels, Standard Panels and Professional Panels.

```
SetPanelistCredit(credit, comment)
SetPanelistCredit(credit, comment, sectionId)
SetPanelistCredit(credit, comment, panelistId)
SetPanelistCredit(credit, comment, panelistId, sectionId)
```

All of these update a global credit counter in the panel called CreditBalance which can be found inside the PanelistParticipation folder. The instance of the function that is being used, depends on the number of parameters and their type.

credit is an integer that represents the number of points that is to be added to the balance. *comment* is a String that can be used to include some information about why the points were added. The limit for *comment* is 256 characters.

If you include just *credit* and *comment* as parameters, you set *credit* points to be added to the credit balance from the survey the panelist is responding to. These points will be added as soon as interview status is set to "complete", "screened" or "incomplete" for the respondent. A repeated call to the function in the same survey will overwrite the previous credit assigned for the survey.

If you would like to assign different sets of points for different parts of a survey, you can also include `sectionId`. `sectionId` is a String with maximum 40 characters that can identify different parts of the survey. You can call the sections whatever you like as long as you are within the 40 character limit and do not use Unicode characters. The credits (points) will be active as soon as the respondent's interview status is set. If the panelist does not have any status (=incomplete), a pending flag will be set and the points will not become active until the status changes to for example "complete", "screened" or "quotafull".

If you need to update points for a different panelist, you can do this by including the `panelistId` (integer) of that panelist. This can for example be done to credit panelist's that refer friends to the panel. You may send the panelistid (respid in panelist survey) of the panelist that referred into the survey in the URL and retrieve it with `Request`(see QueryString on page 199 for more information), and then use SetPanelistCredit to credit his/her points balance.

Note: Panelistid must be a valid number/integer. If panelistid is a String, then the current panelist will be used irrespective of the panelistid value.

SetPanelistCredit() can over-write an existing row depending on how the function is called. The full function call, including all arguments is:

```
SetPanelistCredit(credit, comment, panelistId, sectionId)
```

Credit and comment are required arguments. If you wish to make more than one set from a “single survey record”, you will also need to include the sectionId.

Example: Assuming a starting credit value of 5000, run the following script for a specific record in the survey:

```
SetPanelistCredit(-1000,'redemption')
```

At this point you will have a balance of 4000. If you change the script to the following and re-launch and re-enter:

```
SetPanelistCredit(-2000,'redemption')
```

on viewing the balance you will see 3000, not the expected 2000. If you wish to have each expression counted “separately”, you must add a sectionId. This will cause separate rows to be written for each. For example:

```
SetPanelistCredit(-1000, 'redemption','R-20130827170301')
```

In the example, a sectionID of "R-" is specified and then YYYYMMDDHHMMSS is appended. This is pulled from a new Date() object. You can alter your code to send in a sectionID that will be different each time a panelist runs over the script. This ensures that a new row will be written.

Note: If you just want something unique per row, use .getTime() from a new Date() object.

Note: comment is limited to 256 characters and sectionId is limited to 40 characters.

Also note that rows are marked 'pending' until a status (complete, screened, etc) is set for the record.

10.1.10.2. Using Custom Variables with SetPanelistCredit

Custom Variables variables are supported in Standard and Professional Panels.

```
SetPanelistCreditWithCustomVariables(credit, comment, fieldNames,
fieldValues)
SetPanelistCreditWithCustomVariables(credit, comment, sectionId,
fieldNames, fieldValues)
SetPanelistCreditWithCustomVariables(credit, comment, panelistId,
fieldNames, fieldValues)
```

`fieldNames` (array of Strings) are the names of the custom fields in the credit transaction loop in the database to be updated, and `fieldValues` are the corresponding values to set in these fields.

10.1.10.3. GetPanelistCreditBalance

`GetPanelistCreditBalance()` can be used to retrieve the credit balance for a panelist as an integer. You can get either the total balance or the balance within a specific period. GetPanelistCreditBalance is supported in Basic Panels, Standard Panels and Professional Panels.

```
GetPanelistCreditBalance()
```

returns the current total for a panelist as an integer.

With the Date Time parameters `fromDate` and `toDate` you can get the balance for a panelist within a specific period as an integer (the sum of all the transactions within that period), e.g:

```
GetPanelistCreditBalance(fromDate, toDate);
```

Displaying Current Credit Balance in a Survey

If you want to show the panelist their current credit balance in a survey, you can include the following in the text of an info node or question:

```
^GetPanelistCreditBalance()^
```

Calculating Credit Balance for the Last 30 Days

The following function can be used to calculate the credit balance, looking at the last 30 days ("adding" -30 days).

```
function last30()
{
var toDate : DateTime = DateTime.Now;
var fromDate : DateTime = toDate.AddDays(-30);

return GetPanelistCreditBalance(fromDate,toDate);
}
```

10.1.10.4. GetPanelistCredits

The `GetPanelistCredits` function can be used to retrieve a subset of a panelist's transactions, as an array of `PanelistCredits` objects. `GetPanelistCredits` is supported in Basic Panels, Standard Panels and Professional Panels.

```
GetPanelistCredits(topN);
GetPanelistCredits(fromDate,toDate);
GetPanelistCredits(topN, fromDate, toDate, startPosition,
orderAscending);
```

`GetPanelistCredits(topN)` will return an array of PanelistCredits objects (see below), representing the last `topN` transactions. `topN` is an integer.

`GetPanelistCredits(fromDate,toDate)` will return an array of `PanelistCredits` objects (see below), representing transactions performed between `fromDate` and `toDate`. `fromDate` and `toDate` are both of type DateTime.

`GetPanelistCredits(topN,fromDate,toDate,startPosition,orderAscending)` will return an array of `PanelistCredits` objects (see below), representing the first `topN` transactions performed between `fromDate` and `toDate`, if ordered ascending (true for `orderAscending`) or descending (false for `orderAscending`). `topN` is an integer, `fromDate` and `toDate` are both of type DateTime and `orderAscending` is Boolean.

The `PanelistCredits` objects in the array returned have the following properties:

```
pCredit.CreditId
```

`CreditID` (integer) is the unique identifier for a transaction.

```
pCredit.PanelistId
```

`PanelistID` (integer) is the responseid of the panelist the credits are assigned to.

```
pCredit. SecondaryPanelistId
```

`SecondaryPanelistId` (integer) is the responseid of another panelist. This field holds a different panelistid if points have been set from a different panelist's survey (for example for referrals).

```
pCredit.PanelistTransactionId
```

`PanelistTransactionId` (integer) is the id of the transaction, unique to this panelist.

```
pCredit.SectionId
```

`SectionId` (String) is an identifier a survey designer may assign to the credits, for example if assigning credits to panelist at various points in the survey.

```
pCredit.SurveyId
```

`SurveyID` (String) is the project number (pXXXXXX) of the project the credits were assigned from.

```
pCredit.Credit
```

`Credit` (integer) is the number of credits (points) assigned to the panelist in this transaction. A negative number would represent points withdrawn.

```
pCredit.Comment
```

`Comment` (String) is a text assigned to the transaction, for example explaining what the transaction represents.

```
pCredit.Created
```

`Created` (DateTime) is the time the transaction was made (added).

```
pCredit.Pending
```

`Pending` (Boolean) represents whether the transaction is pending (not active) or not. If the transaction is pending, the flag is set to true, if not, it is set to false.

Retrieving and Listing the last 10 Panelist Credits Transactions inside a Survey

The following functions will list out the date, project number and credits (points) of the last 10 credits transactions assigned to the current panelist in an HTML table:

```
function ShowCredits()
{
 var pCredits = GetPanelistCredits(10);
 var txt = "<table>\n"
 txt += "<tr><td>Date</td><td>Project</td><td>Points</td></tr>\n";
 for(var i : int = 0;i<pCredits.length;i++)
 {
  txt+="<tr><td>"+pCredits[i].Created+"</td>";
  txt+="<td>"+pCredits[i].SurveyId+"</td>";
  txt+="<td>"+pCredits[i].Credit+"</td></tr>\n";
 }
 txt += "</table>\n";
 return txt;
}
```

10.1.10.5. Using Custom Variables when Getting Panelist Credits

Custom variables are supported in Standard and Professional Panels.

```
GetPanelistCreditsWithCustomVariables(topN, fieldNames);

GetPanelistCreditsWithCustomVariables(fromDate,toDate, fieldNames);
```

will retrieve panelist credit transactions with the default properties (see GetPanelistCredits on page 114 for more information) and in addition any custom variables listed in fieldNames (array of Strings), either the topN first records or the records between fromDate and toDate.

Retrieving and Listing the last 10 Panelist Credits Transactions inside a Survey, including custom variables

The following functions will list out the date, project number, credits (points) and the two custom variables listed in the array fieldNames, of the last 10 credit transactions assigned to the current panelist in an HTML table:

```
function ShowCreditsWithCustomVariables()
{
 var fieldNames = ['customVariable1', 'customVariable2'];
 var pCredits = GetPanelistCreditsWithCustomVariables(10, fieldNames);

 var txt = "<table>\n"
 txt += "<tr><td>Date</td><td>Project</td><td>Points</td><td>Custom
Variable 1</td><td>Custom Variable 2</td></tr>\n";
 for(var i : int = 0;i<pCredits.length;i++)
 {
 txt+="<tr><td>"+pCredits[i].Created+"</td>";
 txt+="<td>"+pCredits[i].SurveyId+"</td>";
 txt+="<td>"+pCredits[i].Credit+"</td>;
 txt+="<td>"+pCredits[i].CustomFieldValues[0] +"</td>;
 txt+="<td>"+pCredits[i].CustomFieldValues[1] +"</td></tr>\n";
 }
 txt += "</table>\n";
 return txt;
}
```

10.1.11. Standard and Professional Panels

This section describes functions that can be used in registration and profile update surveys linked to a Standard or Professional Panel to add and update panelists.

Standard Panels and Professional Panels are Confirmit add-ons. If you do not have access to this functionality, contact your Confirmit account manager for more information.

Registration and profile update surveys are set up as regular Confirmit surveys, and are linked to the panel database through a setting on the Project Management > Overview page General tab:

This setting will allow scripts in the survey to add new panelists to the panel database and retrieve and update data from the panel database for the current panelist. Very often the registration and profile update surveys will be linked from the Panel Portal.

10.1.11.1. CreatePanelist

In a registration survey linked to a Panel as described below, the person registering to participate in the panel can be asked for email (which will be used as the login name and unique identifier), password and other information. The panelist can be added to the panel database with the `CreatePanelist` function:

```
CreatePanelist(fieldNames, fieldValues)
```

CreatePanelist will return the panelist id (positive integer) if the insert succeed, or a negative integer if the insert of a new panelist fails, for example because a panelist is already registered with the provided email address (see below).

The parameters are both arrays of Strings: `fieldNames` are names of the fields in the panel database to insert data into, and `fieldValues` are the corresponding values to insert into these fields. `fieldNames` needs at least to include "`email`", but if the panel database has other user provided unique fields, these fields need to be included as well.

The codes returned from the insert of a new panelist are listed below. A negative code (less than zero) represents an error and means that the panelist creation did not succeed.

Code	Description
0	The panelist was created successfully. A new panelist record is added to the database.
-1	Error on the survey end, for example not being able to connect to the panel database etc.

-2	The unique constraint is violated, i.e. a panelist already exists with the provided values in the unique fields (email + any user provided unique fields)
-3	Other database errors when inserting the new panelist.
-5	The password failed on one or more of the password complexity rules, according to the Panel settings for password complexity.

Inserting a new Panelist in a Panel

In a registration survey that is linked to a Panel as described above, there is an open text question with question id “email”. In the validation code of this open text question the following script can be included to insert the new panelist in the Panel database, and provide an error message if the insert fails:

```
var fieldNames : String[] = ["email"];
var fieldValues : String[] = [f('Email')];
var result = CreatePanelist(fieldNames, fieldValues);
if(result<0 && result!= -2 && result!= -5)
{
```

RaiseError();

SetErrorMessage(LangIDs.en,"Unknown error: Couldn't register with this email address. Error code:" + result);

```
}
else if(result == -2)
{
```

RaiseError();

SetErrorMessage(LangIDs.en,"Can't register with this email address. It is already in use.");

```
}
else if (result == -5)
{
```

RaiseError();

SetErrorMessage(LangIDs.en,"Password does not comply with password complexity rules. Please choose a different password.");

```
}
```

10.1.11.2. UpdatePanelVariables

If a survey is linked to a Panel as described above, and the panelists are identified either through panelistid being included in respondent table (a survey you have sampled to), because the survey is opened within a protected page (requires login) in the panel portal (profile update), or because you have just inserted this panelist with the CreatePanelist function (registration survey), you may use the `UpdatePanelVariables` function to update variables in the panel database.

```
UpdatePanelVariables(fieldNames, fieldValues)
```

The parameters of the `UpdatePanelVariables` function are both arrays of Strings: `fieldNames` are names of the fields in the panel database to update, and `fieldValues` are the corresponding values to set in these fields.

UpdatePanelVariables will return a negative integer if the update fails:

Code	Description
0	Update successful.
-1	Error on the survey end, for example not being able to connect to the panel database etc.

-2	The unique constraint is violated, i.e. another panelist already exists with the provided value(s) you are trying to set for (one of) the unique field(s) (email + any user provided unique fields).
-3	Other database errors when updating the panelist.
-4	The panelist does not exist.
-5	The password failed on one or more of the password complexity rules, according to the Panel settings for password complexity.

Update Panel Variables in a Panel

Here is an example from a survey linked to a Panel as described above, where the questions gender, dateofbirth, USstate, householdsize, carowner, empstatus, marstatus and email are asked. A hidden open text question status_desc is used to store the status returned from the update call. The following script can be placed in a script node to update these variables in the panel database for the current panelist:

```
var fieldNames : String[] =
["gender","dateofbirth","USstate","householdsize","carowner","empstatu
s","marstatus","email"];
var fieldValues : String[] =
[f('Gender'),f('dateofbirth'),f('USstate'),f('householdsize'),f('carow
ner'),f('empstatus'),f('marstatus'),f('email')];
var result = UpdatePanelVariables(fieldNames, fieldValues);

if(result <0 && result!= -2 && result!= -4 && result!= -5)
 {
 f('status_desc').set("Unknown error. Error code:" + result);
 }
else if(result == -2 )
 {
 f('status_desc').set("Update of email address failed. It is already
in use for another panelist.");
 }
else if(result == -4 )
 {
 f('status_desc').set("Panelist does not exist in the panelist
database.");
 }
else if(result == -5 )
 {
 f('status_desc').set("Password does not comply with password
complexity rules.");
 }
else
 {
 f('status_desc').set("Succesfully updated.");
 }
```

10.1.11.3. GetPanelVariables

If a survey is linked to a Panel as described above, and the panelists are identified either through panelistid being included in respondent table (a survey you have sampled to) or because the survey is opened within a protected page (requires login) in the panel portal (profile update), you may use the `GetPanelVariables` function to update variables in the panel database.

```
GetPanelVariables(fieldNames)
```

Note: If this function is used in a survey, then the survey must be launched as a Limited survey. Failure to do this will result in an error message being generated for each respondent who attempts but fails to enter the survey either because they are not uploaded or because they do not enter via a portal.

`GetPanelVariables` will retrieve values from the fields provided in `fieldNames` (array of Strings) and return these values. The data types will be according to the question types (for example date, numeric, open text etc.).

Retrieving Data for a Panelist in a Panel

The following script can be placed in a script node in the beginning of a profile update survey linked to a panel as described above to retrieve the fields gender, dateofbrith, USstate, carowner, empstatus, marstatus and email and set these values in the corresponding questions in the survey. The script will skip empty values (empty string “” or null), and also shows how true and false for the carowner variable, which is a Boolean in the Panel, are converted to codes 1 and 0.

```
var fieldNames : String[] =
["gender","dateofbirth","USstate","householdsize","carowner","empstatu
s","marstatus","email"];
var variables = GetPanelVariables(fieldNames);
for(var i=0;i<fieldNames.length;i++)
 {
  if(fieldNames[i]=="carowner")
   {
     if(variables[i])
      f(fieldNames[i]).set("1")
     else
      f(fieldNames[i]).set("0")
  }
  else if(variables[i]!="" && variables[i]!=null)
   f(fieldNames[i]).set(variables[i]);
 }
```

10.1.11.4. UpdateSurveyHistoryPanelVariables

The function UpdateSurveyHistoryPanelVariables replaces UpdateSurveyHistoryVariables, and is used for updating the survey history of a panelist directly from a script in the panel survey. It requires that the survey from which it is called is linked to the panel as described previously.

```
UpdateSurveyHistoryPanelVariables(fieldNames, fieldValues)
```

fieldNames and fieldValues are both arrays. fieldNames holds the variable names of the survey history variables you want to update, and fieldValues the corresponding values to assign to them.

Update Survey History Variables in a Panel

This is an example from a survey linked to a Panel as described, where sample has been uploaded from a sample job. The response to a question "surveyrating", asked at the end of the survey, is to be set in the panel's survey history.

```
var fieldNames : String[] = ["surveyrating"];

var fieldValues : String[] = [f('surveyrating')];

UpdateSurveyHistoryPanelVariables(fieldNames,fieldValues);
```

10.1.11.5. UpdateSurveyHistoryVariables (obsolete)

The function `UpdateSurveyHistoryVariables` is used for updating the survey history of a panelist directly from a script in the panel survey, in a similar way to UpdateSurveyHistoryPanelVariables, but requires providing the panelistID and jobNumber in addition to the variables to update and their values. It requires that the survey from which it is called is linked to the panel as described previously.

```
UpdateSurveyHistoryVariables(panelistIdString, jobNumber, fieldNames,
fieldValues)
```

`panelistIDString` (string) is the panelistid (the responseid in the panel database). *`jobNumber`* (string) is the sample job number. *`fieldNames`* and *`fieldValues`* are both arrays. *`fieldNames`* holds the variable names of the survey history variables you want to update, and *`fieldValues`* the corresponding values to assign to them.

It is recommended that you use UpdateSurveyHistoryPanelVariables instead, as that is simpler. However UpdateSurveyHistoryVariables is still supported.

10.1.11.6. IsFromCommunityPortal and GetCommunityPortalReturnUrl

When the registration and/or profile update survey is opened from the Panel Portal, it will be opened within a frame in the Panel Portal. The functions `IsFromCommunityPortal` and `GetCommunityPortalReturnUrl` are provided to seamlessly handle redirecting back to the Panel Portal for panelists that have opened the survey from within the Panel Portal.

```
IsFromCommunityPortal()
```

`IsFromCommunityPortal()` returns `true` if the survey is opened from the Panel Portal and `false` if not. The function can be used to determine when to redirect to the Panel Portal Url provided by the `GetCommunityPortalReturnUrl` function.

```
GetCommunityPortalReturnUrl()
```

`GetCommunityPortalUrl()` returns a URL that can be used to redirect back to the Panel Portal, with authentication (so that the panelist does not need to log on again if he/she was already logged in).

Redirect back to Panel Portal

The following script can be placed in a script node at the end of a survey to redirect back to the Panel Portal for panelists that open the survey from the Panel Portal:

```
if(IsFromCommunityPortal())
{
 Redirect(GetCommunityPortalReturnUrl());
}
else
{
 Redirect("http://www.confirmit.com");
}
```

10.1.11.7. IsFieldValueTaken

In Professional and Standard Panels it is possible to set panel variables to be unique. That means that each panelist must have a unique value in this field. The isFieldValueTaken function can be used in panelist recruitment surveys to check if any previous respondents have selected the same value in a given field.

```
isFieldValueTaken(fieldName, value)
```

isFieldValueTaken returns true if there is already a panelist in the sample with the provided value (string) in the question with question id fieldname (string), false otherwise.

It can be used in the validation code of the question defined in "uniqueid" in a panel survey:

```
if(isFieldValueTaken("uniqueid", f("uniqueid").get())) {
```

SetQuestionErrorMessage(LangIDs.en,"The value in uniqueid is already taken. Please input a different value.");

f("uniqueid").set("");

RaiseError();

```
}
```

The function can also be used in the same way in ordinary projects, to check that the value entered in a question with a given question id is unique.

If the function is used in ordinary projects, the indexed property should be set on the question being checked to improve performance. If the function is used to check variables in a panel, the indexed property should be set on the variable being checked in the panel to improve performance.

10.1.11.8. isEmailTaken

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

```
isEmailTaken(email)
```

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

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

Note: If you call the function from a survey that is connected to a panel (for example a registration/update profile survey), then the string argument is compared with the email column in the panel, while if the function is called from a survey that is not connected to any panel, then the argument is compared to the email column in the response database of the actual survey.

10.1.11.9. DeleteCurrentResponse

Since the data of registration and profile update surveys are usually posted back to the panel database before the end of these surveys, it is usually not necessary to keep the data in the survey itself. Therefore a function is provided to enable deletion of the entire response at the end of the survey after all update scripts have been run:

```
DeleteCurrentResponse()
```

DeleteCurrentResponse will remove the entire record with all responses from the survey database.

When DeleteCurrentResponse is used at the end of the survey in a script node, it should be followed immediately by a redirect script. This is because any attempt to open a survey page (including the end page) after the record has been deleted will give an error, as the respondent no longer exists (see IsFromCommunityPortal and GetCommunityPortalReturnUrl on page 119 for more information).

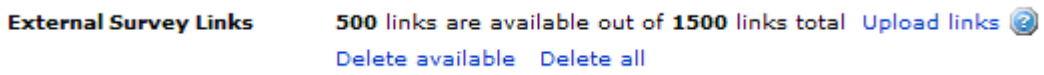
10.1.11.10. Functions for Sample Only

RedirectToExternalSurvey and GetExternalSurveyState

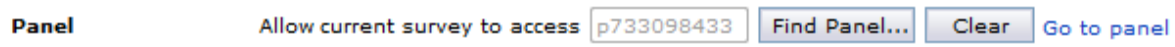
Note: "Sample Only" is a Panel add-on to be used by access panel providers who, as a part of their business, sell sample to other companies, on projects where they do not perform the data collection themselves. Any other usage of this functionality is prohibited. Contact your Confirmit account manager for more information.

With the "Sample Only" add-on enabled, you can upload a repository of external survey links to the sample job. To upload the links:

Create a text file (.txt) with 'Link' as the only column header and the external survey links listed, one link per row. On "Job Information", the interface shown below is available to upload external survey links to the job:



These survey links will then be included in the sample file(s). These can then be uploaded to a Confirmit project, which must be linked to a Panel on Project Overview:



This Confirmit project is used for:

- Emailing survey invitations and reminders to panelists
- Redirecting panelists to the external survey links
- Update status, panelist credits etc. if panelists are redirected back to the survey when finished in the external survey link with status.

You can also add questions to this survey; you could for example ask panelists to rate the survey experience etc. when they are finished with the external survey, or you may need an initial screener before respondents are redirected. Note however that you are NOT allowed to use this redirect survey for general data collection.

The following function can be used in this survey to handle redirects and statusing:

```
RedirectToExternalSurvey(queryString)
```

`RedirectToExternalSurvey` will redirect the panelist to the url uploaded to the background variable "ExternalSurveyLink", with `querystring` (optional, String) added to the link. `querystring` can be used if you need to send in some values with the link. This can be useful if the system running the survey is capable of fetching these values sent in with the link.

For example, if you have a background variable **gender** in the sample file, you could send in that value with the external survey link by using:

```
RedirectToExternalSurvey("?gender="+f("gender").get())
```

If you send in several parameters like this, then you must separate them with the ampersand (&) character.

The `RedirectToExternalSurvey` function will also set a session cookie that will be used when panelists are redirected back to the survey after finishing the external survey, to identify the panelist and set the correct status on the panelist for this job.

At the end of the external survey, there should be a redirect back to the Confirmit survey with the correct status in the survey link. This status will then be set both on the redirect survey, and back to the panel (in survey history). The statusing URL is one of the following:

http://<server>/wix/<pid>.aspx?status=complete

http://<server>/wix/<pid>.aspx?status=screened

http://<server>/wix/<pid>.aspx?status=quotafull

http://<server>/wix/<pid>.aspx?status=incomplete

http://<server>/wix/<pid>.aspx (this will default to complete)

<server> is to be replaced with either survey.confirmit.com or survey.euro.confirmit.com if you are an On Demand user, or your system's survey domain if you are an On Premise user. <pid> is to be replaced with the project id of the survey that is used for the redirect.

When panelists are redirected back, the system will retrieve the necessary information from the cookie to identify the panelist so that the right status is set, and open the survey after the redirect. The survey will be opened on the node following the script node with the Redirect script (for example a question, a script node or a stop node). Additional questions or scripts, for example to set Panelist Credits (points), can be placed after the Redirect script to be executed when the panelist is redirected back to the redirect survey. The redirect survey will be opened automatically at the point where the redirect was done when the panelists return from the external survey, due to the details stored in the cookie.

The cookie is a "session cookie", so is removed when the panelist closes his/her browser.

The function `GetExternalSurveyState` can be used to keep track of what state the panelist is in, i.e. whether status has been set or not.

```
GetExternalSurveyState()
```

GetExternalSurveyState will return one of the following state codes:

ExternalSurveyState.NONE

ExternalSurveyState.NOT_STARTED

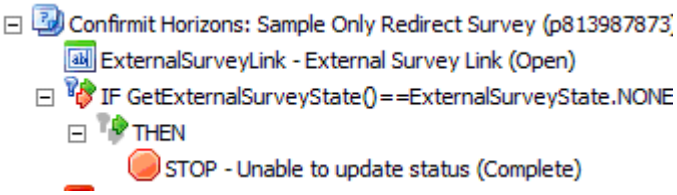
ExternalSurveyState.IN_PROGRESS

ExternalSurveyState.STATUS_SET

The `GetExternalSurveyState` function is useful if a panelist has disabled cookies in his/her browser. If the session cookie cannot be found, you will be unable to automatically update the panelist status. To treat this special case, you can include a condition with the expression

```
GetExternalSurveyState() == ExternalSurveyState.NONE
```

followed by a stop node so that these panelists don't cause any errors in scripts you have for example to update panel variables or points. The text on this stop node could for example notify the panelist that their points status was not updated automatically and that they will have to wait some days for this to be done, depending on the procedures for this. It is probably a good idea to flag these incidents by setting a hidden variable as well so that you keep track on how often it happens.



Note: The statusing survey must have "Encrypt system request parameters" (default) enabled in Survey Settings/Web for statusing to work.

If you follow the procedure described above, "Interview complete - Sample Only" units will be logged for completes instead of regular "Interview complete - CAWI ". These units are solely intended to be used by access panel providers who, as a part of their business, sell sample to other companies, on projects where they do not perform the data collection themselves. Any other usage of this functionality is prohibited. The following must be in place for the "Interview complete - Sample only" units to be logged:

1. The project must be linked to a Standard or Professional Panel.
2. The respondent must have a value in the panelistid column in the respondent table.
3. The respondent must have a value in the External Survey Link column in the respondent table.

All of this happens automatically when uploading sample from the sample job if you follow the procedure described above.

10.1.11.11. AddPanelSurveyHistory

This function is used to create a survey history entry for the current panelist, it is used in scenarios where the panelist is participating in a survey linked to the panel that they are a member of, but have not been directly sampled to.

```
AddPanelSurveyHistory( panelistId, jobNumber)
```

The input parameters are strings: panelistId is the panelist's ID from the linked panel and jobNumber is the job number in the panel that the survey history entry will be associated with.

The following syntax can be used to update the current panelist in job number 10 in the linked panel. An error will be reported if the insert fails:

```
AddPanelSurveyHistory(GetRespondentValue('PanelistId'),"10");
```

The following criteria must be met for the panelist's survey history to be updated successfully:

1. The survey must be linked with a Standard or Professional Panel.
2. The specified panelist ID must exist in the panel.
3. The survey needs the panel fields in the Respondent table: panelistid, jobnumber and runnumber

On successful execution of the function, the survey panel history for the specified panelist will be updated in the panel for this survey. If the job number does not exist, a new job will be created with this job number.

Note: This function is required when a Survey Router is used to route panelists from one survey to another where the routed-to survey is one that the panelist was not sampled to.

10.1.12. Basic Panels

This section describes two functions available for Basic Panels but which also may be used in ordinary projects.

Note: Basic Panels is a Confirmit add-on. If you do not have access to this functionality, contact your Confirmit account manager for more information.

10.1.12.1. isUsernameTaken

In Basic 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 is already a panelist in the sample with the provided *`userName`* (string) in the question with question id "username", `false` otherwise. Note that the question id must be "username".

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

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

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

Note: If you call the function from a survey that is connected to a panel (for example a registration/update profile survey), then the string argument is compared with the username column in the panel, while if the function is called from a survey that is not connected to any panel, then the argument is compared to the username column in the response database of the actual survey.

10.1.13. Classification Functions

Use these functions to check that the data input by the respondents, for example date and age information, is of the correct format.

10.1.13.1. 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.

Checking that a Response in an Open Text List Question is an Integer

If you have an Open Text List question *details*, 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 open text list question. So, if "age" has code 4 in the open text list 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 Range on page 72 for more information).

10.1.13.2. IsDateFmt and IsDate

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

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

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

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

All other characters are treated as separators.

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

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

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

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

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

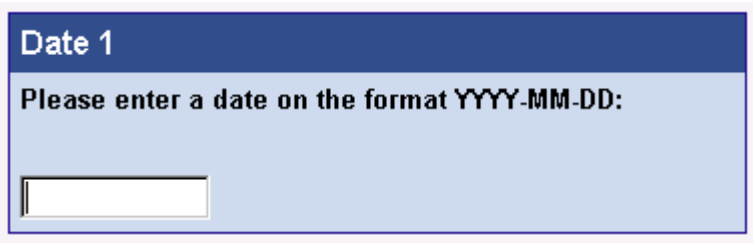
`IsDateFmt` and `IsDate` return an object with the properties `day`, `month` and `year` if the date is valid and in the correct format, `null` otherwise. A general description of objects follows (see Objects on page 138 for more information), and examples where the object returned from `IsDateFmt` and `IsDate` and the properties `day`, `month` and `year` are used are given.

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`.

Note: If you are using the Optimized Database format introduced in Confirmit 12.5, the Date question type will be available. This gives a date input box and calendar popup, and has built-in validation so that you do not need to use IsDate or IsDateFmt to validate the date.

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.

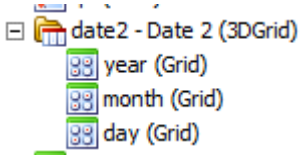


Here is a script you can use in the validation code of such a question:

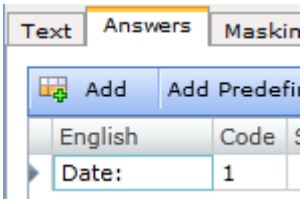
```
if(!IsDateFmt(f("date1").get(),"YYYY-MM-DD")) {  RaiseError();
SetQuestionErrorMessage(LangIDs.en,"Invalid date. Please correct. Use
the format YYYY-MM-DD"); }
```

Validating Date with Drop-downs for the Date Parts

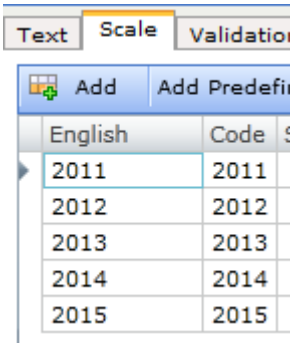
You can also set up date in three questions with drop-downs: One for year, one for month and one for day. To get these on the same line, you can place three grid questions in a 3D grid question as here:

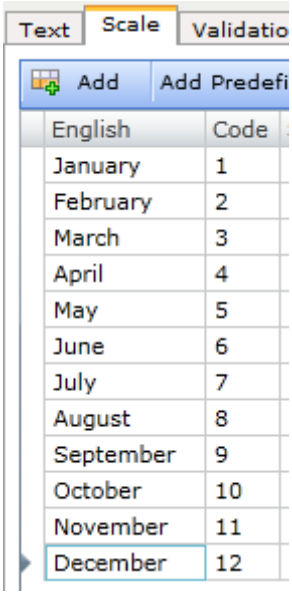


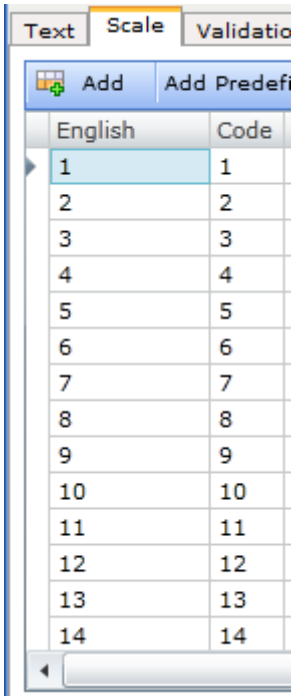
The 3D grid's answer list will just have one item:



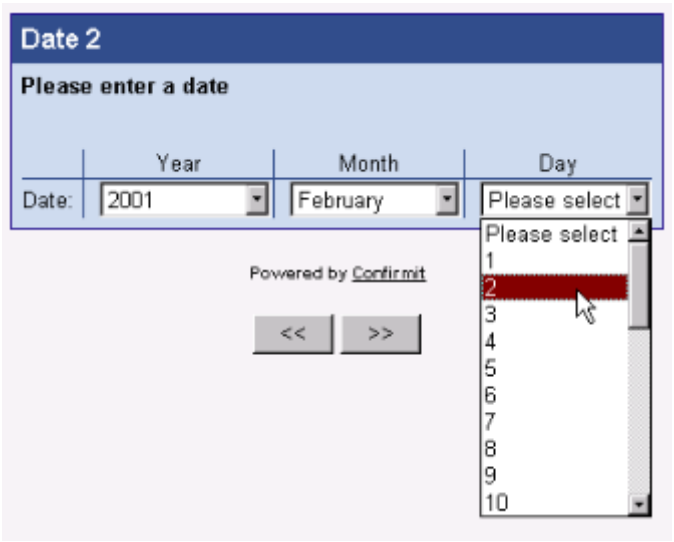
The grid's answer list will be years, months and dates. It is important that you define codes that are equal to the number of the years, months (1-12) and dates (1-31):


Text
Scale
Validatio
Add
Add Predefi
English
Code
2011
2011
2012
2012
2013
2013
2014
2014
2015
2015


Text
Scale
Validatio
Add
Add Predefi
English
Code
January
1
February
2
March
3
April
4
May
5
June
6
July
7
August
8
September
9
October
10
November
11
December
12



The question will then 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"].ge
t()))
 {
  RaiseError();
  SetQuestionErrorMessage(LangIDs.en,"Invalid date. Please correct.");
 }
```

10.1.13.3. IsEmail

```
IsEmail( argument )
```

`IsEmail` checks whether the argument has the format of a valid email address. It 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 @ character etc.).

Validation of Email Address Format

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

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

10.1.13.4. IsNet

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

Determines whether an IP address belongs to an Internet network.

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

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

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

You can use the `RequestIP` function to get the respondent's IP address (see RequestIP on page 86 for more information).

Excluding Respondents from Specific Networks

Assume you wish to exclude respondents with the network number "192.168.18.0", subnet mask "255.255.255.0" from taking a specific survey (for example because they work for the company that is running the survey).

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

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

10.1.14. General Utilities

Use these functions to send emails etc.

10.1.14.1. SendMail

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

`SendMail` can be used to send an email from inside a survey. For information on sending of multipart emails (see SendMailMultipart on page 130 for more information).

Arguments:

Argument	Description

From	Message sender (string)
to	Recipient's address (string)
subject	Message subject line (string)
Body	Message body (string)
*cc**	Carbon Copy (string)
*bcc**	Blind Carbon Copy (string)
*mailformat**	(int) (obsolete)
*bodyformat**	(int) 0 – HTML, 1 – Plain text
*codepage**	(int) The codepage the mail is to be sent with. 1252 is Western European, 65001 is Unicode (UTF-8).

*) optional

The last 5 arguments are optional, but if you include any of them, you have to include all arguments that precede them. For example, if you do not want a carbon copy (`cc`), just use an empty string (`""`) for that argument (see APPENDIX D: Codepage on page 221 for more information).

Send Confirmation Email at the End of a Survey

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

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

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

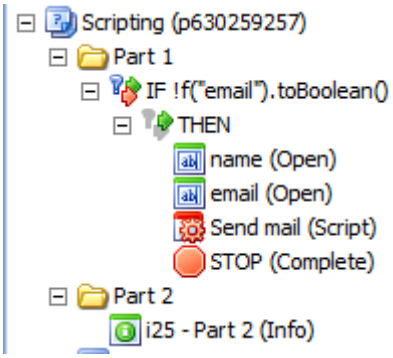
The email will be sent to the respondent, with a blind copy sent to "bcc@example.com". The respondent will not know that this copy has been sent as the address is in the bcc field. You must include an empty string ("") for the cc argument if you do not want to cc anybody but do want to bcc as in this 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".

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 Confirmit's ASP):

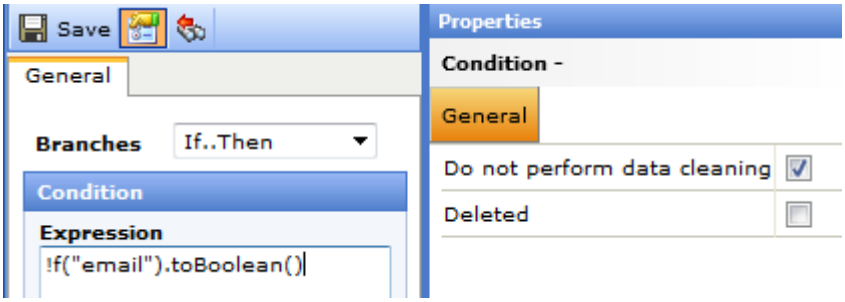
```
var body : String = "Thank you for registering.\r\n\r\n";
body += "Here is the url to your follow up survey:\r\n";
body += GetRespondentUrl()+"\n";

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 the code below:

```
^f("name")^
```

10.1.14.2. SendMailMultipart

```
SendMailMultipart(from,to,subject,bodyText,bodyHtml,cc,bcc,codepage)
```

`SendMailMultipart` can be used to send a multipart email from inside a survey. A multipart email is an email where the body text can be defined both as plain text and as HTML. The recipient's mail client will determine which version is to be displayed to the recipient.

Arguments:

Argument	Description
From	Message sender (string)
to	Recipient's address (string)
Subject	Message subject line (string)
BodyText	Message body, plain text (string)
BodyHtml	Message body, HTML (string)
cc*	Carbon Copy (string)
bcc*	Blind Carbon Copy (string)
Codepage*	(int) The codepage the mail is to be sent with. 1252 is Western European, 65001 is Unicode (UTF-8) (default value).

*) optional

The last 3 arguments are optional, but if you include any of them, you have to include all arguments that precede them. For example, if you do not want a carbon copy (*cc*), just use an empty string (`""`) for that argument (see APPENDIX D: Codepage on page 221 for more information).

10.1.14.3. SendPdfMail

The `SendPdfMail` function allows the generation and emailing of a PDF file from a survey with for example a summary of the respondent's answers. Typically the file is sent to the respondent, but it could also be sent to someone who is to review the respondent's answers.

The syntax of the function is:

```
SendPdfMail(html,from,to,body,subject,filename)
```

Arguments (all strings):

Argument	Description
Html	A string with the html that the PDF file should be created from.
From	Message sender
to	Recipient's address
body*	Message body

Subject	Message subject (optional)
Filename	Filename of pdf file (optional). Note that the filename must only contain alphanumeric characters. Do not specify the extension of the file.

*) optional

As the content of the PDF file is a string containing HTML, there is considerable flexibility concerning the layout and content of such a respondent-specific report.

If no subject is provided, the subject of the email will be “Respondent Report”. If no filename is provided, the name of the pdf file will be “respondentreport”.

The call to the function initiates a task that runs on the batch servers to generate and send a zipped PDF file. This means that, depending on the queue of tasks on the batch server, there may be a delay in producing and sending the file. If the PDF report is to be sent to the respondent, it is recommended that the respondent is informed of this possible delay. The function will not run in RDG mode.

For customers using Confirmit on Demand, there is a charge for each of these reports, just as for PDF exports from Reportal. It is possible to disable the creation and sending of all such reports. Contact Confirmit if you do not want your users to have access to this functionality.

For customers using Confirmit on Premise (own server installation), this functionality requires the PDF export add-on. Contact Confirmit for more information.

If this functionality is disabled, the call to the `SendPdfMail` function will not fail (so no script errors will occur), but the task will not create or send a PDF file.

The PDF-software does not parse anchor tags; it only adds the link text. So to have clickable links in the PDF file that is generated with SendPdfMail, you must include the URL itself.

http://www.confirmit.com

NOT:

Link text

If you want blue text and underline, you can define this in a tag:

HYPERLINK "http://www.confirmit.com%3c/span" http://www.confirmit.com

Sending a PDF to the respondent with answers from the survey

Here is a short example where the respondent answers questions on *name*, *email*, *age* and *gender*. Following these questions is a script that generates and sends a PDF file with the answers the respondent has given:

```
var html ="<html><body>\n";
html += "<h1>Answers for "+f("name").get()+"</h1>\n";
html += "<table><tr><th>Question</th><th>Answer</th></tr>\n";
html +=
"<tr><td>"+f("email").label()+"</td><td>"+f("email").get()+"</td></tr>
\n";
html +=
"<tr><td>"+f("age").label()+"</td><td>"+f("age").get()+"</td></tr>\n";
html +=
"<tr><td>"+f("gender").label()+"</td><td>"+f("gender").valueLabel()+"<
/td></tr>\n" ;
html += "</table>\n";
html += "</body></html>\n";

SendPdfMail(html,"noreply@us.confirmit.com",f("email").get(),"Here are
your responses to the survey.","Your responses","responses");
```

Note: Non-ASCII characters are not accepted by the PDF creator. If non-ASCII characters must be used, write valid HTML for the required character and include a metatag such as the following: <meta http-equiv="content-type" content="text/html;charset=utf-8" />

10.1.14.4. Redirect

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

`Redirect` can be used to redirect the respondents to a different site (*`url`*).

Important
If a redirect is included at the end of the survey, it is important that the interview status is also set using the `SetStatus` function (see GetStatus and SetStatus on page 81 for more information).

Note: The use 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 Confirmit.

`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.

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 as below:

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

`Redirect` is used in combination with `SetStatus` and `SetInterviewEnd` so that the respondent is given the correct status and end time stamp before being redirected out of the survey.

Note: The use of scripted Redirect() to jump into or out of questions that are in Call Blocks will cause problems and should not be attempted. A key attribute of Call Blocks is that they can be entered from multiple points in a survey, so the survey engine will not have a specific “entry-point” that has been used to enter that call block. Therefore when the Call Block finishes it does not know where in the main body of the survey to return to, and the survey will simply error. Refer to the Authoring User Guide for further details on Blocks.

10.1.14.4.1. Opening a Specific Survey Page or Call Block

If you for example use `Redirect` to redirect the respondent to another application, but want to send the respondent back to the Confirmit survey on a particular page, you can include a parameter in the survey link to indicate which page is to open. This parameter is `__qid` (two underscores + qid). Valid values are the question ids of any question on the root level of the survey (outside loops). When opening a survey url with this parameter, the survey will open on the page that contains the question indicated in `__qid`.

Example:

```
http://survey.confirmit.com/wix/pXXXXXXXX.aspx? __sid__=ww-
redcOmXd5s2yCeqAQYTjtCOc0bqXHg9QNh28l-
FFg7WyktC2GBFh8qJPBNGtY0&__qid=q4
```

This will just open that particular page (here: the page with question q4). No attempt is made to transfer background variables, set interview start or save any data.

Similarly, if you use __gotoCallBlock=CallBlockID (two underscores + gotoCallBlock) you can direct respondents directly to a specific call block. When the call block is finished, the interview will finish.

10.1.15. Survey Router Functions

This section describes functions that can be used in surveys linked to a Survey Router.

Survey Router is a Confirmit add-on. If you do not have access to this functionality, contact your Confirmit account manager for more information.

Survey Router surveys are set up as regular Confirmit surveys, and are linked to the Survey Router via the Survey Router menu. A survey can be registered in one Survey router and this is shown on the **Project Management > Overview** page General tab:

Survey Router Registered in My example router

This setting will allow scripts in the survey to find project IDs to route the respondent/panelist to or perform the routing itself.

10.1.15.1. GetAvailableSurvey

GetAvailableSurvey is used to get a Project ID that the respondent/panelist could be routed to based upon supplied selection criteria.

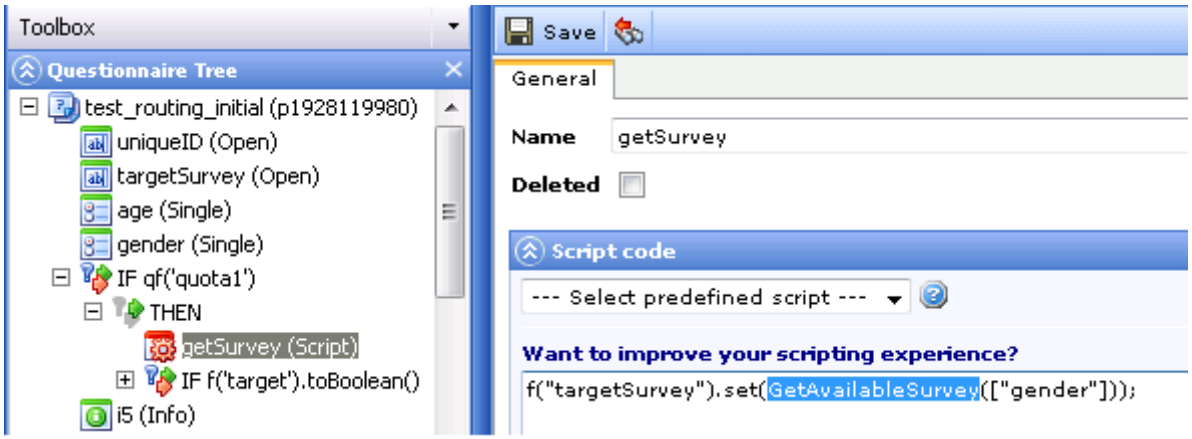
```
GetAvailableSurvey(fieldnames{, projectList, mode})
```

The list of question IDs supplied in fieldNames (array of Strings). Up to 5 questions IDs can be supplied (this value is configurable for On-Premise license holders) that you want to be checked by the Survey Router selection algorithm. The list of optional project ID to be included/excluded from the selection criteria supplied in projectList (array of Strings). Mode (Boolean) optional unless a projectList is supplied, determines whether the projectList is an inclusion or an exclusion list, where true demotes an inclusion list and false denotes an exclusion list.

In the following example both "gender" and "age" questions are considered in the survey selection but both project IDs "p1234567" and "p7654321" will be excluded from the selectable surveys.

```
var fieldNames : String[] = ["gender", "age"]
var projectList : String[] = ["p1234567", "p7654321"]
f("targetSurvey").set(GetAvailableSurvey(fieldNames, projectList, 0));
```

In this example the "gender" question is to be checked. If for example the respondent has answered the "gender" question in the original survey by stating he is "male", then the survey routing algorithm will check the remaining surveys in the group and ignore any surveys where the quota for males is already full. As in this case "gender" is the only selection criteria required, the routing algorithm would now select the survey with the highest priority and open that survey for the respondent.



Availability of the respondent's quota profile does not consider Optimistically populated (currently active) respondents; only those that have completed the survey and have caused the quota cell to be incremented.

In the event several projects satisfy the selection criteria and they all have the same priority, a project will be selected from the group at random.

In the event that no projects satisfy the selection criteria, no project IDs will be returned. When a potential project is found, the system will verify that the respondent has not already participated in that specific survey; if they have then the survey will not be considered as available for selection and the system will attempt to find another survey. After attempting to find another survey three times (this value is configurable for On-Premise license holders) due to the fact that the respondent has already participated, the system will stop trying to find a survey and will return nothing (as if no surveys are available for selection).

An exact match is required for all the criteria listed in the array. So in this case if a survey in the group does not have a "gender" question, then that survey will not be considered for selection by the algorithm.

10.1.15.2. RedirectToRouterSurvey

RedirectToRouterSurvey is used to seamlessly route the respondent from one survey to another inside of the Survey Router they are registered with.

```
RedirectToRouterSurvey(projectId, username{, fieldnames})
```

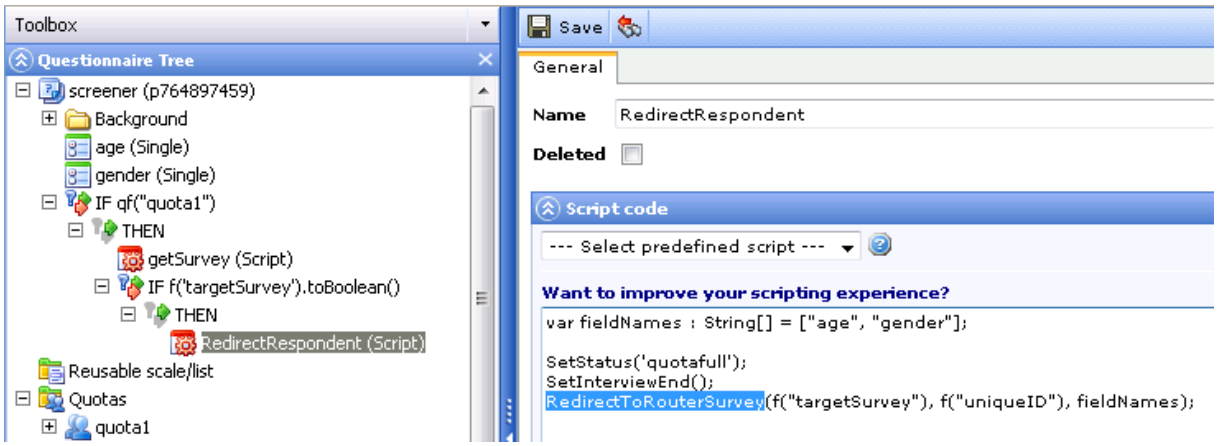
The project ID that the respondent is to be routed to is supplied in projectId (string), the unique identifier for the respondent is supplied in username (string) and the optional list of question IDs that should be passed to the routed-to survey is supplied in fieldNames (array of Strings).

In the following example the respondent will be routed to the project ID stored in the question "targetSurvey" using the unique identifier supplied in the question "uniqueID" and passing the values to questions "age", "gender" and "region".

```
var fieldNames : String[] = ["age", "gender", "region"]
RedirectToRouterSurvey(f("targetSurvey"), f("uniqueID"), fieldnames);
```

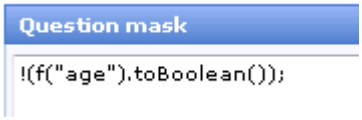
Open text, single choice and numeric questions can be passed to the target survey, other question types are not supported, these can be normal, background or hidden question types, but no validation is performed on the data until the question is executed in the target survey. If the question passed does not exist in the target survey no error will be reported.

In this example, having the quota of the current project they will automatically be routed to a new project within their Survey Router group with both age and gender questions being passed to the target survey. The respondent has been set to Quota full" in the initial survey.



Note: Survey router redirection will only occur if the Survey Setting > Web Options > Survey type is set to "Limited survey with external respondent creation" in the target survey.

Typically, you will not want to ask the same questions in the target survey as you have already asked in the initial survey. To prevent this, a question mask can be applied to the questions that are passed to the target survey. In this case you may check if the question has already been answered, and if so, do not display the question again:



Note: If you use this masking, remember to check the "No cleaning on question masking" property (refer to the Authoring User Guide for more information). If this property is not checked, then the data will be cleared if the page is empty.

10.2. Creating Your Own Functions

You can create your own functions and use them where needed. It is recommended to use functions:

- Whenever there are pieces of code that it is likely that you will reuse several times in the questionnaire.
- When you need to refer to values (often computed expressions) that are not stored in survey variables.
- To group statements that perform a well-defined task, to make the code easier to read, and to make your code more modularized.

Functions can be called from anywhere within the questionnaire.

Type annotation in a function specifies a required type for function arguments, a required type for returned data, or a required type for both. If you do not type annotate the parameters of a function, the parameters will be of type Object. Likewise, if the return type for a function is not specified, the compiler will infer the appropriate return type.

Using type annotation for function parameters helps ensure that a function will only accept data that it can process. Declaring a return type explicitly for a function improves code readability since the type of data that the function will return is immediately clear.

10.2.1. Defining functions

Before you can use a function, you must define it. In Confirmit, functions are usually defined in script nodes. The syntax of a function definition is as follows:

```
function FunctionName(p1 { : type},p2 { : type},...,pn { : type}) : type
 {
   <statements>
 }
```

The function name is used to refer to the function in function calls. The same naming rules apply to function names as to variable names. The convention for function names is to start them with a capital letter, to distinguish them from variable names. The arguments (p_1,p_2,...) are the names of the variables that receive the values passed to the function. You may define functions without any arguments at all, and functions with a variable number of arguments.

Type annotation on parameters and the function itself (for the return type) is not required.

10.2.2. Function Call

As we have seen in numerous examples, to call a function simply use the function name followed by parentheses containing the arguments, if any:

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

A function defined in a script node in a Confirmit questionnaire, may be called in all script context, i.e. from other script nodes, conditions, response piping (within ^s) and in code masks and column masks in the same questionnaire.

The function `RequiredMulti` defined previously (see Defining functions on page 136 for more information) can be called from the validation code of a multi question like this:

```
RequiredMulti();
```

When calling a function, ensure that you always include the parentheses and any required arguments. Calling a function without parentheses causes the text of the function to be returned instead of the results of the function.

10.2.3. Functions with a Fixed Number of Arguments

When you define a function with arguments, you define variables with the argument names as variable names. The arguments will hold the values that are passed into the function, and may be used as normal JScript .NET variables inside the function.

10.2.4. Functions with a Variable Number of Arguments

Previously in JScript you could define functions that took a variable number of arguments, using the `arguments` array. In JScript .NET fast mode this is not available. If you need a function to take an arbitrary number of arguments, you can use a *parameter array*. This is done by including, as the last element in the argument list, an array that is defined with three periods (…), then the name of the array and then a typed array annotation:

```
function FunctionName(... parArray : type[]) {  <statements> }
```

`parArray` can then be used like any other array inside the function.

```
parArray.length
```

returns the number of arguments in this array.

To refer to each argument in this array, use indexing as in all other arrays: 0 for the first argument, 1 for the second, and so on:

```
parArray[index]
```

This makes it possible to write extremely flexible functions, as we have seen examples of in the built-in arithmetic functions in Confirmit; Sum, Count, Average, Max and Min (see Arithmetic Functions on page 68 for more information).

Code Masking Based on a Variable Number of Conditional Expressions

The following function can be used to create a code mask where the items to be included in the mask is determined by the results from a number of conditional expressions that are sent in as parameters. It does require some caution when used: It requires all answers to have numeric codes, and the position of the expression represents each code (1,2,3,…).

```
function Pattern(... parArray : Object [])
{
  var codes = parArray.length;
  var s = new Set();
  for(var i: int = 0; i<codes; i++)
  {
    var code = i+1;
    var result = parArray[i];
    if(result==true)
    {
      s.add(code);
    }
  }
return s;
}
```

Here is an example of calling this function from a code mask:

```
Pattern(f("q1").toBoolean(),f("q2")=="2",f("q3").any("1","3","4"),true
)
```

If q1 is answered, the answer with code 1 would be included. If the answer to q2 is "2", the answer with code 2 would be included. If q3 (a multi) has answer(s) 1, 3 or 4, code 3 is included, and code 4 is always included.

Note: The script node defining this function must be placed before the question in which it is called in the questionnaire tree.

10.2.5. The return Statement

Often you want a function to return a value, either a result from a calculation that has been done in the function, or just a Boolean to indicate whether the function did what it was supposed to. To return a value to the statement that invoked the function, use the return statement:

```
return expression
```

The return statement will terminate a possible loop and send the value to the statement that invoked the function.

Returning a Calculated Value from a Function

The following function will return what percentage the number n is out of the base `b`:

```
function Percentage(n : int,b : int) : float
 {  return (n/b)*100; }
```

11. Objects

JScript .NET is an **object-oriented language**. We will not elaborate on the object-oriented features in this manual. We will just describe the JScript and Confirmit objects that are relevant for survey programmers, and advice the reader to consult a JScript .NET reference for more in-depth knowledge.

All script code in Confirmit is wrapped as functions inside a class. This means that you can not define your own classes in Confirmit scripts.

JScript objects are collections of **properties** and **methods**. A method is a function that is a member of an object. A property is a value or set of values that is a member of an object.

11.1. Properties

Properties are used to access the data values contained in an object.

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

```
objectName.propertyName
```

We have already seen an example of a property on the `Array` object, a property called `length` for the size of the array (for example, the `weekday` array from previous examples):

```
weekday.length
```

11.2. Methods

Methods are functions that are used to perform operations on an object.

You access a method of an object by combining the object's name and the name of a method:

```
objectName.methodName(arguments)
```

Just like in functions, arguments in method calls are optional.

We have already used the basic variable objects of Confirmit returned from the `f` function. We have seen some methods that can be applied on these objects to retrieve information about the questions in a Confirmit survey. For example will

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

return the title of the question with ID *qID*, and

```
f(qID).get()
```

the code or answer stored in the database for that question.

We have also seen a method that can be used to set the answer of a question:

```
f(qID).set(code)
```

11.3. Constructors: Creating Instances of Objects

Instances of objects of a particular object type are created using an operator called `new`, which you already have seen used when creating an array, for example:

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

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

```
variableName = new objectType(arguments);
```

`objectType(arguments)` is called the **constructor** of the object. Some object types have more than one constructor. Constructors differ in the number of arguments they allow.

12. Confirmit's f Function

The f function is probably the most important function available in Confirmit. It is used to access survey variables/questions, and we have already seen numerous examples of how to use it (see Methods of the Form Objects on page 40 for more information)

12.1. Calling the f Function

```
f(qID,{loopiter1,loopiter2,...,loopitern})
```

returns a form object of some kind. A **form** is a question or loop node in a Confirmit questionnaire. The argument *qID* is the question ID as defined in the properties of the question.

The rest of the argument(s) ($loopiter_1, loopiter_2, ..., loopiter_n$) are only used for questions inside loops. These arguments are codes 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.

Let us say you have a question q1 in a nested loop structure with two loops. The outer loop has iterations with codes "1" and "2", and the inner loop has iterations with codes "a" and "b". Then the question q1 is asked four times, and these four instances of q1 can be referenced like this (in the order they were asked) from outside the loop:

```
f("q1","a","1")
f("q1","b","1")
f("q1","a","2")
f("q1","b","2")
```

12.2. Storing the Form Object in a Variable

The form objects can be large data structures with long answer lists and texts. Every call of the f function will result in a new instance being retrieved. If the question has a code or scale mask, this mask will be evaluated for every call on the `f` function for this question. It is recommended to try to reduce the number of calls on the `f` function, for example by not always applying methods and properties directly on the function as we have been doing in all our examples so far. If you will be calling the `f` function on the same question ID several times in a script (especially if you call the `f` function inside a `for` or `while` loop), the best approach is to call it once and store the object instance in a variable instead:

```
variableName = f(qID,{loopiter1,loopiter2,...,loopitern});
```

Usually the difference is not significant, but it is recommended because it will reduce the execution time of your scripts. The interview pages will then load faster reducing the risk of irritating respondents and reducing server performance.

12.3. Compounds

The `f` function returns in fact different objects for different question types. These objects have some properties and methods in common, but some are specific to questions of a specific type.

Grid, multi, open text list, numeric list and ranking 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:

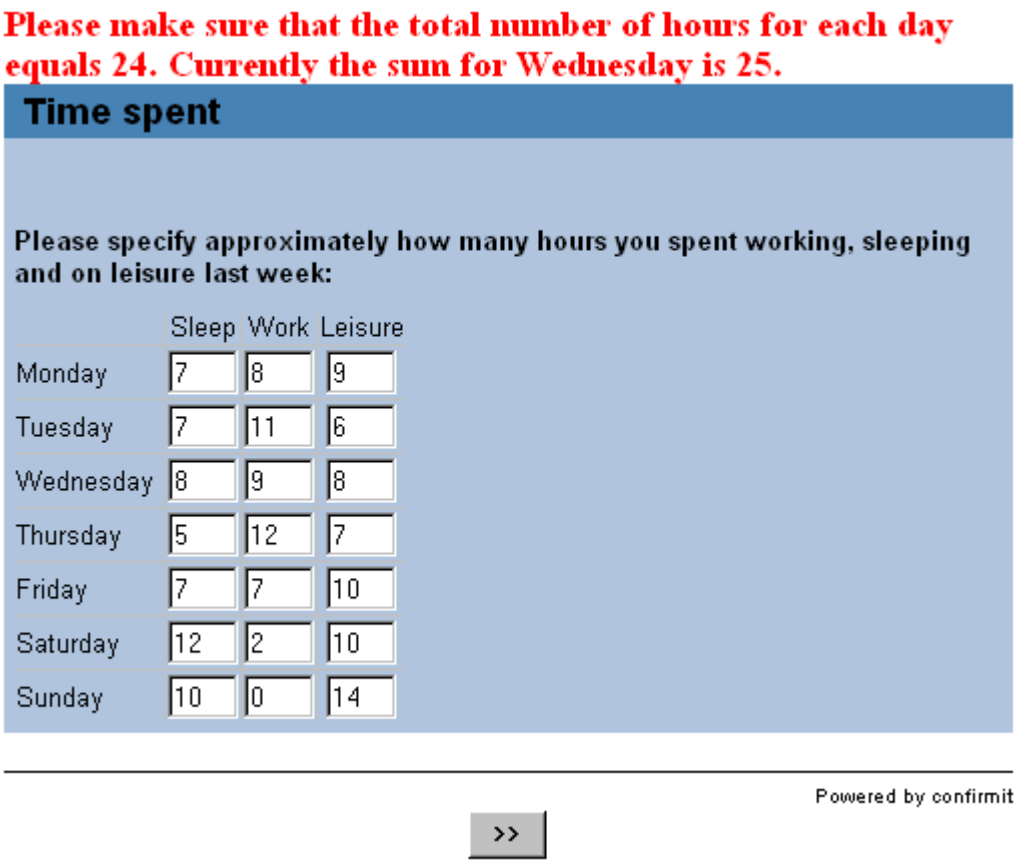
```
x[code]
```

x represents a variable holding an instance of an object returned from calling the `f` function on a compound. *code* is a string representing the code from the answer list of the compound. An example is referring to the element with code `"1"` in a multi question with question ID *q2*:

```
f("q2")["1"]
```

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.



In our previous solution for validation of this question (see The while Statement on page 58 for more information) we looped through the weekdays. For each day, we called the f function for three questions q2, q3 and q4:

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

Now, instead we can define three variables sleep, work and leisure to hold the form objects:

```
var sleep = f("q2");
var work = f("q3");
var leisure = f("q4");
var codes = sleep.domainValues(); // array with all codes
var i : int = 0;
var correctSum : Boolean = true;
while(i<codes.length && correctSum) //iterate through codes (rows)
 {
  var code = codes[i]; //current code
  //calculate sum:
  var sum : int = Sum(sleep[code],work[code],leisure[code]);
  if(sum != 24)
   {
    correctSum = false;
   }
  i++;
 }
if(!correctSum)
 {
   RaiseError();
   SetQuestionErrorMessage(LangIDs.en,"Please make sure that the total
number of hours for each day equals 24. Currently the sum for
"+sleep[code].label()+" is "+sum+".");
 }
```

The way the script was initially set up you could have as much as 21 calls (3*7=21) on the f function inside the loop. We have now reduced this to 3 function calls.

12.4. Properties

The form objects can have the following properties, depending on the type of the question (x represents a form object returned from the f function):

```
x.CODED
```

CODED is true if the question referenced is a **coded variable.** If not, CODED is undefined. Coded variables are single questions and loops, and answer list items of a grid or a 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 an open text list, numeric list or ranking question.

```
x.DICHOTOMY
```

DICHOTOMY is true if the question referenced is a **dichotomy.** If not, DICHOTOMY is undefined. The elements of a multi question (referenced with f(qID)[code]) are dichotomies.

```
x.COMPOUND
```

COMPOUND is true if the question referenced is a **compound.** If not, COMPOUND is undefined. Multi, open text list, numeric list, ranking and grid questions are compounds.

```
x.NUMERIC
```

NUMERIC is true if the question referenced is numeric. If not, NUMERIC is undefined.

```
x.EXTERNAL
```

EXTERNAL is true if the question referenced gets its answerlist from Database Designer (either as a lookup table or a hierarchy).

```
x.DATE
```

DATE is true if the question referenced is a date question. If not, DATE is undefined.

```
x.BOOL
```

BOOL is true if the question referenced is Boolean (has the Boolean property set). If not, BOOL is undefined.

```
x.GEO
```

GEO is true if the question referenced is a Geolocation question. If not, GEO is undefined.

Note: The Boolean property and Date question type are only available when using the Optimized database format.

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.

Using a combination of COMPOUND with DICHOTOMY and OPEN enables the user to differentiate between a grid and a multi and a numeric list, open text list or ranking question since the elements of a multi are a DICHOTOMY, but DICHOTOMY would be undefined for a grid, numeric list, open text list or ranking, and elements of numeric list, open text list or ranking are OPEN, but OPEN is undefined for grid and multi.

```
var form = f("q1");
if(form.COMPOUND)
 {
 var cats = form.domainValues();
 var code = cats[0];
 if(form[code].DICHOTOMY)
 {
 //form is a multi
 }
 else if(form[code].OPEN)
 {
 //form is a numeric list, open text list or ranking question
 }
 else
 {
 //form is a grid
 }
}
```

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, date, numeric, multi, open text list, numeric list, ranking 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 code "98".)

Here is the definition of the function:

```
function ClearForm(form)
 {
  if(form.COMPOUND) //form with multiple items
   {
    var fcodes = form.domainValues(); //all codes in form
    for(var i : int = 0;i<fcodes.length;i++) //iterate through code
     {
       form[fcodes[i]].set(null); //clear item
     }
   }
  else //form with one item
   {
    form.set(null);
   }
 }
```

Copying the Contents of any Form into Another

Here is a function that will copy any type of question. It will return true if the copying succeeds, and false without copying anything if it does not. (If the from and to questions were not the same type of questions).

```
function CopyForm(from,to) : Boolean
 {
  if((from.CODED && to.CODED) || (from.OPEN && to.OPEN) ||
(from.DICHOTOMY && to.DICHOTOMY))
    {
    to.set(from.get());
    return true;
   }
  else if(from.COMPOUND && to.COMPOUND)
   {
    var fcodes = from.domainValues();
    var tcodes = to.domainValues();
    if(fcodes.length == tcodes.length)
     {
      for(var i : int = 0;i<fcodes.length;i++)
       {
        if(fcodes[i].toString()==tcodes[i].toString())
         {
          to[fcodes[i]].set(from[fcodes[i]].get());
         }
        else
         {
          return false;
         }
       }
      return true;
     }
   }
  return false;
 }
```

12.5. The f Function Methods

The methods of the form objects are listed below. The methods are described in detail in sections Conversion Methods in Confirmit, Methods of the Form Objects, and Methods of the Set Object.

Methods	CODED	OPEN	BOOL	DATE	NUMERIC	DICHOTOMY	COMPOUND	GEO
toBoolean	X	X	X	X	X	X	X	X
toNumeric	X	X			X	X		
toInt	X	X	X		X	X		
toDecimal	X	X			X			
toDate				X				
day				X				
month				X				
year				X				
datestring				X				
get	X	X	X	X	X	X		X
set	X	X	X	X	X	X	X	X
label	X	X	X	X	X	X	X	X
text	X	X	X	X	X	X	X	X
instruction	X	X	X	X	X	X	X	
value	X		X	X				X
valueLabel	X		X					

domainValues	X		X				X	
domainLabels	X		X				X	
categories							X	
categoryLabels							X	
values							X	
getType	X	X	X	X	X	X	X	X
any	X						X	
all							X	
none	X						X	
between					X			
inc	X						X	
size	X						X	
union	X						X	
isect	X						X	
diff	X						X	
members	X						X	
isNearBy								X
latitude								X
longitude								X

13. Working with Sets

Sets are typically used in code masks of single, multi, open text list, numeric list, ranking and grid questions, in loops, and in scale masks of grids. A set consists of a collection of strings that represent codes. When used in code and scale masks, the codes contained in the set are used to determine which answer alternatives are to be displayed to the respondent. Only answers corresponding to codes from the set returned from the expression in the code or scale mask will be shown.

13.1. Constructor

An instance of the Set object can be created like this:

```
s = new Set();
```

You will also get instances of the Set object returned from the functions `a`, `set`, `nset`, `nnset`, and `Filter`.

13.2. Functions Returning Sets

A number of functions return sets. These are described in the following sections.

13.2.1. The a Function

The function `a` returns a set consisting of all possible codes on a question (i.e. the codes of all items in the answer list).

```
a(qID)
```

13.2.2. set, nset and nnset

There are three functions you can use to define sets with specific codes – `set`, `nset` and `nnset`.

```
set(code1,code2,...,coden)
```

will return a set consisting of the codes listed within the parenthesis. For example

```
set("1","3","4")
```

returns a set consisting of the codes "1","3" and "4".

```
nset(n)
```

will return a set with codes "1","2",...,n. n must be an integer greater than 0. For example

```
nset(5)
```

returns a set consisting of the codes "1","2","3","4" and "5".

```
nnset(m,n)
```

will return a set consisting of codes from `m` to `n` inclusive, i.e. `m,m+1,...,n-1,n`. For example

```
nnset(6,10)
```

returns a set consisting of the codes "6","7","8","9" and "10".

13.2.3. lset

```
lset(params)
```

lset can be used to construct a set. The input to the function can either be an object such as an Array, or a list of parameters from which the function will create a set.

```
lset(new Array(1,2,8))
```

returns a set consisting of the codes 1, 2 and 8.

13.2.4. 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 codes 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 codes of the answers "Cadillac", "Chevrolet" and "Chrysler" if they are in the answer list of *qID*.

Filtering an Answer List by the First Characters in the Answer

There may be situations where you have a single question with a long answer list, which you want to filter based on text input by the respondents. In this case 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.

For example, imagine you have a single question *carmake* where the respondent is to be asked to select a car make from a list. There are a lot of different car makes, so the selection list will be long. Here you can place an open text question *prefix* in front of the car question. In the prefix question the respondent is asked to type in the first letter in the name of the car, and this will filter the answer list such that only those car makes starting with the input letter will be presented. The code mask of the car question can have this code mask:

```
Filter("carmake",f("prefix").get())
```

13.2.5. The f Function

The `f` function can also be used in set context (code 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 code 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 section, are `add` and `remove`.

```
f(qID {,loopiter1,loopiter2,...,loopitern})
```

The curly brackets {} are used here to indicate that the loop iterations are optional arguments (see Confirmit's f Function on page 139 for more information).

13.3. Methods of the Set Object

In the syntax below, s_1 and s_2 represents instances of set objects or form objects (returned from the `f` function). For `add` and `remove` s_1 represents only an instance of the set object, because these methods are not defined for form objects.

13.3.1. inc

```
s1.inc(code)
```

`inc` returns `true` if the *code* is contained in the set s_1.

For example, for a multi question `q1`, the expression

```
f("q1").inc("1")
```

is `true` if the answer with code "1" has been selected on *q1*.

If you need an expression that is `true` if the answer with code "1" **or** the answer with code "2" has been selected, you can use this code:

```
f("q1").inc("1") || f("q1").inc("2")
```

If you need an expression that is `true` if the answer with code "1" **and** the answer with code "2" has been selected, you can use this code:

```
f("q1").inc("1") && f("q1").inc("2")
```

13.3.2. size

```
s1.size()
```

`size` returns the number of elements in the set s_1.

Asking for Favorite Only when More than 1 Item is Chosen

If you have a multi question cars asking for cars the respondent would like to drive, followed by a single question cars_favorite asking for the favorite among those selected in cars, the single question is only needed when more than 1 element is chosen on the multi. This can be achieved with the following expression in a condition:

```
f("cars").size()>1
```

The *cars favorite* question will then have the following code mask so that it only shows answers from the cars question::

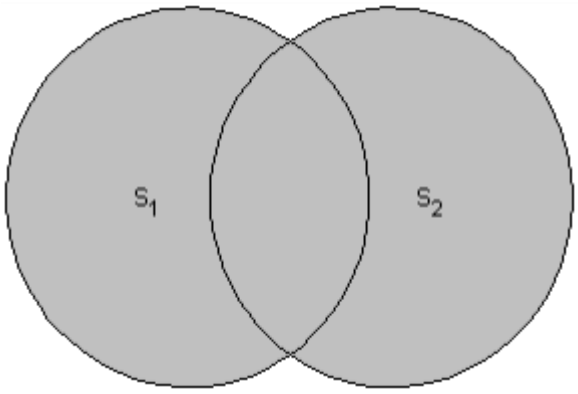
```
f("cars")
```

13.3.3. union, isect and diff

There are three methods available to use for set operations: `union`, `isect` and `diff`. These methods are useful when you need complex code masks, for example when you want to filter the answer list based on answers to two previous questions. Examples of this are: Showing the items answered on both questions, on any of the questions, on one but not the other etc.

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

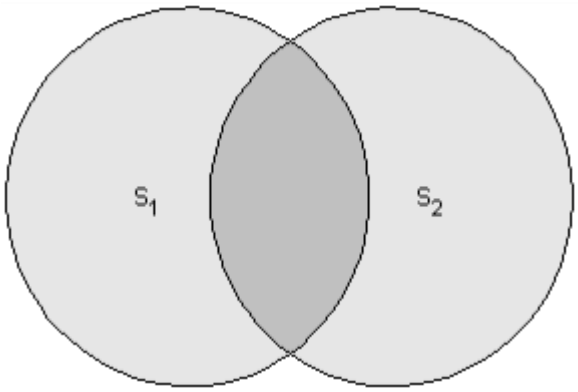
The **union** of two sets s_1 and s_2 is the set obtained by combining the members of both sets. So union will return a set consisting of all elements in s_1 **or** s_2.



s1.union(s2)

```
s1.isect(s2)
```

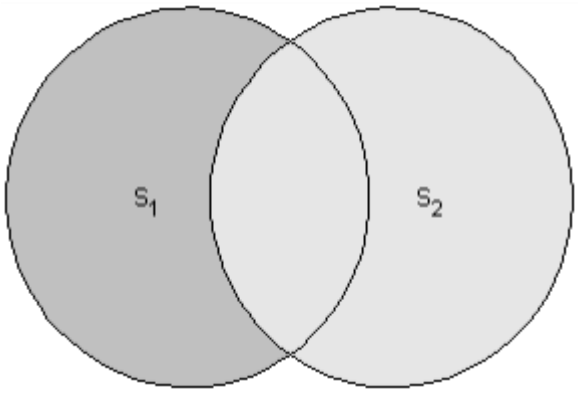
The **intersection** of two sets s_1 and s_2 is the set of elements common to s_1 and s_2. So `isect` will return a set consisting of the elements that are both in s_1 **and** s_2.



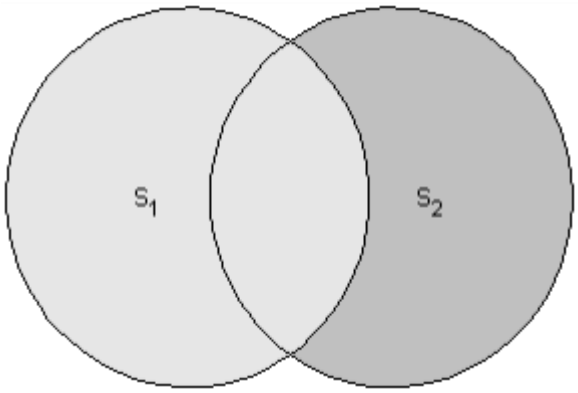
s1.isect(s2)

```
s1.diff(s2)
```

The **difference** between two sets s_1 and s_2 will yield a set consisting of the elements that are in the first set, but not in the other. For difference the order of the sets in the expression is significant. `s1.diff(s2)` will return a set consisting of the elements in s_1 that are not in s_2, but `s2.diff(s1)` will return a set consisting of the elements in s_2 that are not in s_1. As the illustrations show, these are two completely different sets.



s_1.diff(s_2)



s_2.diff(s_1)

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 code 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 code 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 code mask like this:

```
f("q1").diff(f("q2"))
```

Similar, for items answered in *q2* but not in *q1*:

```
f("q2").diff(f("q1"))
```

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 codes in the answer list, and use the diff method to remove the codes of the answers selected on *q1*. The code for such a code mask will be like this:

```
a("q2").diff(f("q1"))
```

You have to use the same codes for corresponding items in the answer lists of the two questions. This can easily be achieved for example by using a predefined list.

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 code to "Don't know" that is different from the other codes, for example by using a large number like "99" or letters like "DK". We will use "DK" in this example. In the code mask of *q3* we can use this code:

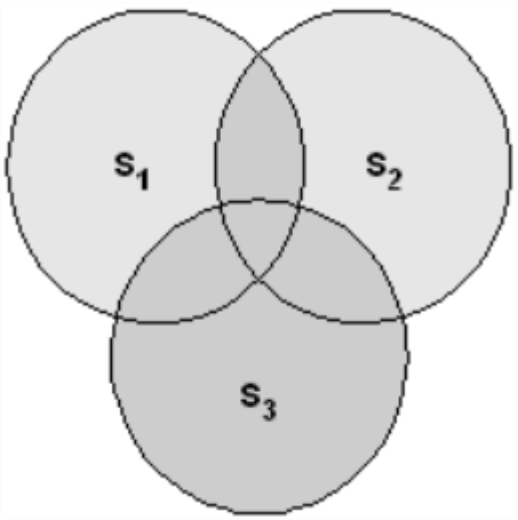
```
f("q1").union(set("DK"))
```

13.3.4. Combining Set Operators

You may build expressions where several set operators are combined. When several set operators are included in the same expression, the expression is evaluated from left to right. So if you for example have three sets s_1, s_2 and s_3, the expression

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

will give a set consisting of all codes either in the sets s1 and s2, or in set s_3:

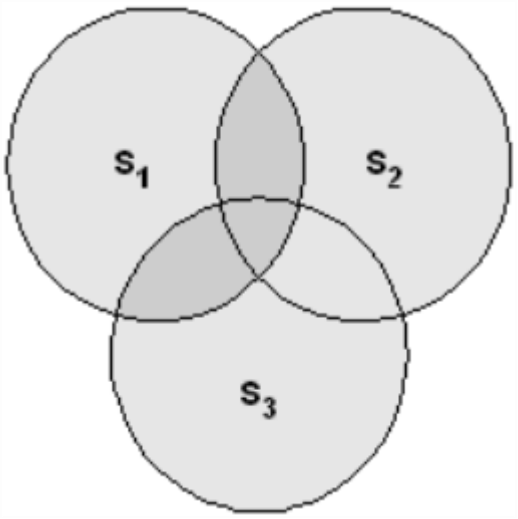


s1.isect(s2).union(s3)

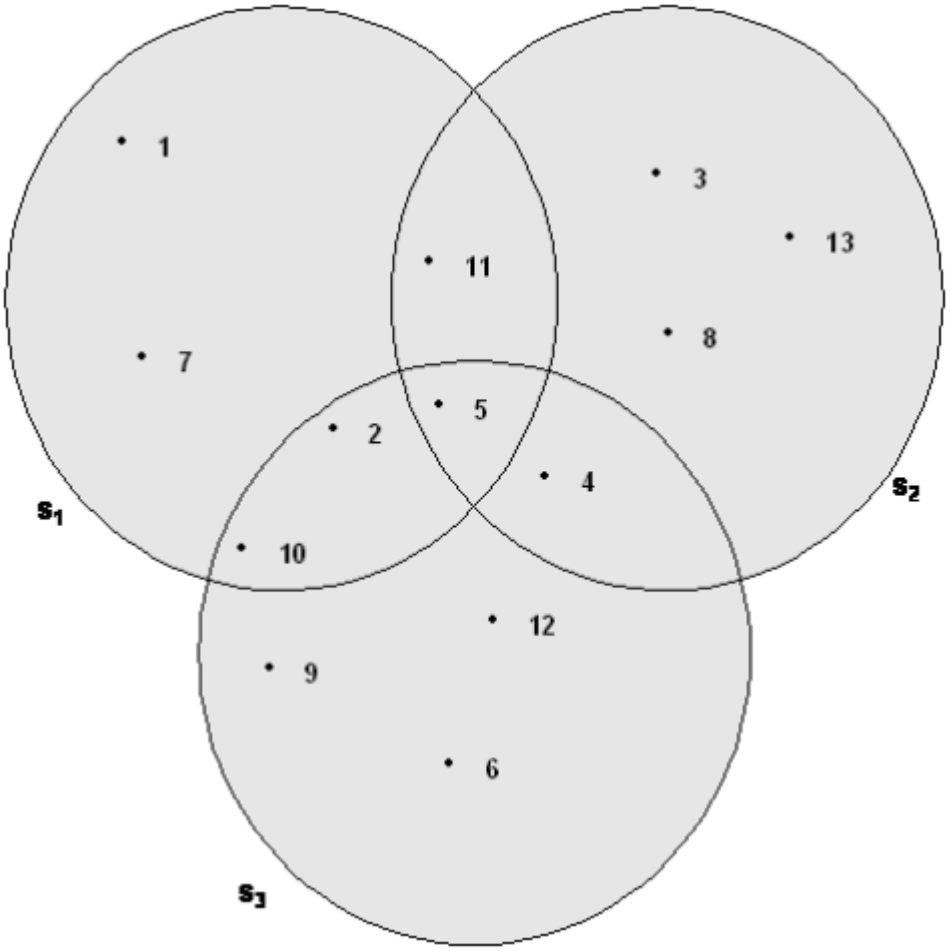
To force a sub-expression to be evaluated before other expressions, you can insert entire expressions inside the parentheses of the set methods. In the expression

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

the sub-expression s2.union(s3) 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 codes both in $s_1$and in either of s_2 or s_3:



s1.isect(s2.union(s3))



You have three sets: `s1`, `s2` and `s3`. `s1` consists of the codes `1,2,5,7,10,11`, `s2` consists of the codes `3,4,5,8,11,13` and `s3` 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?

1. s1.union(s3.diff(s2))
2. s1.union(s3).diff(s2)
3. s1.isect(s2.diff(s3))

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

The answers are given in APPENDIX A Answers to Exercises.

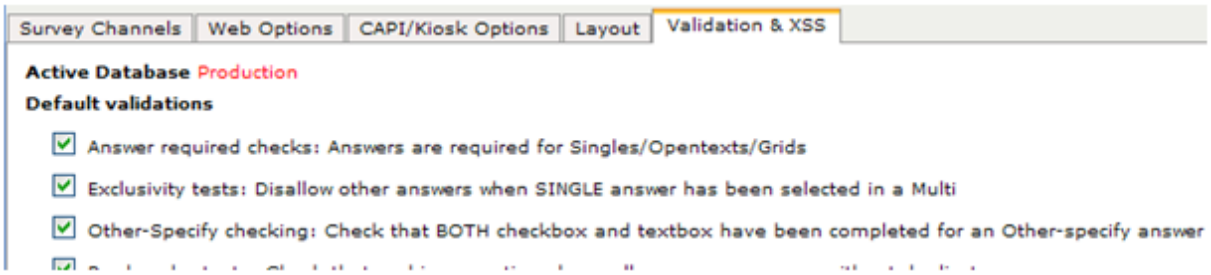
13.3.5. members

```
s1.members()
```

`members` is used to convert a set to an array.

Validating "Other, specify" in a 3D Grid

For grids/singles/multiples it is possible to automatically validate Other-specify by checking the "Other - Specify checking..." setting in **Project Management > Survey Settings > Validation & XSS** tab:



However this setting does not apply to 3D grids since Confirmit cannot deduce which question within a 3D grid should be validated with the Other-specify; instead it is necessary to use some manual validation code.

The question ID to refer to an Other-specify within a 3D grid is:

```
3d-grid-id_code_other
```

For example, we have a 3D grid (g1) containing a multi (q1) and a rating grid (q2); the 3d grid has answers/codes 1-4 with the 4th code being an Other specify.

We wish to validate so that the respondent is forced to add Other-specify text if they select this option in the multi q1. The validation code for this is:

```
var codes = a("q1").diff(set("97","98")).members();
if(f('q1').inc('4') && !f('g1_4_other').toBoolean() )
 {
 RaiseError();
 SetQuestionErrorMessage(LangIDs.en,"Please specify in the Other
Specify box");
 }
if(!f('q1').inc('4') && f('g1_4_other').toBoolean())
 {
 RaiseError();
 SetQuestionErrorMessage(LangIDs.en,"Please do not specify in the
Other Specify box without selecting this answer");
 }
```

13.3.6. add and remove

To add and remove items from a set, there are two methods available: `add` and `remove`.

```
s1.add(code);s1.remove(code);
```

`add` will add `code` to the set s_1. If code is already in the set s_1, the set will not be changed. `remove` will remove `code` from the set s_1. If code is not in the set s_1, the set will not be changed.

These methods are not defined for the form objects (returned from the f function), only for the set object.

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 codes = form.domainValues();
  var s = new Set();
  var i : int;
  for(i=0;i<codes.length;i++)
   {
    var code = codes[i];
    if(form[code].get() == "4" || form[code].get() == "5")
     {
      s.add(code);
     }
   }
  return s;
 }
```

This function returns a set with the codes of the items that have received a score of "4" or "5". In the code mask where you want to use it , call this function with the question ID of the grid (for example *q3*) as argument:

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

Note: If the respondent does not score any grid elements "4" or "5", then ScoreFilter() will return an empty set/mask and the respondent will be presented with an empty question (since all answers are masked out). It would therefore be reasonable to also include a question mask (or condition node):
`ScoreFilter("q3").size()>0`
Then, if the question is 'empty', it will be skipped.

13.3.7. .any, .all and .none

These methods can be used to check whether a set includes specific codes:

```
s1.any(code1,code2,…,coden);
s1.all(code1,code2,…,coden);
s1.none(code1,code2,…,coden);
```

any will return true if *any* of the codes listed can be found in the set s1.

all will return true if *all* of the codes listed can be found in the set s1.

none will return true if *none* of the codes listed can be found in the set s1.

13.4. User-Defined Functions in Code or Scale Masks

The previous example used a user-defined function in the code mask field. This is a convenient way of doing it, but there are a few pitfalls with this approach.

The code 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 code mask, the code mask is evaluated. So the function in the code mask of a question will be called every time there is a reference to the question in scripts elsewhere in the survey.

Due to this, the time it takes for the respondent to load the survey pages will increase slightly because there are more scripts to execute.

You are therefore recommended to be cautious when using functions in code 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.

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

As described previously (see add and remove on page 152 for more information), we have a grid where the elements are given a 1-5 rating, and we want the next question to use only the elements that are given a score of "4" or "5". Now let us use a script to set a hidden multi question, *scorefilter*, instead of using a function. The grid has question ID *carrating*.

```
var form = f("carrating");
var codes = form.domainValues();
var i : int;
for(i=0;i<codes.length;i++)
 {
  var code = codes[i];
  if(form[code].get() == "4" || form[code].get() == "5")
   {
    f("scorefilter")[code].set("1");
   }
  else
   {
    f("scorefilter")[code].set("0");
   }
 }
```

Then the code mask uses *scorefilter* instead of the function call:

```
f("scorefilter")
```

14. Some Useful JScript .NET Objects

This chapter describes some of the objects provided in JScript .NET. The objects that are most useful for scripting in Confirmit, and the properties and methods that are most frequently used, are presented. For descriptions of other JScript .NET Objects, consult a JScript .NET reference manual.

14.1. The Date Object

The `Date` object enables basic storage and retrieval of dates and times, and can be used to set timestamps, do calculations of time differences and to validate dates in the respondent's answers.

The `Date` object type of JScript .NET provides a common set of methods for working with dates and times, either the Local Time or Universal Coordinated Time (UTC). Since we are working with scripts that run on the Confirmit servers (server-side), not on the respondent's PC (client-side), **Local Time** means the time on the Confirmit servers you are working on. **Universal Coordinated Time (UTC)** (sometimes also called "Zulu Time") was formerly called Greenwich Mean Time (GMT) and is the mean solar time at the prime meridian (0° longitude).

Date values are stored internally as the number of milliseconds since January 1, 1970 UTC. For dates before this date, this will be a negative number. The range of dates that can be represented in an instance of the Date object is approximately 285,616 years on either side of January 1, 1970. This should be sufficient for most of us.

14.1.1. Constructors

You can create a new instance of the `Date` object in three ways:

```
dateObj = new Date();
dateObj = new Date(dateVal);
dateObj = new Date(year, month, date{, hours{, minutes{,
seconds{,ms}}}});
```

If you just use the constructor `Date()`, the instance of the `Date` object will be set to current date and time.

If you use `Date(dateVal)` and *dateVal* is an integer, *dateVal* represents the number of milliseconds since midnight January 1, 1970 Universal Coordinated Time. Negative numbers indicate dates prior to 1970. If *dateVal* is a string, *dateVal* is parsed according to the rules in the parse method (see parse on page 155 for more information)).

Here are two ways of getting an instance of the `Date` Object that is set to midnight 2000 (UTC):

```
d = new Date(946684800000);
d = new Date("Sat, 1 Jan 2000 00:00:00 UTC");
```

When you use `Date(`*year*`,`*month*`,`*date*`{,`*hours*`{,`*minutes*`{,`*seconds*`{,`*ms*`}}}})`, the first three arguments are required. *year* has to be the full year, for example 1984 (and not just 84). *month* is an integer between 0 and 11 (This means that January is 0, December is 11. This is similar to how we have seen that arrays are indexed, where 0 is the first element). *date* is an integer between 1 and 31.

The next arguments are optional (that's what the curly brackets indicate), but when one of them is included, all preceding arguments must be supplied. This means that if you specify *seconds*, it has to be preceded by *hours* and *minutes*. *hour* is an integer from 0 to 23 (midnight to 11pm). *minutes* and *seconds* are integers from 0 to 59 and *milliseconds* is an integer from 0 to 999. Here is an example:

```
d = new Date(2000,0,1,0,0,0,0);
```

This may or may not be equal to the two examples above using the other constructors. It depends on the server settings on the Confirmit server. The two examples above both use UTC. With this last constructor, the time zone cannot be specified. The time zone of the Confirmit server will be used.

If an argument is greater than its range or is a negative number, other stored values are modified accordingly. So 150 seconds will be redefined as 2 minutes and 30 seconds. The 2 minutes will be added to the *minutes* value. If this exceeds 60 minutes, hours will be modified and so on.

14.1.2. Date Object Methods

The `Date` object has two **static** methods. They are called static because they are called without creating an instance of the `Date` object. They are `parse` and `UTC`. The other methods are invoked as methods of a created instance of the `Date` object. In code below *dateObj* is an instance of a `Date` object.

14.1.2.1. Static Methods

14.1.2.1.1. parse

Parses a string containing a date, and returns the number of milliseconds between that date and midnight, January 1, 1970.

```
Date.parse(dateVal)
```

dateVal is a string containing a date.

The `parse` method returns an integer value representing the number of milliseconds between midnight, January 1, 1970 and the date supplied in *dateVal*.

The `parse` method is a static method of the `Date` object. Because it is a static method, it is invoked as shown in the following example, rather than invoked as a method of a created `Date` object.

```
Date.parse("January 1, 2000 00:00 AM")
```

The following rules govern what the `parse` method can successfully parse:

- Short dates can use either a "/" or "−" date separator, but must follow the month/day/year format, for example "7/20/96".
- Long dates of the form "July 10 1995" can be given with the year, month, and day in any order, and the year in 2-digit or 4-digit form. If you use the 2-digit form, the year must be greater than or equal to 70.
- Any text inside parentheses is treated as a comment. These parentheses may be nested.
- Both commas and spaces are treated as delimiters. Multiple delimiters are permitted.
- Month and day names must have two or more characters. Two character names that are not unique are resolved as the last match. For example, "Ju" is resolved as July, not June.
- The stated day of the week is ignored if it is incorrect given the remainder of the supplied date. For example, "Tuesday November 9 1996" is accepted and parsed even though that date actually falls on a Friday. The resulting Date object contains "Friday November 9 1996".
- JScript .NET handles all standard time zones, as well as Universal Coordinated Time (UTC) and Greenwich Mean Time (GMT).
- Hours, minutes, and seconds are separated by colons, although all need not be specified. "10:", "10:11", and "10:11:12" are all valid.
- If the 24-hour clock is used, it is an error to specify "PM" for times later than 12 noon. For example, "23:15 PM" is an error.
- A string containing an invalid date is an error. For example, a string containing two years or two months is an error.

14.1.2.1.2. UTC

`UTC` returns the number of milliseconds between midnight, January 1, 1970 Universal Coordinated Time (UTC) (or GMT) and the supplied date.

```
Date.UTC(year, month, day{, hours{, minutes{, seconds{,ms}}}})
```

The arguments are equal to those of `the Date(`*year*`,`*month*`,`*day*`{,`*hours*`{,`*minutes*`{,`*seconds*`{,`*ms*`}}}})` constructor (see Constructors on page 154 for more information).

If the value of an argument is greater than its range, or is a negative number, other stored values are modified accordingly. For example, if you specify 150 seconds, JScript .NET redefines that number as two minutes and 30 seconds.

The difference between the `UTC` method and the corresponding `Date` object constructor is that the `UTC` method assumes UTC, and the `Date` object constructor assumes local time. (Local time when doing server side scripting will be the time zone of the Confirmit servers, not the time zone of the client (the PC of the respondent)).

The `UTC` method is a static method. Therefore, a `Date` object does not have to be created before it can be used.

```
Date.UTC(2000,0,1,0,0,0,0)
```

14.1.2.2. Methods for Setting or Retrieving Values of Parts of Dates

The methods with UTC in the method name use Universal Coordinated Time. The other methods use local time, and since we are dealing with server-side scripts that will be the time of the Confirmit server. The following table show these methods grouped based on the date part they operate on. Curly brackets, { and }, are used to indicate optional arguments.

getFullYear()	getUTCFullYear()	setFullYear(*numYear*{, *numMonth*{, *numDate*}})	setUTCFullYear(*numYear*{, *numMonth*{, *numDate*}})
getMonth()	getUTCMonth()	setMonth(*numMonth*{, *numDate*})	setUTCMonth(*numMonth*{, *numDate*})
getDay()	getUTCDay()		
getDate()	getUTCDate()	setDate(*numDate*)	setUTCDate(*numDate*)
getHours()	getUTCHours()	setHours(*numHours*{, *numMinutes*{, *numSeconds*{, *numMilliseconds*}}})	setUTCHours(*numHours*{, *numMinutes*{, *numSeconds*{, *numMilliseconds*}}})
getMinutes()	getUTCMinutes()	setMinutes(*numMinutes*{, *numSeconds*{, *numMilliseconds*}})	setUTCMinutes(*numMinutes*{, *numSeconds*{, *numMilliseconds*}})
getSeconds()	getUTCSeconds()	setSeconds(*numSeconds*{, *numMilliseconds*}}	setUTCSeconds(*numSeconds*{, *numMilliseconds*})
getMilliseconds()	getUTCMilliseconds()	setMilliseconds(*numMilliseconds*)	setUTCMilliseconds(*numMilliseconds*)
getTime()		setTime(*milliseconds*)	
getTimezoneOffset()			

For the methods that set date parts, the other parts of the date are modified accordingly if the value of an argument is greater than its range or is a negative number. So, if you for example use the `setMinutes` method with 62 as argument, minutes will be set to 2 and hours will be increased with 1. If this makes hours exceed its limit, date is changed to the next day, and so on. This gives a very efficient way of working with dates in scripts.

For all methods with *`numMonth`* as argument, remember that JScript .NET months use the numbers 0 to 11. 0 is January and 11 is December.

14.1.2.2.1. getFullYear, getUTCFullYear, setFullYear and setUTCFullYear

```
dateObj.getFullYear()
dateObj.getUTCFullYear()
```

`getFullYear` and `getUTCFullYear` return the year value in the `Date` object as an absolute number, thus avoiding the Y2K problem.

```
dateObj.setFullYear(numYear{, numMonth{, numDate}})
dateObj.setUTCFullYear(numYear{, numMonth{, numDate}})
```

setFullYear and setUTCFullYear set the year value in the `Date` object.

`numYear` is required and is a numeric value equal to the year.

`numMonth` and *`numDate`* are both optional. If *`numDate`* is supplied, *`numMonth`* must also be supplied. *`numMonth`* is a numeric value equal to the month (0-11). *`numDate`* is a numeric value equal to the date (1-31).

If you do not specify the optional arguments, the value will be retrieved from the corresponding get method. For example, if *`numMonth`* is not specified, JScript .NET uses the value returned from the `getMonth` or `getUTCMonth` method.

If the value of an argument is greater than its range or is a negative number, other stored values are modified accordingly.

There are two methods called `getYear` and `setYear`. They are obsolete and kept for backwards compability only. It is **not** recommended to use them, because of Y2K problems. Use the FullYear methods instead.

14.1.2.2.2. getMonth, getUTCMonth, setMonth and setUTCMonth

```
dateObj.getMonth()
dateObj.getUTCMonth()
```

`getMonth` and `getUTCMonth` return the month value in the `Date` object. The value returned is an integer between 0 and 11 indicating the month value in the `Date` object. Note that this is different from the conventional way of numbering months (1-12). The numeric value for January is 0 and for December the value is 11, one less than the usual way of numbering the months. If "Jan 1, 2000 00:00:00" is stored in a `Date` object, `getMonth` returns 0.

```
dateObj.setMonth(numMonth{, dateVal}) dateObj.setUTCMonth(numMonth{,
dateVal})
```

`setMonth` and `setUTCMonth` set the month value in the `Date` object. *`numMonth`* is required, and is a numeric value equal to the month (0-11).

`dateVal` is optional. It is a numeric value representing the date. If not supplied, the value from a call to the `getDate` or `getUTCDate` method is used.

If the value of *`numMonth`* is greater than 11 or is a negative number, the stored year is modified accordingly. For example, if the stored date is "Jan 1, 2000" and setMonth(14) is called, the date is changed to "Mar 1, 2001."

14.1.2.2.3. getDay and getUTCDay

```
dateObj.getDay()
dateObj.getUTCDay()
```

`getDay` and `getUTCDay` return the day of the week of the `Date` object.

The value returned from the `getDay` method is an integer between 0 and 6 representing the day of the week and corresponds to a day of the week as follows:

Value	Day of the Week
0	Sunday
1	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday
6	Saturday

14.1.2.2.4. getDate, getUTCDate, setDate and setUTCDate

```
dateObj.getDate()
dateObj.getUTCDate()
```

`getDate` and `getUTCDate` return the day of the month value in a `Date` object. The return value is an integer between 1 and 31 that represents the date value in the `Date` object.

```
dateObj.setDate(numDate)dateObj.setUTCDate(numDate)
```

`setDate` and `setUTCDate` set the day of month value of the `Date` object.

numDate is a numeric value equal to the day of month value. If the value is outside the number of days in the month stored in the Date object or a negative number, the other values are modified accordingly, e.g. so that if the date of the object is January 1st 2000 and you use setUTCDate(32) on that object, you get February 1st 2000.

14.1.2.2.5. getHours, getUTCHours, setHours and setUTCHours

```
dateObj.getHours()
dateObj.getUTCHours()
```

getHours and getUTCHours return the hours value in a Date object.

The value returned is an integer between 0 and 23, indicating the number of hours since midnight.

```
dateObj.setHours(numHours{, numMin{, numSec{,
numMilli}}})dateObj.setUTCHours(numHours{, numMin{, numSec{,
numMilli}}})
```

setHours and setUTCHours set the hours value in a Date object.

The only required argument is *numHours*, which is a numeric value equal to the hours value.

The rest of the arguments are optional. If they are not included, the results from using the corresponding get functions are used. For example, if *numMin* is not provided, the result from getMinutes or getUTCMinutes is used. All preceding arguments must be included if one of the optional arguments is included. For example, if *numSec* is used, *numMin* must also be included in the method call.

numMin is a numeric value equal to the minutes value, *numSec* is a numeric value equal to the seconds value and *numMilli* is a numeric value equal to the milliseconds value.

If the value of an argument is greater than its range or is a negative number, other stored values are modified accordingly. For example, if the stored date is "Jan 1, 2000 00:00:00", and setHours(30) is called, the date is changed to "Jan 2, 2000 06:00:00." Negative numbers have a similar behavior.

14.1.2.2.6. getMinutes, getUTCMinutes, setMinutes and setUTCMinutes

```
dateObj.getMinutes()
dateObj.getUTCMinutes()
```

getMinutes and getUTCMinutes return the minutes value in a Date object.

The value returned is an integer between 0 and 59 equal to the minutes value stored in the Date object.

```
dateObj.setMinutes(numMin{, numSec{,
numMilli}})dateObj.setUTCMinutes(numMin{, numSec{, numMilli}})
```

setMinutes and setUTCMinutes set the minutes value in a Date object.

The only required argument is *numMinutes*, which is a numeric value equal to the minutes value.

The rest of the arguments are optional. If they are not included, the results from using the corresponding get functions are used. For example, if *numSec* is not provided, the result from getSeconds or getUTCSeconds is used. All the preceding arguments must be included if one of the optional arguments is included. If *numMilli* is used, *numSec* must also be included in the method call.

numSec is a numeric value equal to the seconds value and *numMilli* is a numeric value equal to the milliseconds value.

If the value of an argument is greater than its range or is a negative number, other stored values are modified accordingly. For example, if the stored date is "Jan 1, 2000 00:00:00", and setMinutes(62) is called, the date is changed to "Jan 1, 2000 01:02:00." Negative numbers have a similar behavior.

14.1.2.2.7. getSeconds, getUTCSeconds, setSeconds and setUTCSeconds

```
dateObj.getSeconds()
dateObj.getUTCSeconds()
```

getSeconds and getUTCSeconds return the seconds value in a Date object.

The value returned is an integer between 0 and 59 equal to the seconds value stored in the Date object.

```
dateObj.setSeconds(numSeconds{,
numMilli})dateObj.setUTCSeconds(numSeconds{, numMilli})
```

setSeconds and setUTCSeconds set the seconds value in a Date object.

The only required argument is *numSeconds*, which is a numeric value equal to the seconds value.

numMilli is optional. If it is not included, the results from using the corresponding get functions are used (getMillieconds or getUTCMilliseconds).

numMilli is a numeric value equal to the milliseconds value.

If the value of an argument is greater than its range or is a negative number, other stored values are modified accordingly. For example, if the stored date is "Jan 1, 2000 00:00:00", and setSeconds(124) is called, the date is changed to "Jan 1, 2000 00:02:04." Negative numbers have a similar behavior.

14.1.2.2.8. getMilliseconds, getUTCMilliseconds, setMilliseconds and setUTCMilliseconds

```
dateObj.getMilliseconds()
dateObj.getUTCMilliseconds()
```

getMilliseconds and getUTCMilliseconds return the milliseconds value in a Date object.

The millisecond value is an integer from 0 to 999.

```
dateObj.setMilliseconds(numMilli) dateObj.setUTCMilliseconds(numMilli)
```

setMilliseconds and setUTCMilliseconds set the milliseconds value in the Date object.

numMilli is a numeric value equal to the milliseconds value.

If the value of *numMilli* is greater than 999 or is a negative number, the stored number of seconds (and minutes, hours, and so forth if necessary) is modified accordingly.

14.1.2.2.9. getTime and setTime

```
dateObj.getTime()
```

getTime returns an integer value representing the number of milliseconds between midnight, January 1, 1970 UTC and the time value in the Date object. Negative numbers indicate dates prior to 1970.

```
dateObj.setTime(milliseconds)
```

setTime sets the date and time of the instance of the Date object. *milliseconds* is an integer value representing the number of elapsed seconds since midnight, January 1, 1970 UTC. If *milliseconds* is negative, it indicates a date before 1970.

14.1.2.2.10. getTimezoneOffset

```
dateObj.getTimezoneOffset()
```

Returns the difference in minutes between the time on the host computer (the Confirmit server when doing server-side scripting) and Universal Coordinated Time (UTC). This number will be positive if you are behind UTC (e.g., Pacific Daylight Time), and negative if you are ahead of UTC (e.g., Japan).

14.1.2.3. Conversion Methods

14.1.2.3.1. valueOf

```
dateObj.valueOf()
```

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

14.1.2.3.2. toLocaleString, toString and toUTCString

```
dateObj.toLocaleString()
```

toLocaleString returns a date converted to a string using the current locale. This mean that it will give the Confirmit server time, and the date is formatted according to the Regional Settings on the server.

```
dateObj.toUTCString()
```

toUTCString returns a string that contains the date formatted using Universal Coordinated Time (UTC) convention.

```
dateObj.toString()
```

toString returns the textual representation of the date, in the time of the Confirmit server, but not reflecting the locale settings.

Here is an example of the output from the three different methods from a server with English (United States) as locale and where the time zone is EST (like Confirmit's Confirmit ASP servers):

Method	Output
toUTCString	Sat, 1 Jan 2000 00:00:00 UTC
toLocalString	Friday, December 31, 1999 7:00:00 PM
toString	Fri Dec 31 19:00:00 EST 1999

There also exists a method called `toGMTString`, which returns a date converted to a string using Greenwich Mean Time(GMT). This method is obsolete, and it is recommended that you use the `toUTCString` method instead.

14.1.3. Confirmit Date Functions and Date Questions

Confirmit provides six functions for working with dates. `InterviewStart` and `InterviewEnd` return the interview_start and interview_end time stamps automatically recorded when the respondent starts the interview and reach the end of the interview or a stop node (the end page). Set`InterviewStart` and Set`InterviewEnd` set these timestamps. `IsDate` and `IsDateFmt` are used for validation of dates.

If a survey uses the Optimized database format introduced in (see the Authoring User Guide for further details), a date question type is available. Such date variables can be used both in the surveys when asking respondents for dates (for examples date of birth), and also as "hidden" variables to set and store dates through scripting. The date question type has built in validation and a calendar pop-up interface, so it is not necessary to use the IsDate and IsDateFmt functions for validation of the respondent's answer when using this question type.

Note: The Date question property is only available when using the Optimized Database format.

14.1.3.1. InterviewStart and InterviewEnd

Both of these functions return dates, however they will return the dates as strings. The syntax you can use to get instances of the JScript .NET Date object from `InterviewStart` and `InterviewEnd` is as below:

```
dateObj = InterviewStart();
dateObj = InterviewEnd();
```

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. Assume 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 a numeric question with hidden property (*time_part1*).

```
if(!f("time_part1").toBoolean())
 {
  var d1 = InterviewStart()
  var d2 = new Date()

  var diff = (d2.getTime() - d1.getTime())/1000
  f("time_part1").set(diff)
 }
```

The if-condition is there to ensure that the time is only 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 must ensure you use the correct settings on the hidden numeric question. Decimal places 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. Total digits has to be set high enough to be able to store both the 3 digits after the decimal point and the highest number of seconds a respondent will use.

If you want to calculate the response time for the next part of the questionnaire as well, you can for example use the following script to set that in another hidden numeric question (*time_part2*):

```
if(!f("time_part2").toBoolean())
 {
  var d1 = InterviewStart();
  var d2 = new Date();

  var diff = ((d2.getTime() - d1.getTime())/1000)-
f("time_part1").toNumber();
  f("time_part2").set(diff);
 }
```

14.1.3.2. IsDateFmt and IsDate

These functions are also described in IsDateFmtAndIsDate. 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).

In this example we have an open text question and want the respondent to enter a date in the format YYYY-MM-DD. We want to validate that the format is correct and that the date is a valid date, and also that the date is not after the current date.

This can be done with the following script in the validation code field of the open text question (*date1*).

```
var d = IsDateFmt(f("date1").get(),"YYYY-MM-DD");

if(!d)
 {
  RaiseError();
  SetQuestionErrorMessage(LangIDs.en,"Invalid date. Please correct
using the format YYYY-MM-DD");
 }
else
 {
```

```
  var dt = new Date();
  dt.setFullYear(d.year,d.month-1,d.day);
  var current = new Date();
  if(dt.valueOf() > current.valueOf())
   {
    RaiseError();
    SetQuestionErrorMessage(LangIDs.en,"Please do not enter a date
after the current date.");
   }
 }
```

If the date provided is not valid, `IsDateFmt(f("date1").get(),"YYYY-MM-DD")` will return `null`. Used in a condition `null` will yield the Boolean value `false`. If the date is valid, `IsDateFmt` will return an object. Used in a condition this will yield `true`. So the first `if` condition, with `!d` , makes sure there will come an error message only when the date is invalid or wrong format is used.

It is important that we include the rest of the script in an else branch because the properties `year`, `month` and `day` are not defined if `null` is returned. So we need to make sure that that part of the script only is run when we have an object in the variable `d`, to prevent script errors.

Then we introduce two instances of the `Date` object, and set one of them (`dt`) to the date answered, and the other (`current`) to the current date. We use the `setFullYear` method to set the date. Observe that we have to subtract 1 from the month, since the `Date` object uses the integers 0-11 for month, whereas the `month` property in the object returned from `IsDateFmt` uses the more conventional integers 1-12.

Validating that a Date with Drop-downs 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 for example be set up in a 3D grid with 3 grid questions as drop-downs like this:

Note: The codes of the questions must be numeric and the same as the legal `year`/`month`/`day` values. For *year* the codes must be the full value (e.g. 2002), for *month* they must be numbers between 1 and 12, and for *day* they must be numbers between 1 and 31 (see IsDateFmt and IsDate on page 124 for more information).

Let us say that we want to check that the answer is a valid date (so that e.g. February 30th is not allowed), and that it is a date in the next two weeks from the current date.

```
var d =
IsDate(f("day")["1"].get(),f("month")["1"].get(),f("year")["1"].get())
;

if(!d)
 {
  RaiseError();
  SetQuestionErrorMessage(LangIDs.en,"Invalid date. Please correct.");
 }
else
 {
  var dt = new Date();
  dt.setFullYear
(d.year,d.month-1,d.day);
  var current = new Date();
  var limit = new Date();
  limit.setDate(current.getDate()+14)
  if(dt.valueOf() < current.valueOf() || dt.valueOf() >
limit.valueOf())
   {
    RaiseError();
    SetQuestionErrorMessage(LangIDs.en,"Please select a date within
the next two weeks.");
   }
 }
```

This script takes advantage of the fact that when using `setDate`, the month and possibly also year value will automatically be updated if the value for date is outside the limit for the month. So if the result of `current.getDate()+14` is a number larger than the number of days in a month, `limit` will be set to a date in the next month. For example, if current date is April 20th 2002, the limit will be 4th of May 2002 (April 20th + 14 days).

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 (*date3*) asking for a date. The validation can be done like this:

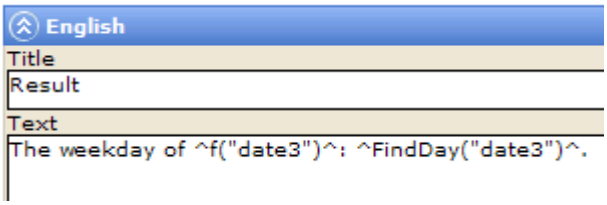
```
var d = IsDateFmt(f("date3").get(),"YYYY-MM-DD");
if(!d)
 {
  RaiseError();
  SetQuestionErrorMessage(LangIDs.en,"Invalid date. Please correct.
Use the format YYYY-MM-DD");
 }
```

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

```
function FindDay(qID) : String
 {
  var d = IsDateFmt(f(qID).get(),"YYYY-MM-DD");
  var dt = new Date();
  dt.setFullYear(d.year,d.month-1,d.day);
  var days = new Array

("Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday
")
  return days[dt.getDay()]
 }
```

After the open text question you can have an info node like this:



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

Exercise 7: Course Registration

Let us say you want to build a questionnaire where people can register for weekly courses that are held on Mondays. You want the respondents to provide the date they want to register for in an Open Text Question, but have to validate the date format (which should be "MM/DD YYYY") and check that the date provided is a Monday and it is after the current date.

Write a validation script for this question. Question ID is *date4*.

The answer is given in APPENDIX A Answers to Exercises.

Converting Data of Birth into Age (in number of years)

If you have an open text question *dob* where respondents insert their date of birth, you can calculate the respondent's age and set it in a hidden question *age* with the following script, which is placed in the validation code of the question:

```
var d = IsDateFmt(f("dob").get(),"YYYY-MM-DD");
if(!d)
 {
  RaiseError();
  SetQuestionErrorMessage(LangIDs.en,"Invalid date. Please use the
format YYYY-MM-DD");
 }
else
 {
  var birthday = new Date();
  birthday.setFullYear(d.year,d.month-1,d.day);
  var today = new Date();
  var years = today.getFullYear() - birthday.getFullYear();
  birthday.setYear(today.getFullYear());
  // If your birthday hasn't occurred yet this year, subtract 1.
  if(today < birthday)
  {
   years-- ;
  }
 f("age").set(years);
}
```

14.1.3.3. The Date Question Type

If a survey uses the Optimized Database format introduced in Confirmit 12.5, a date question type is available that will give both a respondent interface for selecting a date and storing it in the response database as a datetime variable,

The methods and properties available on the form object returned from f("q1") when q1 is a date question, are described previously in chapters "Conversion Methods in Confirmit ", "Methods of the Form Objects" and "Confirmit's f Function". This section provides some examples of using these methods.

Note: The Date question type is only available when using the Optimized Database format.

Set Current Date

If you would like to set current date in a hidden date question *today*, this can be done with the following script:

```
if(!f("today").toBoolean())
{
 f("today").set(new Date());
}
```

Validating that Date answered is not after Current Date

In this example we have a date question *date1*. We want to validate that the date is not after the current date. This can be done with the following script in the validation code field of the date question (date1).:

```
var dt = new Date(f("date1").year(),f("date1").month()-
1,f("date1").day());
var current = new Date();
if(dt.valueOf() > current.valueOf())
 {
 RaiseError();
 SetQuestionErrorMessage(LangIDs.en,"Please do not enter a date after
the current date.");
 }
```

Here we declare two instances of the Date object, and set one of them (dt) to the date answered, and the other (current) to the current date.

Converting Date of Birth into Age (in number of years)

If you have an date question *dob* where respondents insert their date of birth, you can calculate the respondent's age and set it in a hidden question age with the following script, which is placed in the validation code of the question:

```
var birthday = new Date(f("dob").year(),f("dob").month()-
1,f("dob").day());
var today = new Date();
var years = today.getFullYear() - birthday.getFullYear();
birthday.setYear(today.getFullYear());
// If your birthday hasn't occurred yet this year, subtract 1
if(today < birthday)
{
 years-- ;
}
f("age").set(years);
```

14.2. The Math Object

The `Math` object provides a standard library of mathematical constants and functions. The `Math` object cannot be created using the `new` operator, and gives an error if you attempt to do so. This is because it is a built-in object and not an object type. Most of the properties and methods are different mathematical constants and functions that you will seldom need to use in your questionnaires. The most important methods are the rounding methods (see Rounding on page 166 for more information) and the random method (see Random on page 167 for more information).

14.2.1. Properties

The properties of the `Math` object are different mathematical constants.

```
Math.E
```

Euler's constant, the base of natural logarithms. The `E` property is approximately equal to 2.718.

```
Math.LN2
```

The natural logarithm of 2. The `LN2` property is approximately equal to 0.693.

```
Math.LN10
```

The natural logarithm of 10. The `LN10` property is approximately equal to 2.302.

```
Math.LOG2E
```

The base-2 logarithm of e, Euler's constant. The `LOG2E` property is approximately equal to 1.442.

```
Math.LOG10E
```

The base-10 logarithm of *e*, Euler's constant. The `LOG10E` property is approximately equal to 0.434.

```
Math.PI
```

π, the ratio of the circumference of a circle to its diameter, approximately 3.141592653589793.

```
Math.SQRT1_2
```

The square root of ½, or one divided by the square root of 2. The `SQRT1_2` property is approximately equal to 0.707.

```
Math.SQRT2
```

The square root of 2. The `SQRT2` property is approximately equal to 1.414.

14.2.2. Math Object Methods

14.2.2.1. Trigonometric Functions

```
Math.cos(x)
```

returns the cosine of a numeric expression x.

```
Math.sin(x)
```

returns the sine of a numeric expression x.

```
Math.tan(x)
```

returns the tangent of a numeric expression x.

```
Math.acos(x)
```

returns the arc cosine of a numeric expression x.

```
Math.asin(x)
```

returns the arc sine of a numeric expression x.

```
Math.atan(x)
```

returns the arctangent of a numeric expression x.

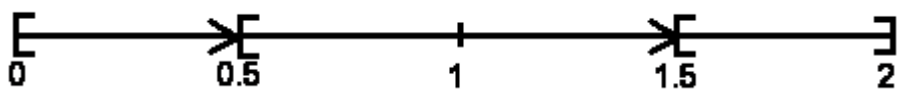
```
Math.atan2(x,y)
```

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

14.2.2.2. Rounding

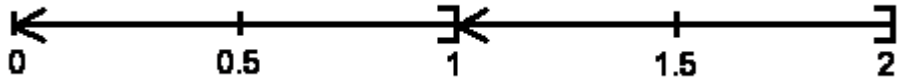
```
Math.round(x)
```

returns a supplied numeric expression x rounded to the nearest integer. If the decimal portion of the number is 0.5 or greater, the return value is equal to the smallest integer greater than the number. Otherwise, round returns the largest integer less than or equal to the number. This is exactly as conventional rounding. For example, 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. Values from and including 1.5 up to and including 2, will be rounded to 2.



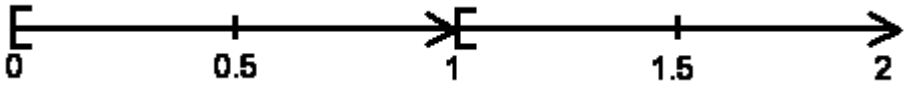

```
Math.ceil(x)
```

returns the smallest integer greater than or equal to its numeric argument x, i.e. rounding **up** to the nearest integer. For example, 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 up to 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. For example, 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. The value 1 and all values from 1 and up to but not including 2, will be rounded to 1, and the value 2 will be rounded to 2.



Rounding to a Number with Two Digits

If we ask people to provide their yearly salary and we want to compute the monthly salary, this can be done by dividing the yearly salary by 12. However this might result in a number with many decimals. If for example we want the result with an accuracy of two digits and need to round it, we can do that by multiplying the result by 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 as below to set the *monthly* value:

```
var yearly : int = f("yearly").toNumber();
var monthly : float = yearly/12;
f("monthly").set(Math.round(monthly*100)/100);
```

14.2.2.3. Random

```
Math.random()
```

returns a random number between 0 and 1 (float), e.g. 0.523. The number generated is from 0 (inclusive) to 1 (exclusive), that is, the returned number can be zero, but it will always be less than one.

This is an extremely powerful feature to use in your surveys when you want to pick responses randomly, or send random respondents to different parts of the questionnaire. Combined with conditions and arithmetic operations you can program extremely powerful solutions for the selections (see SetRandomCategories on page 82 for more information).

Picking n Random Items from the Answers to a Multi Question

If you have a multi question where the respondent picks some brands and want to proceed with more detailed questions on some of the brands the respondent chooses, but not all, you can use a script to randomly pick some of the brands the respondent selected. Let's say the multi question has question ID brands, and we have a hidden multi question with the same answer list that has question ID *random_brands*.

```
var fromForm = f("brands");
var toForm = f("random_brands");
const numberOfItems : int = 3;

var available = new Set();
available = available.union(fromForm);
var selected = new Set();
if(available.size() <= numberOfItems)
 {
  selected = available;
 }
else
 {
  while(selected.size() < numberOfItems)
   {
    var codes = available.members();
    var randomNumber : float = Math.random()*codes.length;
    var randomIndex : int = Math.floor(randomNumber);
    var selectedCode = codes[randomIndex];
    available.remove(selectedCode);
    selected.add(selectedCode);
   }
 }

toForm.set(selected);
```

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 set the selected answers in the *random_brands* multi.

If there number of responses is less than or equal to `numberOfItems` (here: 3), we obviously include all those responses. But if there are more than 3 responses, we have to pick random responses from the answers given.

The random picking is done in this `while` loop:

```
  while(selected.size() < numberOfItems)
   {
    var codes = available.members();
    var randomNumber : float = Math.random()*codes.length;
    var randomIndex : int = Math.floor(randomNumber);
    var selectedCode = codes[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 code 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 `codes` with the available codes. The `members` method converts a set to an array.

Let us say we start with the codes "1","3","4" and "8". The array `codes` will consist of the following items in the first iteration:

```
codes[0] = "1"
codes[1] = "3"
codes[2] = "4"
codes[3] = "8"
```

`Math.random` will give a number between 0 (inclusive) and 1 (exclusive). When this is multiplied with `codes.length` (4 in our example), `randomNumber` will be set to a number between 0 (inclusive) and `codes.length` (4) (exclusive):

```
0<=randomNumber<codes.length
i.e.
0<=randomNumber<4
```

in our example. Let us say that the number returned from `Math.random` is 0.6283. Then `randomNumber` will be 4*0.6283 = 2.5132.

This number is rounded down to the nearest integer by using the `Math.floor` method in the next step:

```
var randomIndex : int = Math.floor(randomNumber);
```

This means that `randomIndex` will be one of 0,1,2,...,`codes.length`-1, i.e. 0,1,2,3.

Important
`Math.floor` must be used, not `Math.round` or `Math.ceil`. Using `Math.floor` is the only way to ensure that the probability of selecting the indexes is the same for all of them and that you get no errors.

`Math.round` would extend the set of possible picks with the index `codes.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.

`codes[2]` is "4", so the code 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 codes, so in our example `codes` will be have the following items in the next iteration:

```
codes[0] = "1"
codes[1] = "3"
codes[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.

Randomly Assigning which Part of a Survey the Respondents Should Answer

To limit the number of questions each respondent has to answer in a long survey, you may want to split the survey into different parts, and randomly pick which part a particular respondent should answer.

This can be done by randomly setting the response to a hidden single question, and then route the respondents to their questions with conditions on this hidden question.

Let us say the hidden single question has question ID *part*. *part* can be set at the beginning of the survey with a script like this:

```
var form = f("part");
if(!form.toBoolean())
 {
  var codes = form.domainValues();
  var randomNumber : float = Math.random()*codes.length;
  var randomIndex : int = Math.floor(randomNumber);
  var code = codes[randomIndex];
  form.set(code);
 }
```

This is very similar to the previous example. Here we pick the code from an array of all codes 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.

See also the built-in function SetRandomCategories (see SetRandomCategories on page 82 for more information).

14.2.2.4. Maximum and Minimum

```
Math.max({number1{, number2{...{, numberN}}}})
Math.min({number1{, number2{...{, numberN}}}})
```

`max` returns the greater of zero or more supplied numeric expressions. `min` returns the lesser of zero or more supplied numeric expressions. The curly brackets are used to indicate that the numerical expressions $number_1,...,number_N$ are optional.

If no arguments are provided, the return value is equal to negative infinity for `max` and positive infinity for `min`. If any argument is `NaN`, the return value is also `NaN` (Not a Number).

However, we would recommend using the Confirmit functions `Max` and `Min`(see Max and Min on page 72 for more information) when working on questions, since these functions automatically converts the answers to numbers.

14.2.2.5. Absolute value

```
Math.abs(number)
```

`abs` returns the absolute value of a numeric expression number.

`abs` can for example be used when you want the difference between two numbers as a positive number, no matter which of them is the highest number.

```
Math.abs(x-y)
```

If `x` is 10 and `y` is 4, `x-y` will return 6. If `x` is 4 and `y` is 10, `x-y` will return –6. The absolute value will be 6 for both.

14.2.2.6. Exponents, Logarithms and Square Root

```
Math.exp(number)
```

`exp` returns e (the base of natural logarithms) raised to a power, e^{number}.

e is Euler's constant, approximately equal to 2.178.

```
Math.log(number)
```

`log` returns the natural logarithm of a *`number`*. The base is e, Euler's constant, approximately equal to 2.178.

```
Math.pow(base, exponent)
```

`pow` returns the value of a base expression taken to a specified power, $base^{exponent.}$

```
Math.sqrt(number)
```

Returns the square root of a numeric expression number. If *`number`* is negative, the return value is zero.

14.3. The String Object

The `String` object type allows strings to be accessed as objects. It allows manipulation and formatting of text strings and determination and location of sub-strings within strings.

14.3.1. Constructors

An instance of the `String` object can be created like this:

```
newString = new String({"stringLiteral"});
```

The curly brackets are used to indicate that the string literal is optional.

`String` objects can also be created implicitly using string literals.

```
newString = "{stringLiteral}";
```

or

```
newString : String = "{stringLiteral}";
```

14.3.2. Properties

```
strVariable.length
"String Literal".length
```

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

14.3.3. Index

Many of the methods of the `String` object refer to **index** on a string. The index is used to refer to a characters position within a string. If you have the string

```
0 1 2 3 4 5 6 7 8 9101112131415
"This is a string"
```

the character a has index 8, because the index starts at 0 for first character. The index of the last character will always be 1 less than the string's length. This is similar to indexing in arrays.

14.3.4. Converting to String from Other Types

Often you want to convert variables of different types to string to use some of the string methods to manipulate on the content. The easiest way of converting a variable to a string, is to concatenate it with an empty string using the + operator:

```
variable+""
```

Since string has higher order of precedence than other types, this will convert the variable to type string. You can also use the `.toString()` method.

14.3.5. Methods for String Objects

Some methods for the string objects that use Regular Expressions are described elsewhere in the manual (see String Object Methods that Use Regular Expression Objects on page 184 for more information). The remainder are described below.

14.3.5.1. Methods Returning a Character or a Character Code at a Specific Index

```
strObj.charAt(index)
```

`charAt` returns the character at the specified *`index`*. Valid values for *`index`* are between 0 and the length of the string minus 1. `charAt` with an index out of valid range returns an empty string.

```
strObj.charCodeAt(index)
```

`charCodeAt` returns an integer representing the Unicode encoding of the character at the specified *`index`*. *`index`* is a number between 0 and the length of the string minus 1. If there is no character at the specified index, `NaN` is returned.

14.3.5.2. Building a String from a Number of Unicode Characters

```
String.fromCharCode({code1{, code2{, ...{, codeN}}}})
```

`fromCharCode` returns a string from a number of Unicode character values. If no arguments are supplied, the result is the empty string.

A String object need not be created before calling `fromCharCode`. The method can be applied directly on the object type.

In the following example, `txt` is set to the string "Confirmit":

```
var txt = String.fromCharCode(67,111,110,102,105,114,109,105,116);
```

This is a way to be able to set Unicode text in scripts.

14.3.5.3. Changing Case

```
strObj.toLowerCase()
```

`toLowerCase` returns a string where all alphabetic characters have been converted to lowercase.

```
strObj.toUpperCase()
```

`toUpperCase` returns a string where all alphabetic characters have been converted to uppercase.

Both of these methods have no effect on non-alphabetic characters.

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 Validation Code on page 6 for more information).

The user name and password can be given in two open text questions, *username* and *password*, the latter with the password property. If you want the password to be case sensitive, but not the username you can use a validation code like this:

```
if(f("username").get().toUpperCase() != "USERNAME" ||
f("password").get() != "Password")
 {
  RaiseError();
  SetQuestionErrorMessage(LangIDs.en,"Wrong username or password.
Please try again.");
 }
```

14.3.5.4. Searching for a Substring within a String

```
strObj.indexOf(subString{, startIndex})
```

`indexOf` returns the character position of the first occurrence of a string *`subString`* within a String object. *`startIndex`* is optional, and is an integer value specifying the index to begin the search. If omitted, searching starts at the beginning of the string. If the *`subString`* is not found, -1 is returned.

If *`startIndex`* is negative, *`startIndex`* is treated as zero. If it is larger than the greatest character position index, it is treated as the largest possible index.

Searching is performed from left to right.

```
strObj.lastIndexOf(substring{, startindex})
```

returns the character position of the last occurrence of a *`subString`* within a `String` object. *`startIndex`* is optional, and is an integer value specifying the index to begin searching within the `String` object. If omitted, searching begins at the end of the string. If the *`subString`* is not found, a-1 is returned.

If *`startIndex`* is negative, *`startIndex`* is treated as zero. If it is larger than the greatest character position index, it is treated as the largest possible index.

Searching is performed right to left.

On the Fly Recoding

You may use the `indexOf` method to search for strings within a text. This can be done to simplify the coding. For example if you have an open text question about car brands you can search for strings like VOLVO or FORD as below:

```
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 who spell brand names wrongly. You can of course check for different common misspellings ("VOVLO" etc.) but usually you will need some form of manual coding as well.

14.3.5.5. Retrieving a Section of a String (Substring)

```
stringObj.slice(start, {end})
```

returns a section of a string. *`start`* is required and is the index of the first character in the section of *`stringObj`*. *`end`* is optional and is the index after the last character in the section of *`stringObj`*. The `slice` method copies up to, but not including, the element indicated by end.

In this example:

```
var txt = "The methods of the String Object can be used for text
manipulation.";
var section = txt.slice(19,32);
```

`section` will be set to the substring

```
"String Object"
```

If *`start`* is negative, it is treated as *`length+start`* where *`length`* is the length of the string. If *`end`* is negative, it is treated as *`length+end`* where *`length`* is the length of the string. If *`end`* is omitted, extraction continues to the end of *`stringObj`*. If *`end`* occurs before *`start`*, no characters are copied to the new string.

```
stringObj.substr(start {, length })
```

returns a substring beginning at a specified location *`start`* and having a specified *`length`*.

`start` is required, and is the starting position (index) of the desired substring. *`length`* is optional and is the number of characters to include in the returned substring.

If *`length`* is zero or negative, an empty string is returned. If not specified, the substring continues to the end of the string.

```
stingObj.substring(start, end)
```

returns the substring at the specified location within a `String` object.

`start` is the index indicating the beginning of the substring and *`end`* is the index indicating the end of the substring. The `substring` method returns a string containing the substring from *`start`* up to, but not including, *`end`*.

The `substring` method uses the lower value of *`start`* and *`end`* as the beginning point of the substring. For example, *`stringObj`*`.substring(0,3)` and *`stringObj`*`.substring(3,0)` return the same substring.

If either *`start`*or *`end`* is `NaN` or negative, it is replaced with zero.

The length of the substring is equal to the absolute value of the difference between *`start`* and *`end`*. For example, the length of the substring returned in *`stringObj`*`.substring(0,3)` and *`strObj`*`.substring(3,0)` is three.

Replacing Last Comma with "and" in a Listing of Answers

If you refer to a multi question, e.g. *brands*, in response piping with ^'s in a question text, then the last two items in the listing are separated with "and" in English.

```
^f("brands")^
```

If the brands "Ford","Mercedes","Volvo" are answers to the brands question, the string returned will be "Ford, Mercedes and Volvo".

However, when you refer to

```
f("brands").categoryLabels()
```

in a script node, e.g. to include the answers in an email text, the result will be an array with the elements. When this is converted to a string (for example by adding it to a string expression), you get a string that lists the elements separated with commas, but not with "and" between the last two elements: "Ford,Mercedes,Volvo".

The following script will replace the last comma with " and ". (Observe the spaces in front of and after and).

```
var body : String = "";
body += "Here are the answers on the brands question:\n\n"

var form = f("brands");
body += form.categoryLabels();

if(form.size() > 1)
 {
  body = body.substring(0,body.lastIndexOf(",")) + " and " +
body.substring(body.lastIndexOf(",")+1,body.length);
 }
SendMail("interviewer@confirmit.com",f("email"),"Answers",body);
```

If there are two answers or more, the answers string will be set to the substring from the beginning of the `answers` string to (but excluding) the last comma, the string `" and "` and the substring from the character after the last comma to the end of the `answers` string.

14.3.5.6. Splitting and Joining Strings

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

`split` returns the array of strings that results when a string is separated into substrings. *`separator`* is a string or an instance of a Regular Expression object (see The Regular Expression Object on page 184 for more information) 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.
```

Generating an Array from a String with Values

Let us say you send in information of what products the respondent uses with the url to an open survey, for example like this:

http://survey.confirmit.com/wix/pXXXXXXXX.aspx?products=1x15x17x19
http://survey.confirmit.com/wix/pXXXXXXXX.aspx?products=6x11x20

1, 6, 11, 15, 17, 19 and 20 are different product codes, and the respondent can have any number of these. Sending them in like this gives more condensed urls than sending in one value (yes/no) for each product. It could possibly be a very long list of products.

In the Confirmit survey we capture the products list with `Request`(see Request on page 199 for more information), and want to set a hidden multi question products with the values sent in with the url. The *products* multi question uses codes that are equal to the product codes that are sent in with the url.

To be able to set the *products* question, we have to split the string returned from `Request("products")` with "x" as delimiter to get an array with the product codes sent in:

```
var txt : String = Request("products")+"";

var productArray = txt.split("x");
f("products").set(productArray);
```

14.3.5.7. Retrieving the String Value

```
strObj.toString()
strObj.valueOf()
```

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

14.3.5.8. Methods that Add HTML Tags to a String

There are a number of methods available that adds HTML code to your strings. They are listed in the table below. They may for example be used in your error messages, or in expressions that will be displayed in info or question titles, texts or answers (with response piping).

Method	Equivalent to
strVariable.anchor(anchorString)	*strVariable* = '<A NAME="'+ *anchorString* + '">' + *strVariable* + '</A>'
strVariable.big()	*strVariable* = '<BIG>' + *strVariable* + '</BIG>'
strVariable.blink()	*strVariable* = '<BLINK>' + *strVariable* + ' </BLINK>'
strVariable.bold()	*strVariable* = '<B>' + *strVariable* + '</B>'
strVariable.fixed()	*strVariable* = '<TT>' + *strVariable* + '</TT>'

strVariable.fontcolor(colorVal)	*strVariable* = '<FONT COLOR="' + *colorVal* + '">' + *strVariable* + '</FONT>'
strVariable.fontsize(intSize)	*strVariable* = '<FONT SIZE="' + *intSize* + '">' + *strVariable* + '</FONT>'
strVariable.italics()	*strVariable* = '<I>' + *strVariable* + '</I>'
strVariable.link(linkstring)	*strVariable* = '<A HREF="' + *linkstring* + '">' + *strVariable* + '</A>'
strVariable.small()	*strVariable* = '<SMALL>' + *strVariable* + '</SMALL>'
strVariable.strike()	*strVariable* = '<STRIKE>' + *strVariable* + '</STRIKE>'
strVariable.sub()	*strVariable* = '_{' + *strVariable* + '}'
strVariable.sup()	*strVariable* = '^{' + *strVariable* + '}'

14.4. Regular Expressions

Regular Expressions are character-matching patterns that are used to find and/or replace character patterns in strings.

With Regular Expressions, you can:

- Test for a pattern within a string. For example, you can test an input string to see if a telephone number pattern or a credit card number pattern occurs within the string.
- Replace text. You can use a Regular Expression to identify specific text in a string and either remove it completely or replace it with other text.
- Extract a substring from a string based upon a pattern match.

The syntax of Regular Expressions is not easy to understand. If you have problems understanding it, you are advised to check the examples, try to modify them and test what the differences are. The Perl scripting language popularized Regular Expressions, and JScript 's support of Regular Expressions is based on that of Perl. So for further reading about Regular Expressions, you may search for Perl documentation in addition to JScript . NET documentation.

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

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

(similar to Regular Expressions in Perl) or

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

(similar to normal syntax for instantiating an object in JScript .NET)

`pattern` is the pattern to be matched and the optional `flags` is a string containing `i`, and/or `m`. The `i` stands for **ignore case** and the `m` for **multi-line search**.

Important
When defining a Regular Expression with the syntax `/pattern/flags`, do not use quotation marks around the strings. However when using the other notation you must use them.

```
var re = new RegExp("confirmit","i");
```

is the same as

```
var re = /confirmit/i;
```

This matches all occurrences of "confirmit".

14.4.1. Regular Expression Syntax

A Regular Expression is a pattern of text that consists of ordinary characters (for example, letters a through z) and special characters, known as meta characters. The pattern describes one or more strings to match when searching a body of text. The Regular Expression serves as a template for matching a character pattern to the string being searched. This is similar to the expressions used for example when searching in the project list in Confirmit. However, Regular Expressions are far more flexible and complex than that.

14.4.1.1. Ordinary Characters

Ordinary characters consist of all characters that are not explicitly designated as meta characters. This includes:

- all upper- and lowercase alphabetic characters,
- all digits,
- all punctuation marks,
- 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. Note that there is no concatenation operator. All that is required is that you put one character after another.

The match will be found in the string

```
"Those people are harmless."
```

but not in the string

```
"Those people are my family."
```

Even though the second string has the characters a, r and m, they are not joined together, so no match is found.

14.4.1.2. Special Characters

There are a number of meta characters that require special treatment when trying to match them. To match these special characters, you must first escape them, that is, precede them with a backslash character (\).

The table below shows those special characters and their meanings. More detailed explanations are given in the following sections.

Special Character	Comment
$	Matches the position at the end of an input string. If the multi-line (`m`) property is set, `$` also matches the position preceding `\n` or `\r`. (newline or carriage return).
( )	Marks the beginning and end of a sub expression. Sub expressions can be captured for later use.
*	Matches the preceding sub expression zero or more times.
+	Matches the preceding sub expression one or more times.
.	Matches any single character except the newline character \n.
[	Marks the beginning of a bracket expression.

?	Matches the preceding sub expression zero or one time, or indicates a "non-greedy" quantifier.
\	Marks the next character as a special character, a literal, a back-reference, or an octal escape.
^	Matches the position at the beginning of an input string except when used in a bracket expression where it negates the character set. If the multi-line property is set, ^ also matches the position following \n or \r (newline or carriage return).
{	Marks the beginning of a quantifier expression.
|	Indicates a choice between two items (or).

To match any of these characters themselves, they must be preceded with \:

Expression	Matches
\$	$
\(	(
\)	)
*	*
\+	+
\.	.
\[	[
\?	?
\\	\
\^	^
\{	{
|	|

```
/Confirmit\?/
```

matches the string "Confirmit?"

14.4.1.3. Non-Printable Characters

A number of useful non-printing characters are occasionally used. The table below shows the escape sequences used to represent those non-printing characters:

Character	Meaning

\c*x*	Matches the control character indicated by x. For example, `\cM` matches a Control-M or a carriage return character. The value of x must be in the range of A-Z or a-z. If not, c is assumed to be a literal 'c' character.
\f	Matches a form-feed character.
\n	Matches a newline character.
\r	Matches a carriage return character.
\s	Matches any white space character including space, tab, form-feed, etc.
\S	Matches any non-white space character.
\t	Matches a tab character.
\v	Matches a vertical tab character.

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

14.4.1.4. Bracket Expressions

Many times, it's useful to match specified characters from a list. For example, you may want to search a string for chapter headings that are expressed as Chapter 1, Chapter 2, etc.

You can create a list of matching characters by placing one or more individual characters within square brackets ([and]). When characters are enclosed in brackets, the list is called a **bracket expression**.

Within brackets, as anywhere else, ordinary characters represent themselves, that is, they match an occurrence of themselves in the input text. Most special characters lose their meaning when they occur inside a bracket expression. There are some exceptions:

The] character ends a list if it is not the first item. To match the] character in a list, place it first, immediately following the opening [.

The \ character continues to be the escape character. To match the \ character itself, use \\.

Characters enclosed in a bracket expression match only a single character for the position in the Regular Expression where the bracket expression appears. The following Regular Expression matches 'Chapter 1', 'Chapter 2', 'Chapter 3', 'Chapter 4', and 'Chapter 5':

```
/Chapter [12345]/
```

If you want to express the matching characters using a range instead of the characters themselves, you can separate the beginning and ending characters in the range using the hyphen (-) character. The Unicode character value of the individual characters determines their relative order within a range. The following Regular Expression is equivalent to the bracketed list shown above.

```
/Chapter [1-5]/
```

When a range is specified in this manner, both the starting and ending values are included in the range. It is important to note that the starting value must precede the ending value in Unicode sort order.

If you want to include the hyphen character (-) in your bracket expression, you must do one of the following:

Escape it with a backslash:

```
[\-]
```

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

```
[-a-z][a-z-]
```

Create a range where the beginning character value is lower than the hyphen character and the ending character value is equal to or greater than the hyphen. Both of the following Regular Expressions satisfy this requirement:

```
[!--][!-~]
```

i.e. characters from ! to – or from ! to ~.

If you want to find all the characters not in the list or range, you can place the caret (^) character at the beginning of the list. If the caret character appears in any other position within the list, it matches itself, that is, it has no special meaning. The following Regular Expression matches chapter headings with numbers different from 1,2,3,4 and 5 (and any other characters different from 1,2,3,4 and 5):

```
/Chapter [^12345]/
```

The same expressions above can be represented using the hyphen character (-).

```
/Chapter [^1-5]/
```

A typical use of a bracket expression is to specify matches of any upper- or lowercase alphabetic characters or any digits. The following JScript .NET expression specifies such a match:

```
/[A-Za-z0-9]/
```

Character	Description
[xyz]	A character set. Matches any one of the enclosed characters.
[^xyz]	A negative character set. Matches any character not enclosed.
[a-z]	A range of characters. Matches any character in the specified range.
[^a-z]	A negative range characters. Matches any character not in the specified range.

a, x, y and z represent characters.

14.4.1.5. Quantifiers

Sometimes, you don't know how many characters there are to match. In order to accommodate that kind of uncertainty, Regular Expressions support the concept of **quantifiers**. These quantifiers let you specify how many times a given component of your Regular Expression must occur for your match to be true.

The following table illustrates the various quantifiers and their meanings:

Character	Description
*	Matches the preceding sub-expression **zero or more times**.
+	Matches the preceding sub-expression **one or more times**.
?	Matches the preceding sub-expression **zero or one time**.
{n}	Matches **exactly** n times, where n is a non-negative integer..
{n,}	Matches **at least** n times, where n is a non-negative integer
{n,m}	Matches **at least** n and **at most** m times. m and n are non-negative integers, where n <= m.

n and m are integers.

With a large input document, chapter numbers could easily exceed nine, so you need a way to handle chapter numbers with more than one digit. Quantifiers give you that capability. The following JScript .NET Regular Expression matches chapter headings with any number of digits:

```
/Chapter [1-9][0-9]*/
```

The first bracket expression [1-9] makes sure that the first digit is in the range 1-9. The second bracket expression with quantifier [0-9]* searches for 0 or more digits in the range 0-9. The quantifier (*) appears after the range expression.

14.4.1.6. Anchors

All the examples so far have recognized patterns anywhere in a string. But you may want the patterns to match only if they e.g. appear at the beginning of the string. **Anchors** provide that capability.

Anchors allow you to fix a Regular Expression to either the beginning or end of a string. They also allow you to create Regular Expressions that occur either within a word or at the beginning or end of a word. The following table contains the list of Regular Expression anchors and their meanings:

Character	Description
^	Matches the position at the beginning of the input string. If the multi-line property is set, ^ also matches the position following \n or \r (newline or carriage return).
$	Matches the position at the end of the input string. If the multi-line property is set, `$` also matches the position preceding `\n` or `\r` (newline or carriage return).
\b	Matches a word boundary, that is, the position between a word and a space.
\B	Matches a non-word boundary.

You cannot use a quantifier with an anchor. Since you cannot have more than one position immediately before or after a newline or word boundary, expressions such as ^* are not permitted.

To match text at the beginning of a line of text, use the ^ character at the beginning of the Regular Expression. This is different from using the ^ within a bracket expression.

To match text at the end of a line of text, use the $ character at the end of the Regular Expression.

To use anchors when searching for chapter headings, the following JScript .NET Regular Expression matches a chapter heading with up to two following digits that occur at the beginning of a line and where there is no text after the heading:

```
/^Chapter [1-9][0-9]?$/
```

Matching word boundaries is a little different but adds a very important capability to Regular Expressions. A word boundary is the position between a word and a space. A non-word boundary is any other position. The following JScript .NET expression matches the first three characters of the word 'Chapter' because they appear following a word boundary:

```
/\bCha/
```

The position of the '\b' operator is critical here. If it's positioned at the beginning of a string to be matched, it looks for the match at the beginning of the word; if it's positioned at the end of the string, it looks for the match at the end of the word. For example, the following expressions match 'ter' in the word 'Chapter' because it appears before a word boundary:

```
/ter\b/
```

14.4.1.7. Alternation and Grouping

Alternation allows use of the | (pipe) character to allow a choice between two or more alternatives, a bit similar to or in Boolean expressions. Expanding the chapter heading Regular Expression, you can expand it to cover more than just chapter headings – for example sections as well. However, it's not as straightforward as you might think. You might think that the following expressions match one or two digits after either 'Chapter' or 'Section' , between the beginning and ending of a line:

```
/^Chapter|Section [1-9][0-9]?$/
```

Unfortunately, what happens is that the Regular Expressions shown above will have two different parts, so this will be equal to getting a match on either of these two expressions:

```
/^Chapter/
```

or

```
/Section [1-9][0-9]?$/
```

So it will match either the word 'Chapter' at the beginning of a line, or 'Section' and 1-2 numbers at the end of the line.

You can use parentheses to limit the scope of the alternation, that is, make sure that the alternation applies only to the two words, 'Chapter' and 'Section'. However, parentheses are tricky as well, because they are also used to create **sub expressions**. By taking the Regular Expressions shown above and adding parentheses in the appropriate places, you can make the Regular Expression match either 'Chapter 1' or 'Section 3'.

```
/^(Chapter|Section) [1-9][0-9]?$/
```

These expressions work properly except that a by-product occurs. Placing parentheses around 'Chapter|Section' establishes the proper grouping, but it also causes either of the two matching words to be captured for future use. So a sub match is captured. In this example we really do not need that sub match.

In the examples shown above, all you really want to do is use the parentheses for grouping a choice between the words 'Chapter' or 'Section'. You do not necessarily want to refer to that match later. We recommend that unless you really need to capture sub matches, do not use them. Your Regular Expressions will be more efficient since they will not have to take the time and memory to store those sub matches.

You can use ?: before the Regular Expression pattern inside the parentheses to prevent the match from being saved for possible later use. The following modification of the Regular Expressions shown above provides the same capability without saving the sub match.

```
/^(?:Chapter|Section) [1-9][0-9]?$/
```

There are times you'd like to be able to test for a pattern without including that text in the match. For instance, you might want to match the protocol in a URL (like http or ftp), but only if that URL ends with .com. Or maybe you want to match the protocol only if the URL does not end with .edu. In cases like those, you'd like to "look ahead" and see how the URL ends. A **lookahead** assertion is handy here.

There are two non-capturing meta characters used for **lookahead** matches:

A **positive lookahead**, specified using ?=, matches the search string at any point where a matching Regular Expression pattern in parentheses begins.

A **negative lookahead**, specified using ?!, matches the search string at any point where a string not matching the Regular Expression pattern begins.

If you want to search for the protocol in a url like "http://www.confirmit.com", but only if it ends with .com:

```
/^[^:]+(?=.*\.com$)/
```

Because of the anchor ^, the match is found at the beginning of the string. Then the first part

```
[^:] +
```

searches for one or more characters different from :. One or more because of the quantifier +, different from : because of the bracket expression [^:]

Then there is a positive lookahead:

```
(?=.*\.com$)
```

which searches through the string to find a match for

```
.* \.com$
```

Because of the anchor $ this has to be at the end of the string. . matches any character except the newline character, and because of the quantifier * we can have 0 or more of these characters before the last part, which is the character . (which has to be escaped with backslash \) followed by com – i.e. ".com".

But the match will be for the first part of the string, i.e. "http" in "http://www.confirmit.com".

Similarly, the expression

```
/^[^:]*(?!.*\.edu$)/
```

will search for the first characters until : is reached in a string not ending with ".edu".

Character	Description
(pattern)	Matches pattern and captures the match. The captured match can be retrieved from the resulting Matches collection, using the $0...$9 properties in JScript .NET.
(?:pattern)	Matches pattern but does not capture the match (it is not stored for possible later use).
(?=pattern)	Positive look-ahead 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 look-ahead.
(?!pattern)	Negative look-ahead 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 look-ahead.

14.4.1.8. Back-References

You can store a part of a matched pattern for later reuse. Placing parentheses around a Regular Expression pattern or part of a pattern causes that part of the expression to be stored into a temporary buffer.

Each captured sub-match is stored as it is encountered from left to right in a Regular Expressions pattern. The sub-matches are numbered beginning at 1 and continuing up to a maximum of 99 sub-matches. Each different buffer can be accessed using

```
\n
```

where n is one or two decimal digits identifying a specific buffer, e.g. \1.

For example, back-references can be used to check for double occurrences of the same words, e.g. in a string such as:

```
Scripting is is fun!
```

The following JScript .NET Regular Expression uses a single sub-expression to check for duplicates:

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

The sub-expression is everything between parentheses. That captured expression includes one or more alphabetic characters, as specified by [a-z]+. The second part of the Regular Expression (\1) is the reference to the previously captured sub-match, that is, the second occurrence of the word. \b is used for word boundary, so that the check is done on complete words.

14.4.1.9. Digits and Word Characters

Character	Description
\d	Matches a digit character. Equivalent to [0-9].
\D	Matches a non-digit character. Equivalent to [^0-9].

\w	Matches any word character including underscore. Equivalent to A-Za-z0-9_].
\W	Matches any non-word character. Equivalent to [^A-Za-z0-9_].

14.4.1.10. Hexadecimal and Octal Escape Values and Unicode Characters

Character	Description
\x*n*	Matches n, where n is a hexadecimal escape value. Hexadecimal escape values must be exactly two digits long. For example, '\x41' matches "A". '\x041' is equivalent to '\x04' & "1". Allows ASCII codes to be used in Regular Expressions.
num	Identifies either an octal escape value or a back-reference (see Back-References on page 182 for more information).
\u*n*	Matches n, where n is a Unicode character expressed as four hexadecimal digits. For example, \u00A9 matches the copyright symbol (©).

Since \ followed by a number *num* can represent both octal escape values and back-references, the following rules apply:

If *num* has one digit and *num* is preceded by at least *num* captured sub expressions, *num* is a back-reference. Otherwise, *num* is an octal escape value if *num* is an octal digit (0-7).

If *num* has two digits and *num* is preceded by at least *num* captured sub expressions, *num* is a back-reference. If the first digit is *n*, and *num* is preceded by at least *n* captures, *n* is a back-reference followed by a literal, the second digit. If neither of these applies, *num* matches an octal escape value when the number is an octal (both digits in the range 0-7).

If *num* has three digits, it matches an octal escape value when the first digit is 0-3 and the two last digits are octal digits (0-7).

14.4.2. Order of Precedence

Once you have constructed a Regular Expression, it is evaluated much like an arithmetic expression, that is, it is evaluated from left to right and follows an **order of precedence**. The following table illustrates, from highest to lowest, the order of precedence of the various Regular Expression operators:

Operator(s)	Description
\	Escape
(), (?:), (?=), []	Parentheses and Brackets
*, +, ?, {*n*}, {*n*,}, {*n*,*m*}	Quantifiers
^, $, *anymetacharacter*	Anchors and Sequences
|	Alternation

14.4.3. The Regular Expression Object

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

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

and

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

Regular Expression objects store patterns used when searching strings for character combinations. After the Regular Expression object is created, it is either passed to a `String` method, or a string is passed to one of the Regular Expression methods.

14.4.3.1. Regular Expression Methods

```
rgExp.exec(str)
```

`exec` executes a search on a string *`str`* using a Regular Expression pattern defined in an instance of a Regular Expression object *`rgExp`*, and returns an array containing the results of that search.

If the `exec` method does not find a match, it returns `null`. If it finds a match, exec returns an array. Element zero of the array contains the entire match, while elements 1 – n contain any sub matches that have occurred within the match.

```
rgExp.test(str)
```

`test` returns a Boolean value (`true` or `false`) that indicates whether or not a pattern defined in the Regular Expression *`rgExp`* exists in a searched string *`str`*.

The `test` method returns `true` if the pattern exists in the string, and `false` otherwise.

Validation of the Format of a Phone Number

Let us say you want the respondent to specify his phone number in a particular format, allowing a digit or + as the first character and spaces between number sets, but no other characters. The following validation code uses a Regular Expression to check the formatting of the phone number, provided that the phone number is applied in an open text question with question ID *phone*:

```
var num : String = f("phone").get();
var re = /^(?:\d|\+)(?:\d| )+$/;
if(!re.test(num))
 {
  RaiseError();
  SetQuestionErrorMessage(LangIDs.en,"Please provide your phone
number, only using digits and space, and with + before your country
code if it is a foreign number.");
 }
```

The Regular Expression starts the search at the beginning of the string (^). Then the first character should be either a digit or + (?:\d|\+). Because of ?: this sub expression is not stored. Then the rest of the string consists of one or more digits or spaces (?:\d|) until the end of the string is reached ($).

In this script there are no restrictions on number of spaces or digits.

Exercise 8: Restricting the Number of Digits in a Phone Number

Based on the previous example, please modify the Regular Expression so that it checks that the phone number is on the format xxx xxx xxxx – i.e. 3 groups of 3+3+4 digits separated with space. (This time with no + for country code).

The answer is given in APPENDIX A Answers to Exercises.

14.4.4. String Object Methods that Use Regular Expression Objects

There are three methods of the `String` Object that uses Regular Expressions as input.

```
stringObj.match(rgExp)
```

`match` executes a search on a string using a Regular Expression pattern, and returns an array containing the results of that search.

If the `match` method does not find a match, it returns `null`.

If it finds a match, `match` returns an array. If the global flag (g) is not set, element zero of the array contains the entire match, while elements 1 – n contain any sub matches that have occurred within the match. If the global flag is set, elements 0 - n contain all matches that occurred.

```
stringObj.replace(rgExp, replaceText)
```

`replace` returns a copy of a string with *`rgExp`* replaced with *`replaceText`*. 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.

Using Regular Expression to Replace Commas With Line Breaks

If you want to send an email and in the email text list the answers to a multi question from the survey, you can use `categoryLabels`. Converted into a string this will give the items separated by commas. If you want to have the answers on one line each instead of separated by commas you have to replace the commas with line breaks. If the email is sent as plain text, you then have to replace the commas with the special character \n.

This script will replace the commas in a listing of the answers given on a multi question brands:

```
var body : String = "";
body += "Here are the answers on the brands question:\r\n\r\n"

var brandsAnswers = f("brands").categoryLabels();
brandsAnswers = brandsAnswers.toString();
brandsAnswers = brandsAnswers.replace(/,/g,"\r\n");

body += brandsAnswers

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

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

```
var body : String = "";
body += "Here are the answers on the brands question:<br><br>"

var brandsAnswers = f("brands").categoryLabels();
brandsAnswers = brandsAnswers.toString();
brandsAnswers = brandsAnswers.replace(/,/g,"<br>");

body += brandsAnswers

SendMail("interviewer@confirmit.com",f("email"),"Answers",body,"","",0
,0);
```

Post Codes in the United Kingdom

Post codes in the UK can be in the following formats:

LN NLL
LLN NLL
LNN NLL
LLNN NLL
LLNL NLL
LNL LLN

where L is a letter and N is a number. A post 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, OR a letter, a number and a letter, AND then a space and two letters and a number.

A validation code for post codes for the UK could be as shown below, with the open text question *postalcode*:

```
var pcode : String = f("postalcode").get();
var re = /^(?:(?:[a-z]{1,2}\d{1,2}|[a-z]{2}\d[a-z]) \d[a-z]{2})|([a-
z]\d[a-z] [a-z]{2}\d)$/i;
if(pcode.search(re) == -1)
 {
  RaiseError();
  SetQuestionErrorMessage(LangIDs.en,"Please provide a post code using
a valid format.");
 }
```

We allow both upper and lower case letters (i). The first 5 combinations are covered by (?:[a-z]{1,2}\d{1,2}|[a-z]{2}\d[a-z]) \d[a-z]{2}). Within this, there are two possibilities for the first part before the blank: 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 of these combinations: A blank, a number and two letters (NLL). The sixth alternative, LNL LLN, is covered in the end: ([a-z]\d[a-z] [a-z]{2}\d). A letter, a digit, a letter, then the blank, two letters and a digit.

Note that not all letters are valid in all positions, so this regular expression will accept some invalid postal codes, but it will at least help the respondent to comply with the basic format of the postal codes.

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 a numeric question with total digits 5, and let the default validation do the work. However, the first number can be a zero (0), and to make sure that it is not removed when the value is stored (as it would for numeric questions), the question should be set up as an open text question with field width 5 instead of as a numeric. The easiest way to validate that all 5 characters are digits, is to use a Regular Expression.

The answer is given in APPENDIX A Answers to Exercises.

14.5. The Array Object

The `Array` object has been used previously in the manual. (see Arrays on page 37 for more information). The methods and properties here can be used on both types.

Typed arrays can be declared as below:

```
var arrayName : type[] = [value0, value1, …, valuen]
or
var arrayName : type[arrayLength];
```

or

```
var arrayName : type[];
arrayName = new type[arrayLength];
```

JScript arrays can be declared as follows:

```
var arrayName = [value0, value1, …, valuen]
```

or

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

or

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

or

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

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

```
arrayName.length
```

The methods of the `Array` object are described in this section.

14.5.1. Combining Arrays

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

`concat` returns a new array consisting of a combination of *arrayName* and any other supplied items. The items to be added (*item1,...,itemN*) 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.

Combining arrays with codes answered on two questions into one array

If you have two multi questions and would like to create an array consisting of all the codes answered on the two multi questions, one possible way is to combine the two arrays you get from the categories method:

```
var q8 = f('q8').categories();
var q9 = f('q9').categories();
var codes = q8.concat(q9);
```

14.5.2. Converting Arrays to Strings

```
array.join(separator)
```

join returns a string value consisting of all the elements of an array concatenated and separated by the specified separator character. If separator is omitted, the array elements are separated with a comma. If any element of the array is undefined or null, it is treated as an empty string ("").

```
array.toString()
array.valueOf()
```

Both toString and valueOf converts elements of an array to strings. The resulting strings are concatenated, separated by commas. This is the same as using join without specifying a separator (or specifying comma as separator).

Converting an Array to a String with Line Breaks between Elements

This example is similar to using `categoryLabels` to list the answers to a multi question and include them in an email text (see String Object Methods that Use Regular Expression Objects on page 184 for more information). Instead of the default commas between the items listed in the array returned from `categoryLabels`, we want line breaks. If the email is sent as plain text, you then have to replace the commas with the special character `\n`.

This script will convert the array to a string and use line breaks as delimiter between the elements of a multi question brands:

```
var body : String = "";
body += "Here are the answers on the brands question:\n\n"

var brandsAnswers = f("brands").categoryLabels();
brandsAnswers = brandsAnswers.join("\n");

body += brandsAnswers

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

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

```
var body : String = "";
body += "Here are the answers on the brands question:<br><br>"

var brandsAnswers = f("brands").categoryLabels();
brandsAnswers = brandsAnswers.join("<br>");

body += brandsAnswers

SendMail("interviewer@confirmit.com",f("email"),"Answers",body,"","",0
,0);
```

14.5.3. Removing and Adding Elements

```
array.pop( )
```

`pop` removes the last element from an array and returns it. If the array is empty, `undefined` is returned.

```
array.push({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.

14.5.4. Changing the Order of the Elements

```
array.reverse()
```

`reverse` returns an `Array` object with the elements reversed. The method reverses the elements of the `Array` object in place (*array*). It does not create a new `Array` object during execution. If the array is not contiguous, the `reverse` method creates elements in the array that fill the gaps in the array. Each of these created elements has the value `undefined`.

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

`sort` returns an `Array` object with the elements sorted. `sortFunction` is optional and is the name of the function used to determine the order of the elements. Because Confirmit script nodes are wrapped inside a class, you have to use the keyword `this` to refer to the current instance (`this.`*sortFunction*). If the sorting function is omitted, the elements are sorted in ascending, ASCII character order. The `sort` method sorts the `Array` object in place; no new `Array` object is created during execution.

If you supply a function in the `this.`*sortFunction* argument, it must return one of the following values:

- A negative value if the first argument passed is less than the second argument.
- Zero if the two arguments are equivalent.
- A positive value if the first argument is greater than the second argument.

Validating Grid with "Other, specify" Alternatives

Assume you have a grid *q1* with two "other, specify"-items:

Quality

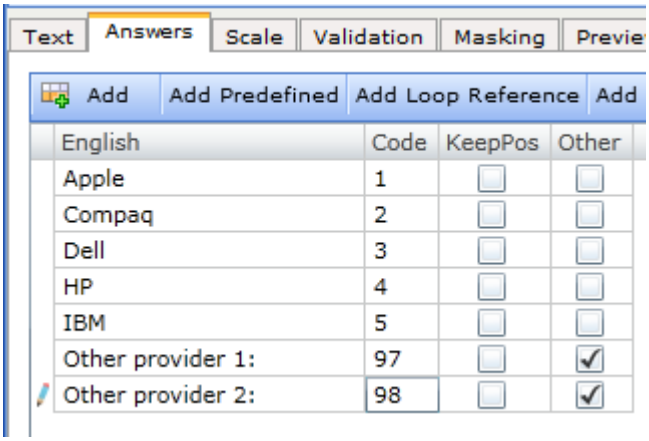
Please give us your impression of the quality of the products provided by these PC manufacturers:

	1 Low quality	2	3	4	5 High quality
Apple	○	○	○	○	○
Compaq	○	○	○	○	○
Dell	○	○	○	○	○
HP	○	○	○	○	○
IBM	○	○	○	○	○
Other provider 1:	○	○	○	○	○
Other provider 2:	○	○	○	○	○

In a question such as this, the "other, specify" items will usually not be required, whereas the remaining items will be required. To achieve this, we must 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 ensure it is turned on when generating web interview files). However, if the respondent accidentally rates one of the "other" items, there is no way for the respondent to unselect that row. We must therefore include 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 codes used:



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

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

var codes = codes.sort(this.NumSort); //sort the codes so the items
are checked in the same order as they are displayed

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

for(var i : int = 0;i<codes.length;i++)
 {
  var code = codes[i];
  if(!form[code].toBoolean())
   {
    //add the label of not answered element to the array:
    notAnswered.push(form[code].label());
   }
 }

codes = new Array("97","98");  //codes of "other, specify"-elements

for(i=0;i<codes.length;i++)
 {
  code = codes[i];
  if(form[code].toBoolean() && !f("q1_"+code+"_other").toBoolean())
   {
    //remove answer
    form[code].set(null);
   }
 }
if(notAnswered.length > 0)
 {
  RaiseError();
  SetQuestionErrorMessage(LangIDs.en,"Please answer for all providers.
Answers are missing 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 `codes` array is set with the statement

```
var codes = 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 codes in the array) to numbers and then do subtraction and return the result. If the result is negative, the first code is less than the second, if it is 0 they are equal and if the result is positive the first one is greater than the second. This is just as the description of the *sortFunction* above.

14.5.5. slice and splice

```
array.slice(start, {end})
```

`slice` returns a section of an array. *start* is the index to the beginning of the specified portion of the array. *end* is optional and is the index to the end of the specified portion of *arrayObj*. The `slice` method copies up to, but not including, the element indicated by *end*. If *start* is negative, it is treated as *length+start* where *length* is the length of the array. If *end* is negative, it is treated as *length+end* where *length* is the length of the array. If *end* is omitted, extraction continues to the end of *arrayObj*. If *end* occurs before *start*, no elements are copied to the new array.

```
array.splice(start, deleteCount, {item1{, item2{, ... {,itemN}}}})
```

`splice` removes elements from an array and, if necessary, inserts new elements in their place, returning the deleted elements. *`start`* is the index from which to start removing elements. *`deleteCount`* is the number of elements to remove. $item_1$,..., $item_N$ are optional elements to insert into the array in place of the deleted elements.

The `splice` method modifies array by removing the specified number of elements from position *`start`* and inserting new elements. The deleted elements are returned as a new `Array` object.

15. Customizing Standard Error Messages

We have seen numerous examples on how to add your own validation code to your Confirmit surveys. This chapter looks into how you can 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 ranking question or a 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 numeric and numeric list questions, the system checks that the answer consists of the symbols 0 to 9 only, and is within the limits defined in total digits, decimal places 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 other than the standard error messages, you must add code in the validation code field of the question where the validation applies. This chapter explains a number of functions that make it easier to build customized error messages for the default answer checks.

15.1. Functions for Standard Validation

The following functions can be used to find out if any of the standard validation in Confirmit has found an error in the respondent's answer(s):

Function	Description
QuestionErrors()	returns true if an error has been raised during any validation, false otherwise.
MissingRequiredError()	returns true if one or more required answers to a question are missing, false otherwise.
ExclusivityError()	returns true if an exclusive answer ("single punch") in a multi has been selected together with any other alternative, false otherwise.
NotSpecifiedError()	returns true if an alternative in an Other-Specify construct has been selected/answered without filling in the associated "Specify" text box, false otherwise.
NotSelectedError()	returns true if a "Specify" value has been entered in the text box of an Other-Specify

	construct without selecting/answering the associated alternative, false otherwise.
RankError()	returns true if the "Ordered" property has been selected for the form and the answers to the questions are non-numeric, do not start at 1 or are non-consecutive, false otherwise.
SizeError()	returns true if one or more open-ended answers exceeded the maximum length specified in field width, false otherwise.
NumericError()	returns true if the “Numeric” property has been selected and the answer is not numeric, false otherwise.
PrecisionError()	returns true if the “Numeric” property has been selected and the answer cannot be stored within the defined total digits, false otherwise.
ScaleError()	returns true if the “Numeric” property has been selected and the answer contains more decimals than in the defined decimal places, false otherwise.
RangeError()	returns true if the “Numeric” property has been selected and the answer is outside the defined range, false otherwise.
PasswordError()	used on a opentext panel variable with the password property set. Returns true if any of the password validation settings for the panel (in panel settings) have not been satisfied.

Example:

```
if(MissingRequiredError())
 {
  SetQuestionErrorMessage(LangIDs.en,"Please provide an answer.");
 }
```

When one of these functions returns true, an error situation has already been flagged, so you do not have to use the RaiseError function.

15.2. Template Based Error Messages

The system allows you to use templates for your error messages in custom validation. For each error type, various elements are filled in that can improve the information content of the messages you report to the respondents. Instead of simply using fixed strings for error messages, the ErrorTemplate function can be used to “plug in” information about the current question, like question texts, current answers etc.

```
ErrorTemplate(spec)
```

spec is a string with the template specification. Inside the spec string you may use different elements that are singled out with caret (^) as pre- and suffix, e.g.

```
^MISSING^
```

Example:

```
SetQuestionErrorMessage(LangIDs.en, ErrorTemplate("^MISSING^ has not
been answered."));
```

This will set the text of the error message to "label(s) has not been answered"

Templates are available for all the standard validation. There are usually several versions of the templates, which differ in how they separate words when they are listed. Some only use commas, some use commas, but "and" or "or" between the last two items. "and"/"or" is also available in several languages, so e.g. in German they will be replaced with "und"/"oder" and so on.

15.2.1. Answer Required Checks

```
MissingRequiredError()
```

returns true when one or more required answers to a question are missing. The following template elements can be used when MissingRequiredError returns true:

Template	Description
MISSING	Labels of all questions that lack answers, comma separated.
MISSING_AND	Same as above, but with the word “and” separating the two last items.
MISSING_OR	Same as above, but with the word “or” separating the two last items.

Example:

```
if(MissingRequiredError())
 {
  SetQuestionErrorMessage(LangIDs.en,
  ErrorTemplate("Please select an answer for ^MISSING_AND^."));
 }
```

15.2.2. Exclusivity Tests

```
ExclusivityError()
```

returns true if an exclusive answer ("single punch") in a multi has been selected together with any other alternative. The following template elements can be used when ExclusivityError returns true:

Template	Description
SEL_EXCL	The labels of all exclusive (single punch) answers that were selected.
SEL_EXCL_AND	Same as above, but with the word “and” separating the two last items.
SEL_EXCL_OR	Same as above, but with the word “or” separating the two last items.
SEL_NEXCL	The labels of all non-exclusive (multi punch) answers that were selected.
SEL_NEXCL_AND	Same as above, but with the word “and” separating the two last items.
SEL_NEXCL_OR	Same as above, but with the word “or” separating the two last items.
DEF_EXCL	The labels of all exclusive (single punch) answers, regardless of which of them that were selected.
DEF_EXCL_AND	Same as above, but with the word “and” separating the two last items.
DEF_EXCL_OR	Same as above, but with the word “or” separating the two last items.
DEF_NEXCL	The labels of all non-exclusive (multi punch) answers, regardless of which of them that were selected.
DEF_NEXCL_AND	Same as above, but with the word “and” separating the two last items.
DEF_NEXCL_OR	Same as above, but with the word “or” separating the two last items.

Examples:

```
if(ExclusivityError())
 {
  SetQuestionErrorMessage(LangIDs.en,
  ErrorTemplate("Please do not check ^SEL_EXCL_OR^ if you check other
answers to the question."));
 }

if(ExclusivityError())
 {
  SetQuestionErrorMessage(LangIDs.en,
  ErrorTemplate("^DEF_EXCL_AND^ can not be combined with any of the
other answers."));
 }
```

15.2.3. Other-Specify Checking

```
NotSpecifiedError()
```

returns true if an alternative in an Other-Specify construct has been selected/answered without filling in the associated "Specify" text box.

```
NotSelectedError()
```

returns true if a "Specify" value has been entered in the text box of an Other-Specify construct without selecting/answering the associated answer alternative.

The following template elements are defined when one of these functions returns true:

Template	Description
OTHER	Label of the alternative/input that requires specification in an Other-Specify construct.
SPEC	The specification

Examples:

```
if(NotSpecifiedError())
 {
  SetQuestionErrorMessage(LangIDs.en,ErrorTemplate("Please specify if
^OTHER^ is chosen."));
 }
if(NotSelectedError())
 {
  SetQuestionErrorMessage(LangIDs.en,
  ErrorTemplate("If you specify ^OTHER^ then please select the ^OTHER^
option."));
 }
```

15.2.4. Rank Order Tests

```
RankError()
```

returns true if the "Ordered" property has been selected for the form and the answers to the questions are non-numeric, do not start at 1 or are non-consecutive. The following template elements are defined when RankError returns true:

Template	Description
RANK_MIN	The label of the first member of the ranking scale, or 1 if the question is open ended.
RANK_MAX	The label of the nth member of the ranking scale, or n if the question is open ended, where n is the

	number of items to rank.

Example:

```
if(RankError())
 {
  SetQuestionErrorMessage(LangIDs.en,
  ErrorTemplate("Please enter consecutive answers in the range
^RANK_MIN^ to ^RANK_MAX^."));
 }
```

15.2.5. Answer Size For Fixed-Width Fields

```
SizeError()
```

returns true if one or more open-ended answers exceeds the maximum length specified in the field width. The following template elements are defined when SizeError returns true:

Template	Description
TOO_LONG	The labels of the inputs that were too long.
TOO_LONG_AND	Same as above, but with the word “and” separating the two last items.
MAX_SIZE	The maximum allowed size.

Example:

```
if(SizeError())
 {
  SetQuestionErrorMessage(LangIDs.en,
  ErrorTemplate("The answers to ^TOO_LONG_AND^ were longer than
^MAX_SIZE^ characters and have been truncated. Please review."));
 }
```

15.2.6. Numeric Validation

```
NumericError()
```

returns true if the "Numeric" property has been selected and the answer is not numeric. The following template elements are defined when NumericError returns true:

Template	Description
NUMERIC_ERRORS	The labels of all elements with a numeric error.
NUMERIC_ERRORS_AND	Same as above, but with the word “and” separating the two last items.

Example:

```
if(NumericError())
 {
  SetQuestionErrorMessage(LangIDs.en,
  ErrorTemplate("Please make sure that the answers to ^NUMERIC_ERRORS^
are numbers only."));
 }
```

15.2.7. Precision Error Tests

```
PrecisionError()
```

returns true if the "Numeric" property has been selected and the answer cannot be stored within the defined total digits. The following template elements are defined when PrecisionError returns true:

Template	Description
PRECISION	The defined total digits
PRECISION_ERRORS	The labels of all elements with a total digits error.
PRECISION_ERRORS_AND	Same as above, but with the word “and” separating the two last items.

Example:

```
if(PrecisionError())
 {
  SetQuestionErrorMessage(LangIDs.en,
  ErrorTemplate("Please enter a number with no more than ^PRECISION^
digits for ^PRECISION_ERRORS^."));
 }
```

15.2.8. Scale Error Tests

```
ScaleError()
```

returns true if the "Numeric" property has been selected and the answer contains more decimals than in the defined decimal places. The following template elements are defined when ScaleError returns true:

Template	Description
SCALE	The defined decimal places
SCALE_ERRORS	The labels of all elements with to many decimals.
SCALE_ERRORS_AND	Same as above, but with the word “and” separating the two last items.

Example:

```
if(ScaleError())
 {
  SetQuestionErrorMessage(LangIDs.en,
  ErrorTemplate("Please enter answers with no more than ^SCALE^
decimals for ^SCALE_ERRORS_AND^."));
 }
```

15.2.9. Range Error Tests

```
RangeError()
```

returns true if the "Numeric" property has been selected and the answer is outside the defined range. The following template elements are defined when RangeError returns true:

Template	Description
RANGE_MIN	The label of the defined minimum value.
RANGE_MAX	The label of the defined maximum value.
RANGE_ERRORS	The labels of all elements with too many decimals.
RANGE_ERRORS_AND	Same as above, but with the word “and” separating the two last items.

Example:

```
if(RangeError())
 {
  SetQuestionErrorMessage(LangIDs.en,
  ErrorTemplate("Please enter answers in the range ^RANGE_MIN^ to
^RANGE_MAX^."));
 }
```

15.2.10. Columns in 3D grid

In 3D grids the CTITLE primitive is available to give the possibility of including the column header (question text) of the question in the error message.

Template	Description
CTITLE	The column header (question text) for the question in the 3D grid.

Example:

```
if(f("q1").size() < 5)
 {
  SetQuestionErrorMessage(LangIDs.en,
  ErrorTemplate("Please select at least 5 items for ^CTITLE^."));
 }
```

16. Useful ASP.NET Intrinsic Objects

This chapter contains examples of some ASP.NET objects that you may find useful in your scripts, with examples of how they can be used. It is not intended as a comprehensive description of them. For more information, consult an ASP.NET reference manual.

16.1. Request

Information can be sent in to the server from the user either from a web form (as the questions in the survey), passed in with the web address URL or stored in a cookie. This, as well as other information about the incoming request can be retrieved using the Request object.

We will look into some of the properties of the Request object. All of these return values of the type `NameValueCollection`. This is a type that represents a sorted collection of associated `String` keys and `String` values that can be accessed either with the key or with the index (integer from 0). This means that their values can be referenced in the same way as you refer to values in an `Array` (see Arrays on page 37 for more information), but also using their `String` key, for example:

```
var aValue = Request("arg");
```

This will look through all values sent in with the request to find a key that matches the name specified in *arg*and return the corresponding value. If there is no match for `arg`, `null` will be returned. This means that you can use the expression

```
Request("arg")!=null
```

to check if it exists before using the value returned.

A more robust approach than using the `Request` object directly is to use the separate properties of the `Request` object to prevent naming conflicts. In the next chapters we will describe some of the most useful properties.

Note: Request and Request.Form are not supported by CAPI. Use RequestForm (see RequestForm on page 199 for more information).

16.1.1. Request.Form

```
Request.Form
```

will give the collection of form values submitted using "POST".

16.1.2. RequestForm

```
RequestForm
```

This was created because Request.Form and Request are not supported by CAPI. RequestForm functions in the same way as Request.Form (see Request.Form on page 199 for more information).

16.1.3. QueryString

```
Request.QueryString()
```

will give the collection of values supplied on the URL or from a form submitted with "GET". If you want to send values into the survey with the url, they can be included after the question mark after *project_ID*.aspx, separated by ampersands (&), e.g.

```
http://survey.confirmit.com/wix/project_ID.aspx?variable1=value1&variable2=value2
```

The keys here are $variable_1$ and $variable_2$, so using the key $variable_1$ as in

```
Request.QueryString("variable1")
```

with the url above will return the value $value_1$.

16.1.4. Request.Cookies

```
Request.Cookies
```

will return the cookies stored from the domain of the Confirmit server. It returns an HttpCookieCollection object (see Cookies on page 202 for more information).

16.1.5. ServerVariables

```
Request.ServerVariables
```

will give the information sent in the header or internal server values.

Here is a table listing an extract of the possible server variables in the collection returned from the `ServerVariables` property:

Variable	Meaning
ALL_HTTP	All HTTP headers sent by the client., with their names capitalized and prefixed with `HTTP_`
ALL_RAW	Retrieves all headers in raw form (the form they were sent to the server by the client).
CONTENT_LENGTH	The length of the content as given by the client.
CONTENT_TYPE	The MIME type of the request, such as www-url-encoded for a form being posted to the server.
GATEWAY_INTERFACE	The Common Gateway Interface (CGI) supported on the server.
HTTP_<*HeaderName*>	The value stored in the header *`HeaderName`*. Any header other than those listed in this table must be prefixed by HTTP_ in order for the ServerVariables collection to retrieve its value. Note that the server interprets any underscore (_) characters in *`HeaderName`* as dashes in the actual header. For example if you specify HTTP_MY_HEADER, the server searches for a header sent as MY-HEADER.
HTTP_ACCEPT	The MIME types the client can accept.
HTTP_ACCEPT_LANGUAGE	The languages accepted by the client.
HTTP_ACCEPT_ENCODING	The compression encoding types supported by the client.
HTTP_CONNECTION	Indicates whether the connection allows keep-alive functionality.
HTTP_COOKIE	Returns the cookie string that was included with the request.
HTTP_USER_AGENT	Returns a string describing the browser that sent the request.
HTTP_HOST	The host name of the server.
HTTPS	Returns "`On`" if HTTPS was used for the request and "`Off`" if not.
HTTPS_KEYSIZE	Number of bits in the encryption used to make the SSL connection. For example, 128.
HTTPS_SECRETKEYSIZE	Number of bits in server certificate private key. For example, 1024.
HTTPS_SERVER_ISSUER	Issuer field of the server certificate.
HTTPS_SERVER_SUBJECT	Subject field of the server certificate.
LOCAL_ADDR	The IP address of the server which is handling the request.

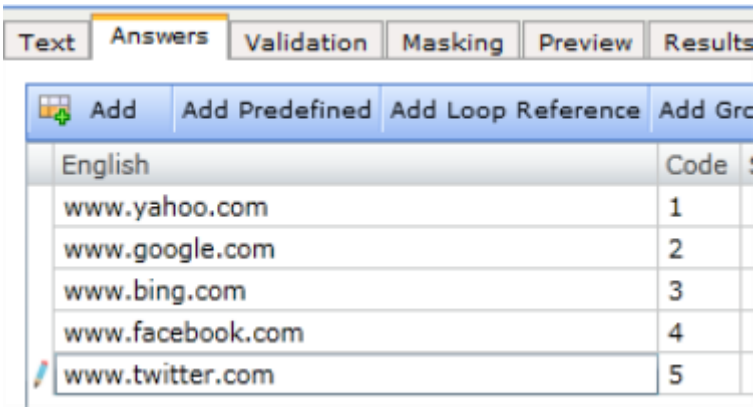
QUERY_STRING	Query information stored in the string following the question mark (?) in the HTTP request.
REMOTE_ADDR	The IP address of the remote host making the request.
REMOTE_HOST	The host name of the client making the request, if available.
REQUEST_METHOD	The type of the HTTP request made: "GET", "POST" or "HEAD".
SERVER_NAME	The server's host name,
SERVER_PORT	The port number to which the request was sent.
SERVER_PORT_SECURE	A string that contains either 0 or 1. If the request is being handled on the secure port, then this will be 1. Otherwise, it will be 0.
SERVER_PROTOCOL	The HTTP protocol and version in use on the server.
SERVER_SOFTWARE	The name and version of the web server software running on the server.

Sending in Values with the URL

Assume 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, as below:

```
http://survey.confirmit.com/wix/<project ID>.aspx?site=1
http://survey.confirmit.com/wix/<project ID>.aspx?site=2
http://survey.confirmit.com/wix/<project ID>.aspx?site=3
http://survey.confirmit.com/wix/<project ID>.aspx?site=4
```

Insert a hidden single question *source* in your questionnaire. The answer list should have the different sites you use and codes that correspond to the values you use in the URLs, for example:



To set this hidden question based on the values sent in with the URL, use the code given below in a script node at the beginning of the questionnaire.

```
f("source").set(Request("site"));
```

Alternatively, you can use the `QueryString` property:

```
f("source").set(Request.QueryString("site"));
```

Important
The script node with Request must be the first node in the questionnaire, because once the respondent moves to the next page, the values are lost.

Now *source* can be used for reporting, quotas, logic etc. in the same way as an ordinary question.

16.2. Response

Response is used to write content to the client (the respondent's browser), including headers and cookies. Here we will only cover the `Write` method and the property used to set cookies.

16.2.1. Write

```
Response.Write(arg)
```

You may use the `Write` method for debugging purposes, for example to get values of variables you are using in your script written to the screen when testing to help identifying why the script does not work as intended. For other purposes it is recommended to use response piping with ^-s inside one of the text fields of a question or an info instead, since you will then have better control of where on the page the text you output is placed.

`arg` can be any type of object, as it will be converted to a string when written to the client.

16.2.2. Cookies

```
Response.Cookies
```

The `Cookies` property indicates the cookies collection, which allows addition of cookies to the outgoing stream, i.e. adding cookies to the client from a Confirmit survey. It can for example be used to prevent respondents of an open survey from taking the survey more than once, as showed in the example below.

The class `HttpCookiesCollection` provides a wrapper for a collection of cookies. You use this to set and modify properties and values in the cookies. Most useful for us is the `Add` method:

```
cookiesCollection.Add(ccookie)
```

The `Add` method allows the addition of a single cookie to the collection. The parameter *`cookie`* is an object of the `HttpCookie` class. `HttpCookie` has two different constructors:

```
varcookie = new HttpCookie(string);
```

which creates and names a new cookie.

```
varcookie = new HttpCookie(string1, string2);
```

which creates, names (*`string1`*), and assigns a value (*`string2`*) to a new cookie.

There is currently a problem with the JScript.NET compiler that gives a type mismatch error when compiling with an `HttpCookie` object. For this reason the variable should be defined as `Object` to avoid this checking:

```
var cookie : Object = new HttpCookie(string);
```

or

```
var cookie : Object = new HttpCookie(string1, string2);
```

Here are the most relevant properties of `HttpCookie`:

```
cookie.Expires
```

The `Expires` property indicates the expiration date and time of the cookie, as a .NET `DateType` value (a bit different than the JScript .NET `Date` object. Consult a .NET reference for more details.) After this expiration date and time, the cookie will not be sent with the request, so you can not retrieve it anymore.

```
cookie.Name
```

`Name` indicates the name of the cookie.

```
cookie.Path
```

Cookies are specific to the site they originate from, so client browsers only send cookies to the DNS domain from which they were created – i.e. a www.microsoft.com cookie can not be picked up by survey.confirmit.com. But you can set a cookie to indicate the directory path on the domain that should receive it using the `Path` property. Using a Path of `"/"` indicates that all directory paths on the server should have access to the cookie.

```
cookie.Value
```

The Value property indicates the value for the cookie.

The majority of cookies are used as a single name and value. However, a cookie can have more than one value. The `Values` property allows you to set several values and also retrieve them again as a `NameValueCollection`:

```
cookie.Values
```

Here is an example on how to set a cookie:

```
var myCookie : Object = new HttpCookie("LastVisit");
var now : DateTime = DateTime.Now; //current time

myCookie.Value = now.ToString(); //convert to string
myCookie.Expires = now.AddMonths(6);
myCookie.Path = "/";
Response.Cookies.Add(myCookie);
```

The value of the cookie can later be fetched like this:

```
var myCookie : Object = Request.Cookies("LastVisit");
if(myCookie != null)
 {
  var last = myCookie.Value;
  //The variable last will now have the timestamp of the last access
 }
```

Using Cookies to Limit Access to an Open Survey

Sometimes the sample for a survey is not known in advance, so that you can not set up the survey as a limited survey with cryptic, individual links. Then you have to distribute the open link to the survey instead. However, you can use cookies to try to prevent the respondents from answering more than once. This is not a bullet-proof method of course, because respondents may have set their browsers to not accepting cookies, they may delete cookies and they may also respond from different PCs,

You can set the cookie from a script node anywhere in the survey depending on how far you think the respondents should have answered before not being allowed to reenter. A cookie using the project id as name can be set with this script:

```
var interval = 3;
var expiry = new Date();
expiry.setMonth(expiry.getMonth()+interval);

var objCookieObject : HttpCookie;
objCookieObject = new HttpCookie(CurrentPID());
objCookieObject.Value = "true";
objCookieObject.Expires = expiry.getVarDate();
objCookieObject.Path = "/" ;
Response.Cookies.Add(objCookieObject);
```

In the beginning of the survey you may then have a condition that uses this expression to check if the cookie is present:

```
Request.Cookies[CurrentPID()] != null
```

If we also wanted to check the value we would also add that to the expression (not really necessary in this scenario):

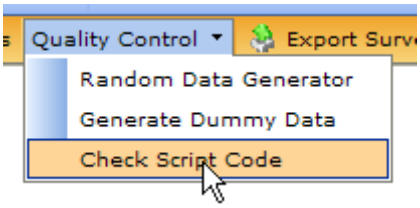
```
Request.Cookies[CurrentPID()] != null &&
Request.Cookies[CurrentPID()].Value == "true"
```

17. Testing Survey Scripts

All scripts should be carefully tested before setting a survey live. Confirmit provides a few helpful tools to use when testing scripts.

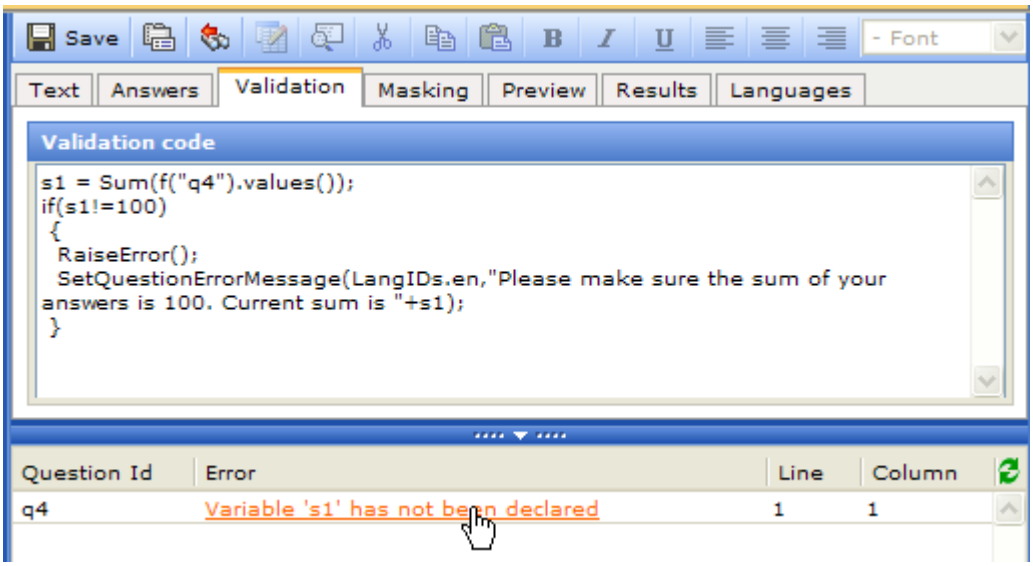
17.1. Check Script Code

You may use the Check Script Code functionality to check for syntactical errors in your script code. This is also done when you launch the survey in test or production mode, but using this functionality gives you a quicker way of checking the script syntax because you do not have to generate the database and interview files.



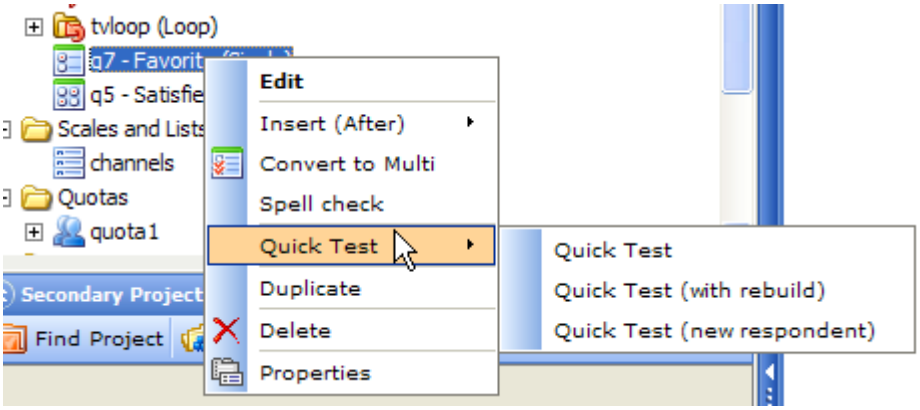
Typical syntax errors that may be found using the script checker are referencing undeclared variables and functions, brackets that do not match - [ ], { } or ( ) - and similar.

The errors found will be listed at the bottom of the screen, and by clicking each error you can open the script node, question or condition containing the error.



17.2. Manual Testing

Confirmit offers two ways of manually testing the surveys. The first of these, Quick Test, gives the ability to quickly open the survey within professional designer at the question you select.



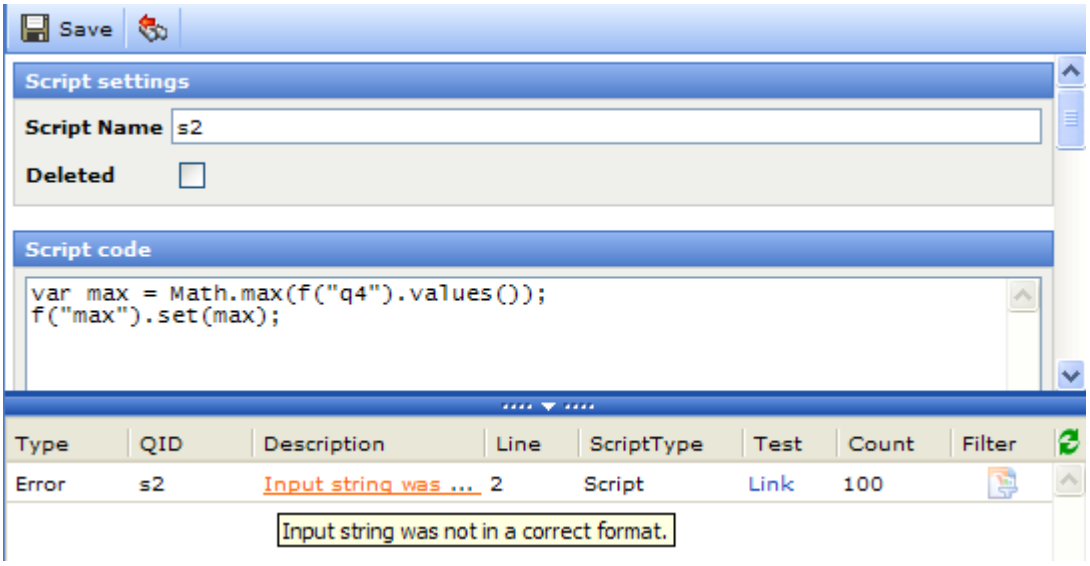
"Quick Test" compiles the survey code and runs it locally on your PC, without writing data back to a database. Data is just kept in the current session. To recompile after making changes, choose the "with rebuild" option. "New respondent" removes the data kept in the session so that you will restart on a blank survey. Quick test is perfect to use when debugging scripts, because the fact that it is not generating a database makes it very quick. This means you can make small changes to a script and test them instantly with Quick Test (with rebuild)..

If you need the test responses stored in a database, you can launch the survey in test mode and run the test interview.

17.3. Random Data Generator

You may have scripts that are syntactically correct, but which will fail run-time. For example, this could be non-existing question ids as parameter in the `f` function or trying to set a string value in a numeric question. Often these errors happen only for certain combinations of answers. To be able to test out as many combinations as possible, you can use the Random Data Generator functionality. (This is a Confirmit add-on. If you do not have access to this functionality, please contact your Confirmit account manager for more information.)

The Random Data Generator lets you run a number of automated test interviews with randomly selected responses. All script code in script nodes, masks, conditions, validation code and response piping will be executed, and any script errors will be recorded and listed for inspection at the end of the RDG run.



The list of errors from the random data generator is clickable, and clicking on the description will open the script, question or info where the error appeared.

In the example above, the problem is that the input to the `Math.max` method is not a list of numbers. In the code above that method returns `NaN` (not a number), which makes the script fail when attempting to set the question *max*, which has the numeric property. A better solution would be to use the Confirmit `Max` function (see Max and Min on page 72 for more information), which automatically converts the values to numbers before finding the maximum.

To help debugging you can use the results tab on the questions to see top line results for each question. It is also possible to filter these results so that you see the results only for those that got a particular error by clicking the filter icon in the rightmost column. You can also click the link in the "Test" column to access the first interview that experienced the error.

17.4. Tips for Debugging

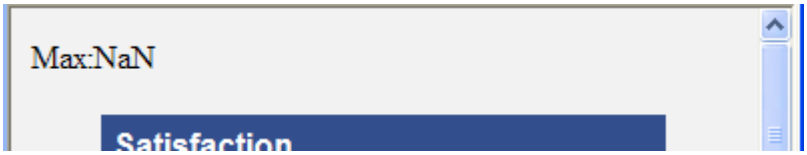
This section contains a few tips to assist you while debugging problem code.

17.4.1. Response.Write

It can sometimes be difficult to find out why a script is failing. In these cases it can help to use the `Response.Write()` method to output some variables to the screen. For example, in the script above which failed when trying to set the **max** question, we could comment out the last line and instead output the `max` variable to screen:

```
var max = Math.max(f("q4").values());
Response.Write("Max="+max);
//f("max").set(max);
```

The text from the `Response.Write` call will appear at the top of the page before any interview HTML. The result from the script above will look something like this:



If an error appears inside of a loop statement, it often helps to use Response.Write to help identify in which iteration the script fails:

```
for(var i : int = 0; i<codes.length; i++)
 {
  Response.Write("i = "+i+"<br>");
 < ... >
 }
```

Other tricks to simplify debugging is to use `/*` and `*/` to comment out part of the code to help identify exactly where the error occurs. This can be done as a "binary search": First you comment out ½ of the code in a script. If the error still occurs, you know that it is in the other half. Then you can comment out half of that section – and so on.

It can also be very helpful to use Top line reporting or the "results" tab on a question, the "Edit" functionality under "Survey Data" or doing a data export of the responses with "error" status to look into what answers actually are stored in the question(s) involved in the script code.

17.4.2. Debug Station

Confirmit Debug Station is a downloadable application provided as part of the Confirmit license, to be used for debugging script code in Confirmit surveys.

Important
Confirmit Debug Station is a specialist feature that is intended to be used only by programmers who write advanced scripts as part of their surveys and who need to debug the script code. You must accept the terms of use as stated on the download page in Confirmit before you can download the application.

The Debug Station is not supported on users' pcs. All support required for this feature is chargeable.

Confirmit Debug Station must be installed on the user's PC along with a third-party .NET debugging application such as Visual Studio 2005. Confirmit Debug Station can then be used to run Confirmit surveys locally on the user's PC, attached to the .NET debugger, to debug script code in the survey (script nodes, code masks, question masks, conditions, validation code and piping).

Launch the survey for debugging, synchronize it to the debug station, and then attach the .NET debugger to the debug station. Set breakpoints in the usercode, then when the survey starts, the debugger will pause the survey when it comes to a break-point in the code.

How to Install the Debug Station

1. Open a project and go to the **Designer > Quality Control > Debug Survey** menu command.

   A page of information including several links opens.

2. Click the link in the text to download the Debug Station, then follow the instructions displayed.

Note: If you already have CAPI installed then you do not need to install Debug Station. Merely replace the activation files.

How to Run the Debug Station

1. Log on to Confirmit and select a project from the project list.
2. Go to the **Professional Designer > Quality Control > Debug Survey** menu command.

   A page of information including several links opens.

3. Click the **Debug Survey** button to launch the survey for debugging.

   The Task pane opens and the launch task is run. This launch will not require or update the survey database.

4. Start Confirmit Debug Station. Start the debugging software and attach it to the Confirmit Debug Station process.
5. In the debugging software, open the **GeneratorCode.js** file (containing the script code) for the survey you wish to debug.

   The file is located in **C:\Program Files\confirmit\data\surveys\wi\pxxxxxxx**. You can now set break points in this file.

6. Select the project in Confirmit Debug Station and start the interview. The debugger will pause the survey when it comes to a breakpoint in the code.

Note: Any interview data collected while the console is in Debug mode will not be synchronized back to the server.

18. Programming Conventions

Programming conventions are important to programmers because they make scripts easier to read and understand. Writing scripts that are easy to understand will make searching for errors easier, and will make it easier to re-use scripts. If programmers agree on a common way of writing scripts, it will be much easier to read the scripts of other programmers. Even the programmer that originally wrote a script may have problems understanding his or her own code after a few months unless it is written and commented in a way that makes it easy to read.

18.1. Comments

It is recommended to make 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 Comments on page 13 for more information).

18.2. Naming Conventions

When naming variables and functions, always try to use as descriptive names as possible. If an array holds the codes from a question, call it something like codes or q1Codes or similar, not just e.g. a. Similarly, a function that copies a multi question should be called something like CopyMulti, not f1. There is no point in making short names in JScript .NET.

A variable name must start with a lower or upper case letter or underscore, and continue with letters, digits or underscore. However, it is recommended to follow the following naming conventions:

- variable names and names of methods and properties should start with a lowercase letter in the first word. The next words should open with an uppercase letter. Examples: gridCodes, availableCodes, randomNumber, indexOuterLoop.
- function names and names of objects should start with an uppercase in the first word, and have uppercase letters in the beginning of all subsequent words. Examples: CopyMulti, SelectItems, CalculateAverage.

18.3. Spaces and Line Breaks

Except from within string expressions, you can use line breaks and spaces as much as you like when writing JScript .NET code. Use these to make your code easier to read. Add spaces 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 which statements belong to the various parts.

18.4. Curly Brackets

It is recommended that you always use the curly brackets ({ and }) in the if, switch, while, do while, for and function statements. If you have just one statement inside the curly brackets, they are optional, but it is recommended that you make it a rule to always use them, limiting the possibility of making mistakes.

Example:

```
if(condition)
  <statement1>;
```

is the same as

```
if(condition)
 {
  <statement1>;
 }
```

so you may find it convenient to drop the curly brackets. But you may soon need to add another statement. If you add it like this:

```
if(condition)
  <statement1>;
 <statement2>;
```

or

```
if(condition)
  <statement1>; <statement2>;
```

it will be equivalent to

```
 if(condition)
 {
  <statement1>;
}
<statement2>;
```

This will be very different from

```
if(condition)
 {
   <statement1>;
   <statement2>;
}
```

18.5. Semi Colon

Statements are separated with semi colon (;) in JScript .NET. However, if you have line breaks between the statements, the semi colon is not required. However, it is recommended to always use it anyway, in case a line break is removed. You may have observed that all examples in this documentation use semi colons between the statements.

However, when working with the if, switch, while, do while, for and function statements you have to be careful with the semi colon. For example, remember that the while statement includes the statements within the curly brackets. If you place a semicolon just after the condition, like this:

```
while (condition); {  <statements> }
```

you will actually end up with a loop that never terminates. The semicolon will be interpreted as the end of an empty statement. It will be this empty statement that will be executed until the condition is false, not the statements within the curly brackets. The code that executes will be similar to this:

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

The condition is likely never to return false, since no statements are executed. This means that the loop will never terminate. This will cause a time-out for the respondent. But notice that there are no syntactical errors in the script, so you will not receive any error message on the script as such.

18.6. The Step by Step Approach

It is possible to do a lot of operations in one long statement. However, this makes the scripts hard to read.

Compare for example the following code, which randomly picks one of the codes in an array codes,

```
selectedCode = codes[Math.floor(Math.random()*codes.length));
```

with this code:

```
randomNumber = Math.random()*codes.length;
randomIndex = Math.floor(randomNumber);
selectedCode = codes[randomIndex];
```

The first code will be a bit more efficient than the latter, because you do not have to store values in the variables randomNumber and randomIndex. However, because it will be much easier to understand what happens in the last one, that approach is recommended.

To increase readability, use temporary variables and do the calculations step by step.

18.7. Writing Efficient Code

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

- Define types on your variables, constants and functions when possible.
- Always make sure that the script does not run through unnecessary iterations in a loop, or unnecessary statements in an iteration. Use `break` or `continue` to terminate or skip to the next iteration.
- If you need to call the `f` function for the same question more than once in a script, store it in a variable and refer to that variable instead.
- Avoid using calls to user-defined functions in code masks. Set a hidden multi question instead and refer to it with the `f` function.
- Use methods of the form objects like `domainValues` and `categories` when possible instead of hard coding codes(1,2,3,…) in your scripts.
- Use short-circuiting evaluations in expressions involving logical and (`&&`) and or (`||`): Place conditions most likely to be `true` first for the logical or (`||`) operator, and conditions most likely to be `false` first for the logical and operator (`&&`).
- Only use local variables in functions (use the `var` keyword).

APPENDIX A: Answers to Exercises

Exercise 1:

```
"207 is not the same as 207, but this isn't really true"
```

Exercise 2:

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:

1. Code sample 1: x=5 and y=5
   Code sample 2: x=5 and y=6
2. Code sample 1: x=5 and y=5
   Code sample 2: x=5 and y=5

Exercise 4:

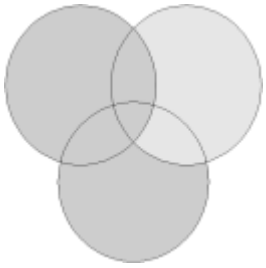
1. f("importance")["2"].valueLabel()
2. f("importance")["2"].label()

Exercise 5:

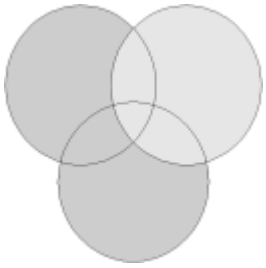
```
if(!f("q2").toBoolean())
 {
  var codes = f("q2").domainValues();
  for(var i : int = 0;i<codes.length;i++)
   {
    var code = codes[i];
    f("q2")[code].set("3");
   }
 }
```

Exercise 6:

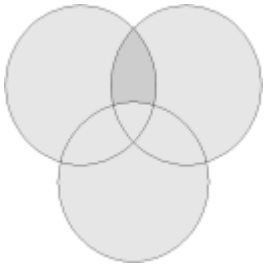
1,2,5,6,7,9,10,11,12



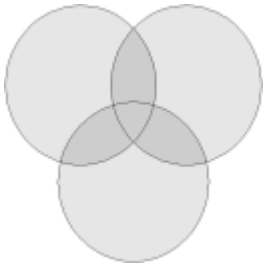
1,2,6,7,9,10,12



11



2,4,5,10,11



Exercise 7:

```
var d = IsDateFmt(f("date4").get(),"MM/DD YYYY");

if(!d)
 {
  RaiseError();
  SetQuestionErrorMessage(LangIDs.en,"Invalid date. Please correct
using the format MM/DD YYYY");
 }
else
 {
  var dt = new Date();
  dt.setFullYear(d.year,d.month-1,d.day);
  var current = new Date();
  if(dt.valueOf() < current.valueOf())
   {
    RaiseError();
    SetQuestionErrorMessage(LangIDs.en,"Please enter a date after the
current date.");
   }
  else if(dt.getDay() != 1)
   {    RaiseError();
    SetQuestionErrorMessage(LangIDs.en,f("date4")+" is not a Monday.
Please register for Mondays only.");
    }
 }
```

Exercise 8:

```
var num : String = f("phone").get();
var re = /^\d{3} \d{3} \d{4}$/;
if(!re.test(num))
 {
  RaiseError();
  SetQuestionErrorMessage(LangIDs.en,"Please provide your phone number
on the format xxx xxx xxxx");
 }
```

Exercise 9:

```
var zipcode : String = f("zipcode").get();
var re = /^\d{5}$/;
if(zipcode.search(re) == -1)
 {
  RaiseError();
  SetQuestionErrorMessage(LangIDs.en,"Please provide a valid zip code
(digits only).");
 }
```

APPENDIX B: Further Reading

For further reading on JScript .NET, we recommend:

MSDN (Microsoft):http://msdn.microsoft.com/en-us/library/72bd815a(v=vs.71)

APPENDIX C: Confirmit Language Codes

Important
Custom languages that are added after the initial release of Confirmit are assigned a Language code number automatically, based on the order in which the languages are added. Custom languages may therefore have different language IDs for each installation of Confirmit (On-Premise, On-Demand Euro, On-Demand US). Therefore, be aware that if you intend to export/import surveys between the different environments (On-Premise/Euro/US/) and you are using custom languages, you should check the relevant language codes for the sites and edit the XML export files as appropriate.

For each installation; go to the Home > Help > Language Overview menu command to view a list of the languages, with their applicable codes, available on that server.

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

APPENDIX E: List of Examples

Example

40. Update Panel Variables in a Panel
41. Retrieving Data for a Panelist in a Panel
42. Update Survey History Panel Variables in a Panel
43. Redirect back to Panel Portal
44. Checking that a Response in a Multi Open Text Question is an Integer
45. Validating Date Format of Open Text Date Question
46. Validating Date with Drop-downs for the Date Parts
47. Validation of Email Address Format
48. Excluding Respondents from Specific Networks
49. Send Confirmation Email at the End of a Survey
50. Invitation Email to a Different Part of the Same Survey
51. Sending a PDF to the respondent with answers from the survey
52. Redirect to Another Site Before the End Page
53. Code Masking Based on a Variable Number of Conditional Expressions
54. Returning a Calculated Value from a Function
55. Using a Variable instead of Repeated Calls on the f Function
56. Deleting the Content of any Question
57. Copying the Contents of any Form into Another
58. Filtering an Answer List by the First Characters in the Answer
59. Asking for Favorite Only when More than 1 Item is Chosen
60. Filtering an Answer List on Items Selected in Two Previous Questions
61. Filtering Answers Not Selected in a Previous Question
62. Always Including a "Don't know" Answer Alternative
63. Validating "Other, specify" in a 3D Grid
64. Using a Function to Filter an Answer List Based on the Answers on a Grid
65. Using a Hidden Multi to Filter an Answer List Based on a Grid
66. Calculating Time Spent
67. Validating that a Date with Drop-downs is within the next two Weeks
68. Finding the Weekday
69. Converting Data of Birth into Age (in number of years)
70. Rounding to a Number with Two Digits
71. Picking n Random Items from the Answers to a Multi Question
72. Randomly Assigning which Part of a Survey the Respondents Should Answer
73. Using modulus to route respondents to different parts of the questionnaire
74. Checking a User Name and Password where Username is Case Insensitive
75. On the Fly Recoding
76. Replacing Last Comma with "and" in a Listing of Answers
77. Generating an Array from a String with Values
78. Validation of the Format of a Phone Number
79. Using Regular Expression to Replace Commas With Line Breaks
80. Post Codes in the United Kingdom
81. Converting an Array to a String with Line Breaks between Elements

82. Validating Grid with "Other, specify" Alternatives
83. Sending in Values with the URL
84. Using Cookies to Limit Access to an Open Survey

APPENDIX F: QSL-to-Confirmit Script Conversions

This table provides examples of conversions from the QSL-script used in Pulse Train projects to the script used in Confirmit projects.

Section	QSL	Confirmit	Confirmit more info.	Description
Routing				
If	IF var_single(1); ENDIF;	f('var_single')=='1'	Use in Condition Node	Check if answer category 1 of single var is switched on
	IF var_single(1 | 2); ENDIF;	f('var_single')=='1' || f('var_single')=='2' or set('1','2').inc(f('var_single'))	Consider nset() and nnset() when using numeric codes	Check if answers 1 OR 2 are switched on
	IF ^var_single(1); ENDIF;	!(f('var_single')=='1') or f('var_single')!='1'	"	Check if single var does NOT have code 1 switched on
	IF var_multiple(1); ENDIF;	f('var_multiple').inc('1')	"	Check if answer 1 switched on in multiple var
	IF var_multiple(1 & 2); ENDIF;	f('var_multiple').inc('1') && f('var_multiple').inc('2') or set('1','2').isect(set('1','2')).size() == 2	Consider nset() and nnset() when using numeric codes	Check if answers 1 AND 2 are switched on in multiple variable

	IF var_single(1) |
	var_multiple(1);
	ENDIF;
		f('var_single')=='1' ||
		f('var_multiple').inc('1')
			"	Check if answer 1 is switched on in EITHER single var OR multiple var

	IF var_single(1) &
	var_multiple(1);
	ENDIF;
		f('var_single')=='1' &&
		f('var_multiple').inc('1')
			"	Check if answer 1 is switched on in BOTH single var and multiple var

	If var_multiple(1 || 5);	InRange('var_multiple'1,5)	Consider nset()	Checks if ANY

Section	QSL	Confirmit	Confirmit more info.	Description
	ENDIF;	or set('1','2','3','4','5').isect(f('var_multiple')).size() > 0	and nnset() when using numeric codes	answers between 1 and 5 are selected (ie 1 or 2 or 3 or 4 or 5)
	If var_multiple(1 && 5); ENDIF	f('var_multiple').inc('1')&& f('var_multiple').inc('2') && f('var_multiple').inc('3') && f('var_multiple').inc('4') && f('var_multiple or set('1','2','3','4','5').isect(f('var_multiple')).size() == 5	Consider nset() and nnset() when using numeric codes	Checks if ALL answers between 1 and 5 are selected (ie 1 and 2 and 3 and 4 and 5)
	IF var_quantity==1; ENDIF;	f('var_quantity').toNumber()==1	"	Check if quantity variable is equal to 1
	IF var_quantity>>1; ENDIF;	f('var_quantity').toNumber() > '1'	Remember .toNumber(), to ensure numeric evaluation	Check if quantity var is greater than 1
	IF var_quantity>=1; ENDIF;	f('var_quantity').toNumber() >= '1'	''	Check if quantity var is greater than or equal to 1
	IF var_quantity<<1; ENDIF;	f('var_quantity').toNumber() < '1'	''	Check if quantity var is less than 1
	IF var_char=='abcde'; ENDIF;	f('var_char')=='abcde'	See also String.match / String.replace / String.search in Scripting manual	Check if character variable is equal to 'abcde'
Run Mode (Web)	IF RUN_MODE()==R$4(WEB); ENDIF;	GetSurveyChannel == 'Cawi'	Use in Condition node	Checks if Run Mode is Web
Run Mode (Cati)	IF RUN_MODE()==R$4(CATI); ENDIF;	GetSurveyChannel == 'Cati'	"	Checks if Run Mode is Cati
Run Mode (Capi)	IF RUN_MODE()==R$4(CAPI); ENDIF;	GetSurveyChannel == 'Capi'	"	Checks if Run Mode is Capi
Case	CASE VAR_SINGLE	Switch(f('var_single').get()	See Switch	Routing using

Section	QSL	Confirmit	Confirmit more info.	Description
	WHEN (1);	{ Case ‘1’ : <code 1 actions>	statement in Scripting manual	Case statement; (an alternative to “if single_var(1) then.....else if single(var(2) then.....else if single_var(3) then .....else if single_var(4) then.....else.....”)
	COMMENT Actions for when var_single is code 1;	break		
	WHEN (2);	Case ‘2’ : <code 2 actions>		
	COMMENT Actions for when var_single is code 2;	break		
	WHEN (3);	Case ‘3’ : <code 3 actions>		
	WHEN (4);	Break		
	WHEN OTHERS;	Case ‘4’ : <code 4 actions>		
	COMMENT Actions for if not caught by above cases;	break		
	ENDCASE;	default: <actions for any other case>		
		}		
While	WHILE (<condition>);	While (<condition>)	See While in Scripting manual – also see ‘Do While’ and ‘For’ in Scripting manual	While loop – stay in while loop while condition evaluates to True
	ENDWHILE;	{ <statements> }		
Redo	REDO <var_name>;	Redirect(GetRespondentUrl()+'&__qid=xx',true)	Where xx is the qid you want to start re-doing. One should also use script to 'end' the redo. Or, consider using Call Blocks.	Ask a question again
Operators				
Arithmetic	+	+	Scripting – Operators	Addition
	-	-	Scripting – Operators	Minus
	/	/	Scripting – Operators	Division
	*	*	Scripting – Operators	Multiplication

Section	QSL	Confirmit	Confirmit more info.	Description
Relational	==	== or === (depending on with/without type conversion)	Scripting – Operators	Equals
	>>	>	Scripting – Operators	Greater than
	>=	>=	Scripting – Operators	Greater than or equals
	=<	<=	Scripting – Operators	Less than or equals
	<<	<	Scripting – Operators	Less than
	<>	!= or !== (depending on with/without type conversion)	Scripting – Operators	Not equals
Logical	&	&&	Scripting – Operators	And operator
	|	||	Scripting – Operators	Or operator
	^	!	Scripting – Operators	Not operator
Calculations				
Temp	TEMPORARY <temp_var>;	vartemp_var : int; OR vartemp_var : int = value (if assigning as well)	Defined in script nodes, and validation code.	Creates a temporary quantity variable
Increment	INCREMENT <var_name>;	++ <var_name>	See Arithmetic Operators in Scripting manual (also useful is -- to decrement)	Increments a variable (either quantity or temp variable)
Zero	ZERO <temp_var_name>;	variablename = 0;	Assuming variable has been previously defined in the script. If not, use vartemp_var_name : int = 0	Sets to zero a temporary variable
Compute	COMPUTE <target_var> (<var1>+<var2>);	f('target_var').set(f('var1').toNumber() + f('var2').toNumber()) or f('target_var').set(Sum(f('var1').toNumber(),f('var2').toNumber())	Using Sum() is a better option, when dealing with several previously defined variables, or the values from a multi question:	Set value in target_var to be sum of var1 + var2

Section	QSL	Confirmit	Confirmit more info.	Description
			varmySum = Sum(f('var_multiple').values()) will sum up any number of answers, given in var_multiple.	
Off	OFF var_multiple 1;	f('var_multiple')[1].set(null)		Switches OFF answer category 1 in var_multiple
On	ON var_single 2;	f('var_single').set('2')		Switches ON answer category 2 in var_single
Clear (single)	CLEAR var_single;	f('var_single').set(null)	Set	Clears any value in single variable

Clear (multiple)	CLEAR var_multiple;	f('var_multiple')[<code>].set(null)
		This will need to be iterated for each code. Or, to clear all codes:
		function ClearMulti(MVar)
		{
		var codes = f(MVar).domainValues();
		for(var i : int = 0; i<codes.length ; i++)
		{
		var code = codes[i];
		f(MVar)[code].set(null);
		}
		}
		ClearMulti('var_multiple')
			Set	Clears any value in multiple variable

Clear (quantity)	CLEAR var_quantity;	f('var_quantity').set(null)	Set	Clears any value in quantity variable
Clear (char)	CLEAR var_char;	f('var_char').set(null)	Set	Clears any value in character variable
Masking				
	<var1> | <var2>	f('var1').union(f('var2'))	Union	Include answers that are switched on in EITHER

Section	QSL	Confirmit	Confirmit more info.	Description
				var1 or var2
	<var1> & <var2>	f('var1').isect(f('var2'))	Isect	Includes answers that are switched on in BOTH var1 and var2
	var1 | var2 + TRUE(1)	f('var1').union(f('var2')).union(set('99'))	Union	Include answers that are switched on in EITHER var1 or var2, also always include last answer (eg Don't Know or Refused)
	var1 & ^(var2)	f('var1').diff(f('var2'))	Diff	Include answers switched on in var1 but exclude answers switched on in var2
Quotas				
Define quota	<quota_name>: LOOKUP_B <var_single> (150 150);	These are defined in the Questionnaire Tree. Refer to the Authoring documentation for further information.		Defines quota <quota_name> where target quotas are 150 in answer1 and 150 in answer2 from <var_single>
Check quotas	IF ^<quota_name>; ABANDONED QUOTAFAIL ; ENDIF;	qf('quotaname')	Use in Condition Node	Checks if target in quota cell has been reached; if it has then interview is abandoned and status set to quotafail
Assign quotas to variable	Compute <quantity_var> Quota_Value(<quota_name>,1)	f('quantity_var').set(qc('quota_name'))	sets current count into question 'quantity_var'.	Computes to <quantity_var> the current number of Achieved interviews in cell 1 of <quota_name>

Section	QSL	Confirmit	Confirmit more info.	Description
	Compute <quantity_var> Lookup_Value(<quota_name>,1)	f('quantity_var').set(qt('quota_name'))	sets quota target into question 'quantity_var'. However, we could just use: var x = qt('quota_name'), if working in a script node.	Computes to <quantity_var> the current number of Target interviews in cell 1 of <quota_name>
Functions				
Count	count(var_multiple)>>1	f('var_multiple').size()	Size	Count function returns number of answers set to True in variable
Random	Random(5)	Math.ceil(Math.random()*5)		Returns a random number between 1 and 5

Range	Range(var_quantity,0,18,66)	var no = f('var_quantity').toNumber();
		var single = f('single_var');
		if(InRange(no,0,17))
		single.set('1');
		else if(InRange(no,18,65))
		single.set('2');
		else if(InRange(no,66,99))
		single.set('3');
				Computes quantity variable into single; in this example var_quantity is a quantity var with range of 0-99 (eg age question), target var is a single with answer categories:
				Under 18
				18 to 65 inc
				66 or more
				Appropriate answer category will be switched on in target depending on value of age_quantity

Remainder	Remainder(dividend,divisor)	Dividend%divisor	Modulus	Returns remainder from dividend divided by divisor; eg if value of

Section	QSL	Confirmit	Confirmit more info.	Description
				var_name is 11 then Remainder(var_name,3) would return 2
Unrecorded	Unrecorded(<var_name>)	(!f('var_name').toBoolean())	Boolean	Returns True if <var_name> is unrecorded
Not Unrecorded	^Unrecorded(<var_name>)	f('var_name').toBoolean()	Boolean	Returns True if <var_name> has a value
Date functions				
Day	Day_of_Month()	vardate_var = new Date().getDate()		Function that returns day of current date (eg 15th Jan would return 15)
Month	Month()	varmonth_var = new Date().getMonth()+1;		Returns month of current date (eg if June, would return 6)
Year	Year()	varyear_var = new Date().getFullYear();		Returns year of current date
Misc.				
Comments	COMMENT this is a comment;	// this is a comment /* this is a multiline comment */		Adding a comment to a script
DTS (answer to variable)	<%var_name>	^f('var_name')^	Piping	Substitutes answer to var_name (if var_name is a categorical variable then it substitutes answer category text)
DTS (conditional text)	<(conditional_expression)/TrueText/FalseText>	^condition ? TrueText :FalseText^	Conditional Expression Ternary Operator	If condition evaluates to True then interview engine inserts TrueText,

Section	QSL	Confirmit	Confirmit more info.	Description
				otherwise it inserts FalseText
DTS (conditional answer to a question)	<(conditional_expression)%var_name>	^condition ? f('var_name') : ""^	Conditional Expression Ternary Operator	If condition evaluates to True then insert answer to var_name
DTS (date)	%D	^new Date()^	Inserts full datetime value	Inserts date
DTS (tel number)	%T	GetTelephoneNumber()		Inserts telephone number (only appropriate for CATI)
Call Count	CALL_COUNT()	GetCallAttemptCount()		Returns the number of telephone calls that have been placed to this record on previous occasions

Index

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

Y