BIRT Report Object Model – Expressions and Scripting

Functional Specification

Draft 7: May 26, 2005

Abstract

Describes the expression and scripting support available to report designs using the Report Object Model.

Document Revisions

Version	Date	Description of Changes
Draft 1	11/29/2004	First BIRT release.
Draft 2	2/4/2004	Updated data-related classes; some minor revisions of others
Draft 3	2/11/2004	Incorporated 1st batch of review comments on data-related classes
Draft 4	2/13/2004	Expanded section on "Scripted Control of Report Items"; Structural changes to make the document more readable; Changed Term "Object" to "Class" when it refers to a type; Miscellaneous fixes. Clarified Script execution scope rule in Section 3.1.
Draft 5	4/20/2005	Clarified field access syntax to use me, instead of this Clarified supported features in R1
Draft 6	5/24/2005	Updated DataSet and DataSource classes to reflect new oda extension design. Removed some unimplemented classes.
Draft 7	5/26/2005	Updated ElementDefn, DataRow class.

Contents

1. Introduction

BIRT provides outstanding scripting support based on the JavaScript (formally "ECMA Script") language. This specification explains the requirements, then the detailed user-level features.

1.1 About this Document

This is a solid draft version of the scripting specification. The major features are identified, and syntax for many of the claaaes is becoming stable.

1.2 Limitations

This specification describes the overall scripting design. This section identifies limitations in the first release due to schedule constraints.

- No two-pass aggregates.

- No decimal data type. Money amounts will be represented as double-precision floating-point numbers.

- No operator support for Dates or Decimal types. Only built-in JavaScript operators are supported.

2. Scripting Overview

Many reports require only a simple data set and standard formatting. However, most business applications are very complex, and custom code is often needed to adapt data for use within a report. Reports often present complex business rules, also expressed in code. Report formatting must sometimes adjust based on data or business rules.

For these reasons and more, BIRT provides a powerful scripting feature. BIRT scripting is based on the Mozilla Rhino implementation of JavaScript (ECMAScript.) Rhino provides excellent integration with Java classes, allowing report scripts to work seamlessly with business logic written in Java.

Report developers are often not Java developers. Instead, they often have web or database experience. JavaScript is an excellent, easy-to-use scripting language accessible to anyone with at least some programming experience. Many excellent books are available, including several tutorials, to help developers get started.

2.1 JavaScript for BIRT

The following description of JavaScript comes from JavaScript, the Definitive Guide by Flanagan:[1]

> JavaScript is a lightweight, interpreted programming language with object-oriented capabilities… Syntactically, the JavaScript language resembles C, C++ and Java… The similarity ends with this syntactic resemblance, however. JavaScript is an untyped language, which means that variables do not need to have a type specified.

---

[1] JavaScript: The Definitive Guide, 4th Edition by David Flanagan, O'Rielly Associates, 2002.

Objects in JavaScript are more like Perl's associative arrays than they are like the structures in C or objects in C++ or Java…

JavaScript appears at first glance to be a fairly simple language, perhaps of the same complexity as BASIC. JavaScript does have a number of features designed to make it more forgiving and easier to use for new and unsophisticated programmers… Beneath this veneer of simplicity, however, JavaScript is a full-featured programming language...

BIRT scripting is based on the Rhino JavaScript engine from mozilla.org. Rhino implements EcmaScript version 1.5 as described in the ECMA standard ECMA-262 version 3: http://www.ecma-international.org/publications/standards/Ecma-262.htm. The first release of BIRT will use Rhino version 1.5R3.

Highlights of JavaScript features include:

- Simple expressions: price * quan

- Scripted expressions: if (custType = "C") { price * quan } else { price * quan * discount }

- If/then/else, looping and other program constructs

- Global functions and variables

- Custom objects

- Arrays, floats, strings, dates

- Wide range of available books and web sites.

An excellent JavaScript reference is JavaScript, The Definitive Guide. This specification will not repeat the valuable background information presented there.

2.2   Rhino Implementation of JavaScript (ECMAScript)

Web site: http://www.mozilla.org/js/

From the web site:

JavaScript is the Netscape-developed object scripting language used in millions of web pages and server applications worldwide. Netscape's JavaScript is a superset of the ECMA-262 Edition 3 (ECMAScript) standard scripting language, with only mild differences from the published standard.

JavaScript can function as both a procedural and an object oriented language. Objects are created programmatically in JavaScript, by attaching methods and properties to otherwise empty objects at run time, as opposed to the syntactic class definitions common in compiled languages like C++ and Java. Once an object has been constructed it can be used as a blueprint (or prototype) for creating similar objects.

JavaScript's dynamic capabilities include runtime object construction, variable parameter lists, function variables, dynamic script creation (via eval), object introspection (via for ... in), and source code recovery (JavaScript programs can decompile function bodies back into their source text)

Intrinsic objects are Number, String, Boolean, Date, RegExp, and Math.

The Rhino engine, created primarily by Norris Boyd (Netscape) is a JavaScript implementation in Java. Rhino is ECMA-262 Edition 3 compliant.

Rhino allows direct access to Java objects using "packages." Rhino can

…reach beyond JavaScript into Java. Scripting Java has many uses. It allows us to write powerful scripts quickly by making use of the many Java libraries available... scripting makes this process easier.

Note that the ECMA standard doesn't cover communication with Java (or with any external object system for that matter). All the [Java extension] functionality… should thus be considered an extension.

2.3 BIRT Scripting vs. Browser Scripting

BIRT reports allow both report scripting (executed in the BIRT Factory and Presentation engines) and browser-scripting (executed in the user's web browser.) These are traditionally referred to as server-side and client-side scripting respectively. JavaScript (as defined by the browser vendor) is the standard client-side scripting language. We have also selected JavaScript as the server-side scripting language.

The reader (and user) must differentiate between these two distinct uses of JavaScript. If one wants to compute a value for a data item then that is server-side scripting. If one wants to pop up a dialog when the user clicks on a report item then that is client-side scripting. Both use JavaScript, but the two scripts are not interchangeable. That is, a client-side script cannot refer to server-side constructs such as the report design. And, a server-side script cannot refer to client-side constructs such as the DOM tree.

This specification focuses on server-side scripting in BIRT. Client-side scripting support is described elsewhere in the ROM specs.

2.4 Expressions

Report developers use expressions in a number of places. Data items display the result of an expression. Expressions choose the formatting rules to apply when adding highlighting to a report. Expressions form the basis of a computed column. Expressions are simply scripts that return a data value. Expressions can reference a wide variety of variables and objects including:

- Data set columns

- Parameter values

- Configuration variables

- Constants

- User-defined variables

Expressions are executed as methods on a JavaScript object that represents the report element.

2.5 Methods

Many report elements provide methods that BIRT calls at various points. They are also viewed as event handlers, because they are called in response to specific events during report execution and presentation. Methods allow the report developer to customize the behavior of a report outside of expressions. For example, methods on the scripted data set allow the developer to open, read, and close a custom data source. Methods have access to all the same variables and functions as expressions.

2.6   Standard Functions

BIRT and JavaScript offer a wide range of functions including:

- String functions

- Date/time functions

- Math functions

- Conversion functions

- Financial functions

- Statistical functions

- Aggregate functions

Expressions can also reference functions defined in Java code, or in the BIRT scripting language (i.e. JavaScript).

3.   BIRT Script Contexts

The report developer must understand the context in which a given script executes. A report has a lifecycle defined by data access, report content creation (Factory) and presentation. Some scripts execute in only one phase of the life cycle, others execute in two or all three phases. Unexpected results can occur if the developer writes code that is not appropriate to the phase in which it executes. This section explains the phases and the resources available in each phase.

3.1   The Scripting Environment

JavaScript is designed to host a series of independent "code snippets" that share a common environment. JavaScript evolved out of the browser environment in which bits of script affected the visual appearance of various parts of an HTML document. In BIRT, the code snippets appear in various places:

- Source files included into the report design.

- Global functions defined in the `initialize( )` method of the report design.

- Methods such a "when the report starts" (i.e., `onStart`) or "when each row is fetched" (i.e., `onRow`.)

- Expressions, such as the computation of the value of a computed column or of a data item.

BIRT introduces two kinds of JavaScript scopes for script execution: one top-level "global" scope and multiple second-level scopes. The global scope serves as the parent scope for all second-level scopes. Global variables and functions, including non-local variables and functions defined in any script, wherever they may appear, belong to the global scope and can be accessed in all other scripts that are subsequently executed. Scripts defined on each report element, including element expressions, are executed within a second-level scope for the report element. In each case, the developer writes the code required. That code can make use of the following scripting environment:

- A set of functions and objects defined by JavaScript itself. These include a string class, math operations, array functionality and so on. These are defined in the ECMA Script specification, and are implemented by Rhino.

- A set of objects defined by BIRT such as the data row, a description of the report, the report design, report parameters and so on. These are defined in the sections that follow.

- A set of global variables, functions and objects defined by the report itself, or by libraries included in the report.

- Java objects and classes imported into JavaScript.

Each of these is explained in greater detail below.

3.2 Report Phases

A script context may execute in any one of five *report phases*. Each describes a phase in the report lifecycle. The phases presented here are descriptive; there may sometimes be a fuzzy line between them.

The five phases are:

- Startup

- Data transform

- Factory

- Presentation

- Shutdown

Each phase defines a slightly different set of available report resources. Some report resources are available in all contexts:

- The report design

- Report parameters

- Configuration variables

- Global functions and variables defined during startup

Other resources are available in some, but not all, phases:

- Elements that describe the report document such as actual lists, report items, etc.

- Data sources and data sets

- Data rows

- Information about the report document, server request, presentation message, etc.

3.2.1 Startup Phase

Startup occurs when the Factory is preparing to run a report, but before the report itself begins executing. Startup phase is when the report sets up parameters, determines output file names, locates configuration variables, and so on. BIRT calls specialized

event handlers in the report design that let the developer define functions or objects for use later within the report.

3.2.2 Data Transform Phase

Data Transform phase includes all the operations that occur when preparing a result set for use in the Factory.

Suppose a report requests a sort on column "x", and this column is defined using an expression: "x = y + z". In order to sort the data, BIRT must first read all the rows, computing column x for each. BIRT can then sort the rows and build the report. The action of reading the rows, computing the computed columns and sorting is called a "pre-processing pass" because it is done prior to the pass over the data that builds the report.

Some reports may require multiple pre-processing passes. For example, a report may have to sort rows to create groups, apply a top-n filter to the groups, and then sort the results by another column.

3.2.3 Factory Phase

Factory phase is when BIRT uses the result set from a data set to build the report content. This is when BIRT detects level breaks, creates sections, determines the default pagination, and so on. Scripts that execute at Factory time generally relate to making decisions about whether to create a given element, about customizing a data set and so on.

3.2.4 Presentation Phase

Once a report is run, the user can view or print it. Such actions occur during presentation phase. At this time, the data sets have been read, report content created and the results written to the report document file. Scripts that run at presentation time deal with preparing an element for presentation such as choosing a color, determining locale-specific behavior and so on.

Presentation phase will often occur in a different process (and perhaps on a different computer) than factory phase. The report may be run, written to disk, and later reopened for presentation. Hence, scripts that may execute at presentation time should not make use of in-memory state (such as variables) defined only within the Factory. For example, suppose a report creates a Factory event that counts the number of time the event is called. That count may or may not be available at presentation time. Scripts that depend on a particular behavior will be troublesome.

In the first release of BIRT, report generation and presentation happen in the same process and report documents are not created on disk. The factory and presentation phases are intertwined, creating some restrictions on scripting. Such restrictions are explained in detail later.

3.2.5 Shutdown Phase

Once a report completes in the Factory, BIRT must wrap up execution. Scripts that execute during the shutdown phase deal with registering the report document, reporting completion status and so on.

3.3   Element State Objects

All BIRT expressions and methods execute as methods on *element state* objects. Every report element has two aspects: a design object that provides information defined in the design file, and a state object that represents a specific instance of the element within a report. A single design object defines a given element. However, the Factory Engine creates any number of element state instances as the report runs: one for each use of the element.

The element state object is created in the Factory, is logically persisted to the report document file, and is rendered in the Presentation engine. (Note that we say "logically persisted." In actual practice, BIRT uses a more complex mechanism, but the implementation is mostly irrelevant for this discussion.)

Many types of element state objects exist. In general, there is a separate element state object for each report element defined in BIRT. Only styles do not have a corresponding element state object. Two types of commonly used state objects are report item state objects and data set state objects.

3.3.1   Report Item State Objects

ROM defines a number of scripts associated with report elements. These scripts are documented in the Scripts section of the various ROM elements. Each script executes as a method on a scripting object of type `ElementState`. This object exists to provide a "home" for the user-defined scripts, and to provide access to the object's properties, state and design. Scripting objects are orthogonal to execution contexts (discussed below). They are simply a way to cleanly define the resources available to the script.

Report items generally (though not necessarily) have an associated data row. All report items have an associated set of visual properties including color, size, position, and more. Scripts and expressions on report items execute sometime during the lifecycle of the report item, except for a number of special cases discussed below.

Scripts on a report item may execute in the Factory, in the Presentation Engine, or both. Script writers should be careful to reference only those values and objects that are available in both contexts.

3.3.2   Data Set State Objects

A data set is a mechanism for retrieving data from an external data provider. Data sets work with a set of rows called the result set. Methods and expressions in a data set context often work with the data set itself, or with each row in the data set.

3.4   Script Contexts

BIRT defines a number of *script contexts*. A script context is an informal concept that defines the objects that are available for use within the script, and defines the time during report execution that the script is executed. Each of the above phases has its own script context. In addition, BIRT defines two additional contexts that span two or more of the above report phases.

- Data row context for scripts executed while reading a data set. Spans data transform, factory and presentation time.

- Element expression context for scripts executed when creating and/or displaying an element. Spans factory and presentation time.

3.5 Data Row Context

The developer can use expressions as part of the data set definition. The full set of expression support is to be defined by the Data Transform Engine, but the following are minimal requirements:

- Computed columns, computed when BIRT needs the value

- Custom filters, computed when BIRT performs the filtering

Computed columns may be computed during data transform, during the Factory or during presentation. Filters may be executed during data transform or during the Factory. Therefore, the developer generally has no way to predict when a computed column script will execute, and so both computed column and row filter scripts should be written so that they will work correctly at any of the defined times.

3.5.1 Available Resources

Scripts in data set context can access any of the following in addition to the standard resources:

- Columns in the current data row

For example, it is natural for a row filter to reference a report parameter or configuration parameter. A computed column can reference other columns in the same row, or perhaps a report parameter.

3.5.2 Practices to Avoid

Scripts in the data set context should not reference the following, as they may not be available in all three phases:

- Data set state

- Variables computed during the Factory

- Contents of the report created in the Factory

- Presentation-time information such as user name, locale, or target output device.

For example, a computed column should not reference a variable computed in a script associated with a report element event because the computed column may be computed before the element is created (that is, during data transform time), or after the element has been completed (that is, at presentation time.)

3.6 Computed Columns

The user can define a computed column within a data set. A computed column is defined as a column whose value is set based on an expression. The expression is evaluated in the data row context, and presented just like any other columns in the data set. A computed column has two parts:

- A column definition (name, etc.)

- An expression.

A computed column can be thought of an assignment:

```
myColumn = expr
```

Computed columns defined on the data set itself are named by the user. Others (such as a sort key defined as an expression) are defined implicitly.

The computed column script may be run during three phases:

- Data transform phase: During a pre-processing pass that implements data transforms requested by the report.

- Factory phase: In some cases, a computed column may be computed while building the report. For example, suppose that sorting can be done by the underlying database, and so no pre-processing pass is required. In this case, rows are read from the data set while the Factory builds the report. If the row contains a computed column, the column is computed just before the content for the column is created.

- Presentation phase: BIRT employs a number of optimizations to reduce the size of the report document file, and increase the speed of Factory. Suppose that a data item references a computed column. Suppose that this data item is the *only* reference to the computed column. If so, then the value of the computed column need not be computed until we are ready to present the data item. If BIRT can detect that such an optimization is possible, the computed column script may be run at presentation phase instead of during Data Transform or Factory phase.

A computed column can reference:

- Any of the standard resources defined above.

- Any non-computed column in this row.

- A computed column defined *before* this column.

- Columns in a row for a data set that encloses this data set.

- Aggregates

User-defined computed columns are evaluated in the order that the user defines them. A computed column can reference other computed columns, but should only reference those previously evaluated. Computed columns that BIRT creates implicitly execute after all user-defined computed columns and can reference such columns.

A computed column can reference aggregates. The Data Engine will determine if it must make two or more passes to evaluate the expressions. The user can define an ambiguous aggregate. (x = sum(x)). BIRT *may* report this as an error, or may simply provide an undefined result.

See the section below for syntax for referencing columns within the current row, columns from enclosing data sets, and parameters.

3.6.1 Computed Column Evaluation Timing

BIRT guarantees that a computed column is computed before it is referenced. Typical references include:

- A filter (the column is computed before the filter)

- Sort (the column is computed before making the sort comparison)

- Data element (the column is computed before the data is formatted for presentation)

- Group key (the column is computed before making the group comparison)

The key point to remember is that BIRT ensures that a computed column is evaluated no later than its first reference, but reserves the right to compute it no sooner than that time. (That is, if a column is referenced only in a data item, its computation may be postponed until presentation time.)

3.7 Custom Filters

The user can specify data set filters using expressions. Such filters have the same scope rules as for computed columns. However, filters execute only in the data transform or factory phases, never (at least in the first release) in the presentation phase.

Filters can be in one of two forms:

```
myColumn == expr
```

Or,

```
expr
```

The first form does an implicit equality check for a given column. The left-hand side, given by the expression, is the value to compare. The right-hand side, given by the user in the UI or schema, is the column to test. This form makes it easy for the UI to create routine, simple equality expressions based on a single column.

The other form is a simple Boolean expression. This expression can reference any valid set of columns, parameters and other values. This form is needed for more complex expressions, such as those that use other relational operators, those that use multiple columns, etc.

In both cases, each filter will be AND-ed with other filters. That is, a row will be accepted only if all filters evaluate to true. (The developer can OR expressions together simply by creating an expression that contains the required logic, using the second form of filter described above.)

3.8 Element Expression Context

Report element expressions provide data values for the Data Item, a computed URL in a hyperlink, the highlight expression for a highlight rule, a group key expression, etc. In each case, the expression evaluates to a data value. The type of the data value is automatically computed by BIRT. The following is a simple expression:

```
row.customerName
```

The following is a computed value:

```
row.price * row.quan
```

The following is a scripted expression:

```
if ( row.custType == "C" )
   { row.price * row.quan }
else { row.price * row.quan * row.discount }
```

And the following is an aggregate:

```
Finance.Percent( Sum( row.price * row.quan )
                 Sum( row.price * row.quan, "overall" ) )
```

Element expressions execute either at Factory or Presentation time. Some (such as the group key expression) execute only at Factory time. See the specification for each expression for details.

The user writes BIRT expressions for any of the various expression properties defined in the Report Object model. The user can also insert scripts at various points during report execution.

3.8.1 Available Resources

An element expression can assume that columns in the current data row are available in addition to the standard resources.

Element expressions may want to refer to information that is available at both factory and presentation times, as long as the script is designed to work with the data at that time, and does not save the results. For example, suppose a data item is set to display the current locale. BIRT will execute this function at presentation time, showing the locale set for the user that caused the data item to be presented. This may be exactly what one user expects, but another may have expected to see the Factory time locale. Hence, it is important to understand when a given script can be executed.

3.8.2 Practices to Avoid

Because element expressions execute during both factory and presentation phases, such scripts should not access:

- The data set or data source

- Elements created after this one. See later sections on element creation timing.

- Factory-specific of presentation-specific information such as jobs, user data, etc.

3.9 Resource Availability Summary

The following table shows what resources are available in which contexts. The table uses the following codes:

- ✓ (Check): the resource is available

- — (Dash): The resource is not available.

- *: The resource is sometimes available; code should avoid using the resource, or should check the context before accessing the resource.

- (n): Indicates a note at the bottom of the table.

Resource	Startup	Data Trans- form	Factory	Presen- tation	Shut- down	Data Row	Element Expr.
Report Design	✓	✓	✓	✓	✓	✓	✓
Parameter Values	✓	✓	✓	✓	✓	✓	✓
Config. Variables	✓ (1)	✓ (1)	✓ (1)	✓ (1)	✓ (1)	✓ (1)	✓ (1)

Resource	Startup	Data Trans- form	Factory	Presen- tation	Shut- down	Data Row	Element Expr.
Report Elements	—	—	✓	✓	✓	*	✓
Data Set State	—	✓	✓	—	—	*	*
Data Source State	—	✓	✓	—	—	*	*
Execution Job	*	✓	✓	*	*	*	*
Print Job	*	—	—	✓	*	*	*
Presentatio n message	*	—	—	✓	*	*	*
User Info	✓ (2)	✓ (2)	✓ (2)	✓ (2)	✓ (2)	✓ (2)	✓ (2)
Data Row	—	✓	✓ (3)	✓ (3)	—	✓	✓ (3)
Report Object	✓	✓	✓	✓	✓	✓	✓
Custom functions	✓ (4)	✓	✓	✓	✓	✓	✓

Notes:

1. Configuration variables are read when a report is loaded. Their value may differ between Factory and Presentation times if these activities occur on different machines, or at different times. In general, a config variable value read in the Factory does not imply that the same value will occur when presenting.

2. User information at Factory time is for the user who requested the report to run. User information at presentation time is for the user who asked to view, print or convert the report.

3. A data row is often available in the Factory and presentation phases, but it depends on the structure of a report. No row is available when a report is just starting or ending. No row is available for sections not bound to data (such as titles.) Availability of the data row is for any given script easily determined by looking at the structure of the report.

4. Custom functions are available during startup only after the code that defines the function is executed. (This is because JavaScript must execute a function definition to define the function.)

4. Overview of BIRT-Defined Variables and Classes

BIRT defines a number of global variables that allow expressions and scripts access to data rows and information about the report. BIRT provides a rich set of classes that allows the application to introspect the report design and report document, and to change the report document. Notice that JavaScript uses the term object (instead of class) when it refers to a custom-defined type.

4.1 Global Variables

Variable Name	Type	Description
report	Report	Information about the report design and execution.
params	Array of values	The values of parameters passed to the report, indexed by parameter name. Values are simple types: numbers, strings, etc.
config	Array of values	The values of configuration variables set by the environment, indexed by config variable name. Values are simple types: numbers, strings, etc.

4.2 Classes

BIRT defines a complete set of classes that let the user work with many aspects of the report design, report task, data set and more. The following provides an overview of these classes. Later sections detail the properties and methods of each class.

Class Name	Description
Report	An entry point to top-level information about a BIRT Factory or Presentation session. Provide access to report design, parameters, configuration variables, report document, data sources, and so on.
ElementDefn	Describes the design-time properties of a report element.
ElementState	Represents the factory-time or presentation-time object that has scripts. Provides access to the runtime state of the element, its design, and more.
ColumnDefn	A definition of a column within a data row.
DataRow	A result set data row for a data set.
DataSet	A run-time description of a BIRT data set.
DataSource	A run-time description of a BIRT data source.
DateTimeSpan	A collection of functions for working with a span between two dates or times.
Total	A static class that provides methods for computing aggregates over data.
Finance	A collection of financial functions.

Class Name	Description
Job	Describes the iServer job that initiated the report.
Message	Describes the Presentation message being processed in the presentation engine.

5. Report State, Design and Document Classes

BIRT provides a global object `report` that provides all information about a report. The report object is of type `Report` and provides access to a wide range of information:

- Report design information
- Report document information
- Parameter values passed to the report
- Configuration variables
- Data source information

The `design` property represents the root of the report design. This object has slots for things like body, page-setup, styles, data-sets, data-sources and so on. Hence, from the `design` object, the application can access all parts of the report design. For example, to find a data set named "DS":

```
ds = report.design.findDataSet("DS")
```

Similarly, to get information about a parameter named P:

```
p = report.design.parameters["P"]
```

Because JavaScript treats properties and array references similarly, the above can also be written as:

```
p = report.design.parameters.P
```

Similarly, the application can get information about the report document as follows:

```
filename = report.document.fileName
```

And, the application can get the value of a parameter as follows:

```
startDate = report.params["Start Date"]
```

Note the distinction between the design ("meta-data") of a parameter, and the value of a parameter.

5.1 `Report` Class

Provides access to all information about the report and the context in which the report is running.

Synopsis

```
report
```

Constructor

The application cannot create instances of this class. Instead, the application accesses the BIRT-created instance stored in the "report" variable.

Description

The Report class provides access to all information about the report and the context in which the report is running. The report object is available using the `report` global variable. The `report` object is a state object, meaning it provides information about the report execution or presentation.

Properties

`design`

> Returns a `ReportDefn` object that describes the design of the report.

> Scripts within a report can obtain information about the design of the report. For example, a library may contain a reusable component that displays or logs the values of each parameter in the report. Such a script can use the report design object to locate all the parameter definitions.

`Document (Availability: After release 1)`

> Returns a `ReportDocument` object that gives information about the report document being created in the Factory, or being read in the Presentation Engine.

> Every report creates a report document to hold the data within the report. The document is a file and may be temporary or persistent. Many details of the document are specific to the deployment environment, but others are generic to all reports.

`params`

> Returns an object array of parameter values passed to the report. The application can then access a parameter using property or array syntax.

> There are two aspects to report parameters: the parameter definition in the report design object, and the parameter value provided by this property. Parameter values are passed to the report when it is run. Parameter values are saved in the report document file and are available at presentation time.

> The global variable "params" is a shortcut to this property.

> Parameter values are read-only; scripts cannot change them.

> Parameter values are a number, string, or date. Check the report design for the allowed data type for any given parameter. Users can elect not to enter a parameter if the design allows. Such parameters will return the JavaScript `undefined` value. The user can also enter a null value for the parameter, in which case the parameter returns the JavaScript `null` value.

> Parameters can be accessed as properties or using array syntax. The following are all equivalent:

```
report.params.startDate
report.params["startDate"]
params.startDate
params["startDate"]
```

dataSources [Availability: after first release]

Returns an array of data sources (connections) defined in the report.

A report design can include any number data sources. Each data source has a name and a state. The state indicates if the data source is connected or not. The application can access the current state of any data source using the dataSources object. Objects can be addressed by name or by position. The specific data source then provides access to the design and state for that data source.

config

Returns an array of configuration variables available to the report. Represented as a name/value pair.

Reports typically obtain information about their environment using configuration variables. A configuration variable may indicate the server on which a database resides, the location of images, or the company name to appear as a report title. Configuration variables are defined externally to the report and are set in the environment. Operating System environment variables can be used to implement a configuration variable, or the deployment system may have an alternative implementation.

The application can index configuration variables by name or position. The value of each variable is a string, or undefined, if the value has not been set.

See Also

ReportDesign Class

DesignElement Class

ReportDocument Class

DataSource Class

5.2 ReportDefn Class

Provides overall design information about the report.

Synopsis

report.design

Extends

ElementDefn Class

Methods

findStyle(styleName)

Finds a style given a style name.

findReportItem(itemName)

Finds a report item given a report item name.

findDataSource(sourceName)

Finds a data source given a data source name.

findDataSet(dataSetName)

Finds a data set given a data set name.

```
findParameter( parameterName )
```

Finds a parameter given a parameter name.

Description

This class represents the definition a report design element. In addition to the standard services of a definition class, the report design provides a way to locate elements by name.

See Also

Report Design Element

5.2.1 `findStyle` Method

Finds a style given a style name.

Synopsis

```
findStyle( styleName )
```

Summary

Availability: First release

Arguments

```
styleName
```

The name of the style to locate.

Returns

An `ElementDefn` object that describes the style, or `null` if no style of the given name is found.

5.2.2 `findReportItem` Method

Finds a report item given a report item name.

Synopsis

```
findReportItem( itemName )
```

Summary

Availability: First release

Arguments

```
itemName
```

The name of the report item to locate.

Returns

An `ElementDefn` object that describes the report item, or `null` if no report item of the given name is found.

5.2.3 `findDataSource` Method

Finds a data source given a data source name.

Synopsis

```
findDataSource( sourceName )
```

Summary

Availability: First release

Arguments

```
sourceName
```

The name of the data source to locate.

Returns

An `ElementDefn` object that describes the data source, or `null` if no data source of the given name is found.

5.2.4 `findDataSet` Method

Finds a data set given a data set name.

Synopsis

```
findDataSet( dataSetName )
```

Summary

Availability: First release

Arguments

```
dataSetName
```

The name of the data set to locate.

Returns

An `ElementDefn` object that describes the data set, or `null` if no data set of the given name is found.

5.2.5 `findParameter` method

Finds a parameter given a parameter name.

Synopsis

```
findParameter( parameterName )
```

Summary

Availability: First release

Arguments

```
parameterName
```

The name of the parameter to locate.

Returns

An `ElementDefn` object that describes the parameter, or `null` if no parameter of the given name is found.

5.3 `ElementDefn` Class

The report element design class, `ElementDefn`, provides information about the element as defined in the design file. This class provides a set of properties as defined in ROM. It also provides a set of "slots" that contain other element designs.

The report element state class, `ElementState`, provides information about a specific instance of a report element within the Factory or Presentation Engine.

Summary

Availability: First release

Properties

`name`

> Provides the name of the element. Not all elements have names.

`extendsElement`

> Provides the name of the parent element that this element extends. Not all elements allow extension.

`propertyDefns[]`

> An array of property definitions. Properties are indexed by name. The return value is a `PropertyDefn` object defined below. This list contains both BIRT-defined and user-defined properties.

`slotDefns[]`

> An array of slot definitions. Slots are indexed by name. The return value is a `SlotDefn` object defined below.

`[property]`

> Provides the value of a design-time property that has this name. The properties are those described for each element in the ROM specs. (Issue: before or after cascading?)

`properties[]`

> An array of property values indexed by property name. This is the same as accessing the property directly, but allows access to properties that happen to take the same name as one of the BIRT-defined runtime properties.

`slots[]`

> Many elements are containers: they contain other report elements. The slots array contains the slots as defined in the ROM specs.

`allowsUserProperties`

> True if the element allows user properties, false otherwise.

`hasStyle`

> True if the element has a style, false otherwise. (Issue: What does hasStyle mean?)

`container`

> Returns the element definition, if any, that contains this element.

`containerSlot`

Gets the name of the slot that contains this element.

Description

The element definition class provides the design-time description of report objects defined in the design schema file. The class provides a generic set of properties and methods that work for all elements. There is often no specific class for each kind of design element.

This is a base class that describes all report elements in the report design. It provides design-time information. A single element definition object exists for each element defined in the design file. The application cannot change the properties of such an object; all changes must be made to the design itself at design time.

5.4 `PropertyDefn` Class

The `PropertyDefn` class provides meta-data information about a BIRT- or userdefined property.

Properties

`isUserProperty`

True if this is a user-defined property, false if it is a system-defined property.

`name`

Internal property name.

`group`

Name of the property group that contains the property.

`canInherit`

True if the property can be inherited, false if not.

`isStyle`

True if the property is a style property such as `fontSize`, false otherwise.

`type`

Property type.

`choices`

Array of choices for the property.

`defaultValue`

Default value for the property.

5.5 `SlotDefn` Object

5.6 `ReportDoc` Class

Provides information about the report being produced (in the Factory) or that is being viewed (in the Presentation Engine.)

Summary

Availability: After first release

Properties

`design`

> Returns the root design object for this instance.

`fileName`

> The name of the file (if any) to which this file is written. The meaning of the name depends on the destination.

`destination`

> The kind of destination such as "file-system" or "temporary".

`body`

> A list of sections within the body.

Description

Note: *Much remains to be worked out here.*

6. Scripted Control of Report Items

The previous section focuses on accessing report design information in scripts. The primary purpose of scripting in BIRT is however to allow full control on the data displayed in a report, and to customize report appearance based on the data. Report items are the visual elements that make up a report. Scripts can control various aspects of report items, such as the value displayed, and setting color, size, font, etc. To understand how to control report items within scripts, we must first discuss how items work.

Report items have two aspects: the static design-time description and the dynamic generation or presentation time state. As an example, a design-time description of a data item may not define the `fontSize` property. The design-time value of that property is `null`. The run-time instance of a data item might return an actual value, which is the result of applying the style search algorithm described in ROM Style spec. The static description of a report item is captured by the `ElementDefn` class described in last section. The runtime state is captured by the `ElementState` and `ItemState` classes discussed in this section.

6.1 Report Item Lifecycle

Report Items follow a specific lifecycle:

- Created at the proper time from the Factory.
- Populated with data.
- Written to the report document file.
- Read from the report document file.
- Prepared for rendering
- Rendered to a target output format.

As the Factory runs, it defines report content by combining data rows with rules in the report design. The Factory determines when to create a given item based on these rules. For example, a List header is created at the start of a list, and a list detail is created for each row.

Once an item is created, it is often populated with data. For example, the list header may display data from parameters, from the first row in the data set, from totals over the data set, and so on.

Once the item is created, it may be saved to a report document file (if the report output is to be saved.)

When the user wants to view a part of the report, the Presentation Engine reads the required items from the document file and prepares them for rendering. The presentation engine then converts them into the target output format.

For reports created on demand, the write/read steps may be omitted, and the report output may be directly rendered to the target output format. Only on-demand reports are supported in BIRT release 1.

For performance reasons, the item creation step in the Factory may be *virtual*. For example, the Factory may determine that the report header simply displays data row columns and report parameters. It may decide to wait until presentation time to create the actual items in the header. This also means that concrete "report item objects" may not exist in report document. Factory and Presentation Engines could create report items based on the element design and the data stored in a report document, instead of a persistent report item object.

Because of this optimization, it is important for the script writer to ensure that code is placed in the proper script. In general, changing visual properties should be done in preparation for rendering, not in response to a Factory event.

See the section above about execution contexts for additional background information.

6.2 Element Expressions vs. Element Scripts

BIRT supports use of element expressions instead of hard-coded values on some properties. Examples include the value expression for data item, bookmarks on various elements, visibility and highlight conditions, etc. In addition, BIRT defines element scripts (i.e., event handlers) that are called at specific time points during report generation or presentation phases. Examples include the `onCreate` and `onRender` methods for almost all report elements, and `onStart`, `onRow`, and `onFinish` methods for listing elements.

One design goal of BIRT scripting is to allow most tasks that are traditionally performed through scripting to be achieved through element expressions (except for scripted data set). Some examples are given below:

- Format based on data type. A BIRT style defines three format strings, namely `numberFormat`, `datetimeFormat` and `stringFormat`, to automatically choose the right format string based on data type.

- Display negative numbers in absolute values with () around. This is achieved by specifying subpatterns in format string, such as `"#,##0.0#;(#,##0.0#)"`.

- Display negative numbers in red color. This is achieved through highlight rules.

- Display first name concatenated with last name in a single data item; or display row.price * row.quantity if the product is less than $1M, or row.price * row.quantity * 90% if the product is larger than $1M. Both can be easily done using value expressions.

- Display "Excellent", "Very Good", "Good", "Fair" and "Poor" based on test scores and a set of range rules. This can be achieved using map rules.

- Display correlated values in another data item. Assume that one column (with name "A") in a table displays the order size (`row.orderSize`), and another column (with name "B") displays the order size as a percentage of the total order size. It is recommended that B's value expression be written as `row.orderSize * 100 / totalOrderSize`, instead of `container.B.value * 100 / totalOrderSize`. That is, references to other elements should be avoided. In fact, if the second column is to be searched or extracted, a computed column is recommended.

- Hide two columns if the viewer is not from a management team. This can be achieved using visibility expression.

Nonetheless, BIRT also supports modifying report element properties at runtime through element scripts, to fulfill more complex requirements. It is recommended, however, that a report developer first confirm that element expressions do not fulfill his needs before trying to use element scripts. In BIRT release 1, only JavaScript code is allowed to be executed as element script. This restriction applies to report items, but not data set, data source, and data rows, etc. See section 7 for detail.

Changes made at factory phase are persistent into report document, and are available at presentation time. Changes made at presentation phase are only applicable at presentation time and do not affect the report document.

6.2.1 Setting Expression Properties

Accessing and setting properties that are defined as Expression type deserve some further attention. For example, a data item has a `value` property, whose type is an Expression. Notice that the data item does not really have a value; instead, it has an expression which can be executed to get the value when needed. What this means is that in its `onCreate` method, `me.value=5` is not valid even if the data item displays an integer, because what is set is the value expression, not the actual value. Instead, the correct syntax is `me.value="5"`. The value expression is evaluated at a later time in Factory or Presentation Engine to yield the integer value 5. This logic applies to all expression properties.

If, no the other hand, an expression property appears on the RHS of an expression, it is also the expression string that is assigned to the LHS, not the value after evaluating the expression.

Suppose a report developer wants to compute a value in code (i.e., `onCreate`) and finds that he has to access and store the actual value. This case should be rare, and the recommended approach is to use a computed column. Another way to achieve the goal is to define a custom member variable for the report item. The type of the custom variable is a BIRT supported type other than Expression.

6.3 `ElementState` Class

Represents the run-time state of a report element.

Summary

Availability: Partially in First release

Properties

`design`

Returns the element definition object for this instance. The value is of type `ElementDefn`.

[property]

Allows the application to get or set the runtime value of the report element property with the given name. See the report element specification to determine which properties can be set at runtime.

`container`

Returns the element state object, if any, that contains this element. The element is of type `ElementState`.

slot

Provides access to the elements that this element contains. Slots are defined for each element in the ROM specs.

`dataSet`

Returns the runtime data set object of type DataSet for this instance, if this report element has a data set. This property is undefined if the report element has no data set.

`row`

A shortcut to dataSet.row. Returns the current data row of type DataRow for the data set associated with this report item instance. If this report element has no data set, this property is undefined.

`rows`

A shortcut to dataSet.rows. Returns the current data rows (of type DataRow[]) for the data set associated with this report item instance. If this report element has no data set, this property is undefined.

Description

The element state class defines the runtime state of a report element. There are specific subclasses for various kinds of elements such as data sources, data sets, report items and so on.

All design-time properties of the report element appear on the runtime state object. An `ElementState` object returns the actual value that will be used to render the report. The runtime value may differ from the design time value if the application has customized the property in scripts, if a highlight rule applies, and so on. Not all properties can be set at run time. See the ROM specs for which can be set. If the application attempts to set a read-only property, then an exception is raised.

DataSet Class

DataRow Class

6.4 Commit Points and Execution Blocks

6.4.1 Accessing Other Report Items

One can navigate from one `ElementState` object to another at run time. For example, if in a table, the header row has a data item named "A", and the detail row has two columns named "B" and "C", when we are on "B", the following scripts are valid:

me.fontName →	returns the font name property for this item
me.container →	returns the parent (a table row) object
me.container.C →	returns report item instance created with the same data row but based on item C

me.container.container	→	returns the detail slot
me.container.container.container	→	returns the table object
me.container.container.container.header	→	returns the header slot
me.container.container.container.header.C →		returns the item C

Navigations lead to further complications because a report item can access and modify not only its own properties, but also properties defined for other report items. One such example if for the font color of one report item ("C") to depend on the value in another report item ("B"). Another example is for background color of a table to change based on an aggregate value displayed in the table footer (i.e., if the total sales is less than a threshold, change the table's background color to red). The first example should in fact be handled using computed columns. The second example is what leads to the concept of execution blocks.

6.4.2 Commit Points in Factory Engine

BIRT engine is designed with the assumption that it can only use limited memory and has to optimize for performance, because it may eventually be used as an enterprise-class engine to render reports with millions of rows. The implication of this is that the Factory engine has to "commit" after generating a portion of the report. Commit means that committed contents cannot change without screwing up the page layout. This is not equivalent to writing to disk, even though they are likely coupled during report document generation.

BIRT engine therefore defines many "commit points" at which contents are committed to the page. At factory time, the following commit points are defined:

- At the end of generating a container (table, list, grid and free-form)

- At the end of generating a listing band (header, detail, and footer)

- At the end of generating a group band (group header and group footer)

If a table band has multiple rows, each individual row does not define a separate commit point. However, in a table detail band, each data row (not table row!) defines its own

commit point. Simple report items such as date and image items do not define separate commit points.

Because a table can contain multiple bands, this means that a table band may be committed before the table itself is committed.

6.4.3 Execution Blocks in Factory Engine

Commit points effectively divide Factory Engine execution into "blocks", with each execution block starts at the time the Factory Engine starts to generate a container or band, and ends when the Factory Engine hits the corresponding commit point. It is apparent that one execution block could be nested in another one. We use term "parent block" and "child block" to convey such a relationship. The execution block for a report item is the execution block that is introduced *by the creation of its container or band ancestor*, not the execution block that is introduced by the creation of itself. The following rules determine what can be modified at runtime, starting with a specific report item instance:

- If two report item instances share the same execution block, they can reference and modify each other's (rum-time modifiable) properties.

- A report item instance may modify properties on its parent instance, grandparent instances, and so on, if the modification does not affect pagination (for example, `fontColor`). If it modifies properties that affect pagination, the modified property value will still be observed at presentation time, but the pagination does not change.

- A report item may access, but not modify other report item instances that are not its ancestors and do not share the same execution block with the first item. Even in this case, if the second report item has not been created, null value will be returned. This means that it is allowed to access a report item in table header from a report item in table detail, but not vice versa; nor can the report item in table detail band access another item in the table footer.

To summarize, BIRT's goal is to allow properties on report items in the same execution block and ancestor report item instances of the current instance be accessible and modifiable. In addition, report items that have already been created (and potentially have already been committed to disk) are also accessible (but not modifiable).

It is possible that a script can still navigate to a report item instance deemed non-accessible from current the item, and receives an object that is not null. BIRT just does not guarantee that such a reference is always valid.

Report item instances that are in the same execution block are created all at once. The `onCreate` method on each report item instance is subsequently invoked. This guarantees that report item instances can refer to each other independent of the order that the report items are defined in the design file.

6.4.4 Presentation Engine

Presentation Engine observes similar execution blocks as Factory Engine. As a result, report item instances that are in the same execution blocks are created (i.e., restored based on report document) all at once, and the `onRender` method on each report item instance is subsequently invoked.

At presentation time, only properties on report items in the same execution block can be modified, and only such report item instances and ancestor report item instances can be accessed.

Report item emitters are not guaranteed to be invoked (at about the same time) right before the commit point for report item instances in the same execution block. In fact, it is likely that they are invoked at separate time points before the commit point.

6.5 Style Precedence

The ROM style spec defines a precedence rule in the style search algorithm. Specifically, highlight rules takes precedence over element hierarchy, which in turn takes precedence over containment hierarchy. With scripting, the precedence rule needs to be augmented to take into account style property changes through scripts.

The augmented search algorithm gives presentation time scripting the highest priority, followed by generation time scripting, highlight rules, element hierarchy, and containment hierarchy. The following examples demonstrate the use of such a rule.

Consider a data item with `fontColor` set to red in design, and a value expression `row.quantity`, a highlight rule specified as follows:

`highlightTestExpr`:	not specified, default to the value of this data item
`operator`:	>
`value1`:	100
`style`:	specifies one property: `fontColor=green`

 The highlight rule says that if the order quantity is greater than 100, display the report item in green. If the data item receives value 110, it is displayed in green. Now if the `onCreate` or `onRender` method has script `me.color=red`, the report item will be displayed in red. If the `onCreate` or `onRender` method has script `value="90"`, even if `row.quantity` returns 110, the data item is displayed in red. Notice that in this case, the quantity column may not even be fetched at runtime, because the value expression is reset to "90". Also notice that `me.value=90` is invalid as a script on the data item.

Consider the earlier example where the developer wants to set the background color of a table to change based on an aggregate value displayed in the table footer (i.e., if the total sales is less than a threshold, change the table's background color to red). If instead, the developer wants to set the `fontSize` property of the table instead of background color, notice that all data items who has `fontSize` property set in the element hierarchy will not be affected by the modification to the table style. This is because element hierarchy takes precedence over containment hierarchy, even when the container property is set through script.

6.6 Modification Window

This section has already touched on the `onCreate` and `onRender` methods. They are further discussed here in the context of the modification window in Factory and Presentation Engines. In particular, BIRT defines two specific windows in time during which the application can change the properties of an element. The first is the *factory window* which occurs while the Factory creates the element. Changes made to the element during this time are written to the report document file. The other is the *presentation window* which occurs when the Presentation engine prepares the element

for rendering. Changes made to the element during this time are written to the target output format, but not to disk.

Changes made to an element outside these windows result in undefined behavior: the changes may be retained, or they may be discarded; neither outcome is guaranteed.

The following element scripts provide opportunities to make element changes. They are available on every report element:

onCreate

A script executed when creating the element in the Factory. If this script exists, the Factory is forced to create the element; the Factory cannot create a "virtual" instance of the element as described above.

onRender

A script executed before converting the element to a target output format. This script executes in the presentation engine. In general, the application should put visual customization code in this script for performance.

Many elements define value or other expressions. These element expressions generally execute at presentation time, but may execute at Factory time in some cases. Code in these scripts must not assume one or the other of these contexts, and must work in both. Value expressions can have side effects that produce the same result as code in the onCreate or onRender scripts.

Additional element scripts are defined for listing or listing group elements:

onStart

Available for both elements. Executed after a listing element is created, the data set is open, but before header band is created. It is a convenient place to initialize custom running total variables. Works similarly for listing group element.

onRow

Available only for listing elements, i.e., table and list elements, but not on each individual band, i.e., table detail. onRow means on each data row, not on each row in a table band. It is called for each data row within the scope of the table, not the scope of the table band. It is called right after factory receives the data row from Data Engine. It is a convenient place to update custom running total variables.

Notice that data set might define its own onRow method, which will be called just after the row is fetched from the data set, but before any transform is done on the data.

onFinish

Available for both elements. Executed after a listing element is to be finished, i.e, after the footer band has been processed. The data set is still open so the last row can still be accessed. Works similarly for listing group element.

7. Data Classes

This section describes the many classes that allow scripts to work with runtime instances of data sources and data sets.

7.1 Runtime Instances of Data Classes

When a report item that uses a data set is executed by the BIRT factory, runtime objects that represent the states of the data set and data source used by the report item are created. These objects contain properties like the `queryText` of the data set, ODA connection properties of the data set/data source, the current data row produced by the data set etc. Scripts can use these objects to examine the values of the runtime properties and modify some of them to affect the execution of the data set. Runtime values of properties have initial values copied from corresponding design objects.

A `DataSet` object represents the runtime state of a data set. A `DataSource` object represents the runtime state of a data source. A `DataSet` object is accessible via the `ElementState.dataSet` property. A `DataSource` object is accessible via the `DataSet.dataSource` property, or the `report.dataSources` array.

For example, if a Table report item uses a data set named "Customer Query", which in turn uses a data source named "SaleDB", scripts defined on the Table item will return:

dataSet.name	-> Returns "Customer Query"
dataSet.dataSource	-> Returns the DataSource named "SaleDB"
report.dataSources["SaleDB"]	-> Same as above
dataSet.row	-> Returns the current data row, if data set is open
dataSet.row.CustName	-> Returns the value of the CustName column
	of the current data row, if data set is open
row.CustName	-> Same as dataSet.row.CustName

Runtime DataSet or DataSource objects are also the scope in which scripts defined on the data set and data source objects are executed. For example, if a data set is named "Customer Query", and it defines a `beforeOpen` script, the `beforeOpen` script executes as a method on the runtime DataSet object. The following expressions defined in the script will return:

name	-> Returns "Customer Query"
me.name	-> Returns "Customer Query". The "this" object
	refers to the DataSet object itself
row	-> The current data row. Returns null since no data
	row is available yet when beforeOpen script is executed

7.2 `DataSource` Class

Provides run-time state for a data source.

Extends

`ElementState` Class

Description

The data source class provides access to the Factory-time state of data source. The DataSource object can be accessed via the DataSet object that uses it:

ds = myDataSet.dataSource

Alternatvely, a DataSource object can be accessed from the `dataSources` property of the report object, which provides a list of data sources keyed by name. If a data source is called "SalesDB", the application can access it as:

```
ds = report.dataSources["SalesDB"]
```

or

ds = report.dataSources.SalesDB

Properties

```
extensionProperties      (Modifiable: Yes)
```

If the data source is an ODA data source, this property returns a collection of values for the data source properties defined in the ODA driver extension of extension-point org.eclipse.birt.data.oda.dataSource. These properties are generally used by the ODA driver to connect to the external data provider. This collection is updatable. An ODA data source's beforeOpen script is the usual place to update this collection to programmatically alter data source connection information at report execution time. Updating this collection after a data source is open has undefined behavior.

If the data source is a Scripted Data Source, this property returns null.

```
extensionID       (Modifiable: No)
```

Returns the value of the extensionID of the ODA data source, as defined by the ODA driver extension. The extensionID uniquely identifies a type of data source in the BIRT environement. This property cannot be updated.

If data source is a scripted data source, the value of this property is "SCRIPT".

```
isOpen            (Modifiable: No)
```

Returns a Boolean value indicating whether the data source is connected.

Example

The following is a way to customize the connection properties before opening a data source. The following might appear in a "beforeOpen" event for a JDBC data source

```
ds = report.dataSources["SalesDB"];
ds.extensionProperties.odaUserName = "sesame";
ds.extensionDriverProperties.odaPassword = "open"
my_trace_func("Connecting to data source " + ds.extensionID );
```

7.3 `DataSet` Class

Provides run-time information about a data set.

Extends

`ElementState` class

Properties

`row    (Modifiable: No)`

The data set and report item elements define a property called "row" that represents the current inner-most row. Columns are accessed as members:

`row.customerName`

Or

```
row["customerName"]
Or
row[column_index]
```

The first form is available if the column name is a valid JavaScript identifier. The second form allows access to all projected columns regardless of name. This property is null if the data set has not current data row. This can happen if the data set is not open, or if its result set is empty, or the result set has advanced past its last row.

`rows[ ]            (Modifiable: No)`

Data sets and report items often appear within a nested set of data sets. The `rows` property allows access to rows within this hierarchy. If *n* rows are active, the others are accessible via the rows array. `Rows[0]` is the top-most row, rows[*n-1*] is the same as `row`.

`rows[0].clientCode`

`queryText          (Modifiable: Yes)`

Provides the query text to be sent to the ODA data source. Returns null if the data set does not have a query text, or if the data set is a Scripted data set

The application can modify the query text in the `beforeOpen` script.

Example:

`myDS.queryText = "SELECT * FROM Customer WHERE custName = 'Fred'"`

`columnDefns[ ]          (Modifiable: No)`

An array of column definitions. Each element of the returned array is an object of type ColumnDefn.

Some data sets have a fixed data row definition, others adjust the data row depending on the needs of the report. Data sets with a fixed row definition can return their row schema whether the data set is open or not. However, data sets with a dynamic definition may return null if the application asks for the column definitions when the data set is not open.

`extensionProperties[ ] (Modifiable: Yes)`

Provides the ODA driver-specific extension properties defined in an ODA dataSet extension. Returns a collection of property values. Returns null if the data set is not an ODA Data Set. Content of this collection is updatable. An ODA data set's beforeOpen script is the usual place to update this collection in order to dynamically alter data set behavior at report execution time. Updating this collection after the data set is open has undefined behavior.

Example:

```
myDS.extensionProperties.queryTimeout = 20;
```

extensionID   (Modifiable: No)

Returns the extensionID of this data set as defined by the ODA driver providing this data set. An extension ID uniquely identifies a type of ODA data set in the BIRT environment.

If this data set is a Scripted Data Set, this property has value "SCRIPT"

dataSource   (Modifiable: No)

Returns the DataSource object used by this data set.

See Also

ColumnDefn class

7.4 `DataRow` Class

Represents one row fetched from a data set.

Synopsis

DataRow

Description

The data row class represents one row from a data set. The row provides access to the columns within the row. Rows are defined by a columnDefns meta-data, which is accessible from the DataSet object.

A DataRow object is not meant to be constructed by the user. The BIRT engine will provide one when the scripting context requires it.

Each row column is represented as a property. The property name is the same as the column name as reported by the data set. If the column name happens to be a valid JavaScript identifier, the application can use the dot-syntax to access the column:

```
row.customerName
```

All columns are accessible using the array syntax:

```
row["Customer Name"]
```

Rows can contain anonymous columns. In this case, the user can access the column by position:

```
row[5]
```

Columns are available using a 1-based index. (Column 0 is the BIRT-provided row count column.)

The column names "rowCount", "dataSet", and "columnDefns" are reserved and are not accessible from the row object because they are hidden by current and future BIRT-defined properties of the row object.

Properties

[column]

The column properties return the value of a column given the column name. The set of column names is dynamic and is computed each time the data set is opened. The

set will depend on the actual columns required by the users of the data set. The application can reference columns by name or position. If by position, column 0 is the row count, column 1 is the first data column.

Databases allow a null value. BIRT represents this using a JavaScript null value.

Scripts cannot set the values for column properties in a row object returned by a data set.. The effect of setting a column value in such cases is not defined. The script can, however, set column values on a row object created by the BIRT engine during the execution of a Script Data Set's fetch script.

The row properties are the following:

- If the data set can provide column names, then those names appear as properties.

- If a column name satisfies the rules of a JavaScript identifier, then the expression can use the dot form to access the column: `row.colName`.

- If a column name is not well-formed, then the script must use the array form to access the column: row["*colName*"]. The array form can be used for well-formed column names as well.

The application can also access the array using the column index. Columns are indexed starting at 1. A special BIRT-provided indexed property, row[0], returns the index of the current data row within the current result set. The row index is 0 based. The first row is row 0, the second is row 1, etc. The index is based on the order that the rows are returned from the data set after any filtering and sorting.

`dataSet`

Returns the DataSet object that created the row.

columnDefnsReturns an array of ColumnDefn that defines the column metadata. It is a shortcut to dataSet.columnDefns.

See Also

`row` global variable

`rows` global variable

`ColumnDefn` object

DataSet object

7.5 `ColumnDefn` Class

Describes "meta-data" for a data row column.

Description

The column definition class describes a column in within a result set from a data set. The `ColumnDefn` objects are read-only, the application cannot change them or create instances of this class.

Each time that a data set is opened, it defines its result set schema. The schema may vary between uses because of different filters, aggregates, sorts, columns and so on. The schema is undefined when the data set is not open.

Properties

The application cannot change the value of these properties: they are read-only.

`index`

Returns index of the column within the result set data row for the data set. The index will match the position of the column definition with the schema array of the data row.

`name`

Returns the name of the column as defined by the data set. If the data set does not have column names, then returns null. The column name may be used as a property name within the data row for the data set.

`type`

Returns the data type of the column using one of the data types defined above: INTEGER, FLOAT, DECIMAL, BOOLEAN, STRING or DATETIME. The data type will be null if the column is defined by a BIRT expression, or if the type is not known.

`nativeType`

Returns the database native type name as a string.

`label`

Returns the column label.

`alias`

Returns the alias of the column,

Static Properties

`INTEGER`

A constant for integer columns. Evaluates to "integer".

`FLOAT`

A constant for floating-point columns. Evaluates to "float".

`DECIMAL`

A constant for fixed-precision decimal numbers. Evaluates to "decimal."

`BOOLEAN`

A constant for Boolean values. Evaluates to "boolean."

`STRING`

A constant for string columns. Evaluates to "string."

`DATETIME`

A constant for date-time columns. Evaluates to "dateTime"

See Also

`DataRow` class

`DataSet` class

8. Working with Aggregates

BIRT defines a number of *aggregation functions.* These functions aggregate (summarize) a set of rows. These functions are BIRT extensions to JavaScript. The Engine implements the features by dividing each aggregate into two parts: an aggregation phase that occurs during the data transform phase of the report, and a aggregate access phase that occurs during the Factory and Presentation portion. BIRT *rewrites* scripts that contain aggregates to create revised scripts that perform the two phases. The rewriting is mostly invisible to the report developer, though it may be necessary to be aware of the process when debugging an expression that contains an aggregate.

Totals are defined as if they were static methods on the Totals object. However, aggregates are actually implemented in the data transform engine portion of BIRT.

8.1 Aggregate Overview

Aggregates are computed over a *row set.* A row set is a group of rows from a data set. Each aggregate (except for count) works with one target column from each row. The aggregate summarizes the values of the target column in each row to produce a single final result. For example, the `sum` aggregate adds up the column values, while the `ave` aggregate computes an average of the column values.

The row set can be the entire contents of the result set, or the subset of rows within a group. The group is defined using the grouping features of a table or list. For example, one could total sales for an entire company (the entire data set), or for each region (the region group.) If the total is over the entire result set, then it is an *overall total*. If, however, the total is over a group, then it is a *group total.* Overall totals most often appear in the header or footer band or a list, or in the footer row of a table. Group totals most often appear in the header or footer band of a group.

Aggregates can also consider only rows that satisfy some filter condition. These are *filtered totals.* For example, a college enrollment report may want to show totals for the number of male vs. female students, or the number of students within each class level (freshman, sophomore, etc.) If the report itself lists students alphabetically, then the report footer could use filtered totals to count the number of students in each category.

8.2 User-Defined Aggregations

The developer can define his own aggregate function. The key components of an aggregate include:

- A function (or object method) which receives each row in the sequence.

- A global variable (or object field) that contains intermediate values. Zero, one or more may be required depending on the calculation.

- A function (or object method) which retrieves the final value at the completion of the sequence.

Note that the actual implementation will depend on the selected scripting language. A class-based solution is ideal; a global & function based solution is much less desirable.

User-defined aggregates mean that the development team does not need to anticipate the needs of every report developer; those with unusual needs can simply write their own statistical or other calculations.

8.3   One- vs. Two-pass Aggregates

A *one-pass* aggregate is one that can be computed as rows are read. For example, if a group footer displays a sum over the rows in the group, then BIRT can compute the total as the rows are read.

A *two-pass* (or *look-ahead*) aggregate, on the other hand, is one that must be computed before creating the content for each row. Consider the following example that might appear in a group footer. We want to show the balance of one particular account (the one given by the group) as a percentage of the total balance of all accounts.)

```
Total.percent( balance, Total.sum( balance, Total.OVERALL ) )
```

The `Total.OVERALL` parameter value tells BIRT to compute the sum aggregate over all rows even though the expression itself appears within a group total. In general, all look-ahead aggregates involve an aggregate at a higher grouping level, or an overall total.

To compute this expression, BIRT must make one pass to compute the total balance then can make a second pass to compute the value for each account.

In general, the aggregate type should be transparent to the report developer: the developer simply enters the desired formula and BIRT "does the right thing."

8.4   Grouping and Filtering

All aggregates allow two optional arguments to indicate filtering and grouping:

```
Total.aggFn( …, filter, group )
```

8.4.1   Filter Argument

The filter argument provides a Boolean expression evaluated on each row. Only rows that match the filter are considered when computing the aggregate. For example, a report could sum the credit limits of active customers to determine the maximum credit exposure.

The expression is executed in the data transform phase. It can access columns in the row, parameters, user-defined functions and other resources as defined in the context description above. For example:

```
Total.sum( row.CreditLimit, row.Active == 'Y' );
```

A filter argument of `null` or `undefined` means that no filter is provided. If the application provides no filter, the filter defaults to `undefined`.

8.4.2   Group Argument

Reports frequently want to display totals for the current grouping level. For example, suppose a report is grouped by customers and orders, and the detail records represent line items in orders. Each line item has a unit price and a quantity. Then, the following aggregate:

```
Total.sum( row.price * row.quantity )
```

This aggregate sums values over the current group. If placed in the footer for an order, it displays the sum of line items for that order. If placed in the customer footer, it displays the sum of all the line items for all orders for that customer. And, if placed in the report footer, it displays the sum of all line items for all customers.

Sometimes, however, the application wants to access a total from a different grouping level. For example, we may want to know the total value of one order as a percentage of all orders for the customer. In this case, we must access the total for a group other than the current group. We do this using the grouping argument:

```
Total.sum( row.CreditLimit, null, "Customer Group" );
```

The group argument can be one of the following:

- Null, meaning the current grouping level.

- The name of a group at or above the current grouping level. The name is the one specified in the report design.

- The group key expression for a group at or above the current grouping level.

- The numeric index of a group level. 0 indicates the overall totals, 1 indicates the top-most group, and so on.

- A relative group index: -1 means one group above this one, -2 means two groups above this one, and so on.

- The special name "Overall" which refers to the overall totals for the data set. The static property Total.OVERALL can also be used.

Examples:

```
Total.sum( row.myCol, null, null ); // Current group

Total.sum( row.myCol, null, "Customer Group" ); // By name

Total.sum( row.myCol, null, "row.custID" ); // By group key

Total.sum( row.myCol, null, 1 ); // By group index

Total.sum( row.myCol, null, "Overall" ); // Grand totals

Total.sum( row.myCol, null, 0 ); // Grand totals
```

Note that the filter argument must be provided when using the grouping argument. If no filter is needed, provide `null` as the value of the filter.

Note that the group index is not quoted, it must be given as a number. That is, 1 means the group at level 1, while "1" means the group named "1".

8.5 Custom Aggregates and Running Totals

BIRT provides powerful aggregation features, but there may be times when the report needs a specialized calculations beyond what BIRT offers. In this case, the developer simply implements the total using the on-start, on-row and on-finish scripts for a list, table or group. See the documentation for those items for more on these scripts.

8.6 Total Class

Synopsis

Constructor

The application cannot create instances of this class; the `Total` class exists simply as a holder for the aggregate functions.

Static Properties

OVERALL

A constant used for the group argument that requests the overall total for the data set.

Methods

sum(*expr* [, *filter* [, *group*]])

The sum of a sequence of numbers.

runningSum(*expr* [, *filter* [, *group*]])

Similar to sum, but shows the running total to a given point in the report.

count([*filter* [, *group*]])

The count of rows.

countDistinct(*expr* [, *filter* [, *group*]])

The count of distinct values.

max(*expr* [, *filter* [, *group*]])

The maximum value of a sequence of numbers, strings or dates.

min(*expr* [, *filter* [, *group*]])

The minimum value of a sequence of numbers, strings or dates.

ave(*expr* [, *filter* [, *group*]])

The average value of a sequence of numbers or dates.

weightedAve(*expr*, *weight* [, *filter* [, *group*]])

The weighted average of a sequence of numbers.

movingAve(*expr*, *window* [, *filter* [, *group*]])

A running average of a sequence of numbers in which the user specifies the number of values to consider when computing the average.

median(*expr* [, *filter* [, *group*]])

The mathematical median of a sequence of numbers.

mode(*expr* [, *filter* [, *group*]])

The mathematical mode of a sequence of values.

stdDev(*expr* [, *filter* [, *group*]])

The mathematical standard deviation of a sequence of numbers.

variance(*expr* [, *filter* [, *group*]])

The mathematical variance of a sequence of numbers.

```
first( expr [, filter [, group ]] )
```

The first value in a sequence.

```
last( expr [, filter [, group ]] )
```

The last value in a sequence.

```
irr( expr, startingGuess [, filter [, group ]])
```

The internal rate of return for a series of periodic cash flows

```
mirr( expr, financeRate, reinvestmentRate[, filter [, group ]] )
```

The modified internal rate of return for a series of periodic cash flows

```
npv( expr, rate, [, filter [, group ]] )
```

The net present value of a varying series of periodic cash flows.

```
runningNpv( expr, rate, [, filter [, group ]] )
```

The running net present value of a varying series of periodic cash flows.

Description

General rules:

- Aggregates ignore null values
- The application can optionally specify a grouping level.
- The application can optionally specify a condition.

8.7 `Total.sum` Aggregate

Sums of a sequence of numbers.

Synopsis

```
Total.sum( expr [, filter [, group ]] )
```

Arguments

```
expr
```

The expression to sum. The expression should reference at least one data row column. See additional description above. The result must be a number.

```
filter
```

An optional filter condition. See description above.

```
group
```

An optional group identifier. See description above.

Returns

The sum of the given expression. Returns zero if no rows were available.

Description

Computes the sum resulting adding up a value for each row in the group. The value for each row is computed using the expression given in the expr argument. The sum is obtained by adding all these values together.

Example

The following totals order amounts for a customer:

```
Total.sum( row.OrderAmt )
```

See Also

`Total.count` aggregate

`Total.runningSum` aggregate

8.8 `Total.sum` Aggregate

Sums a sequence of numbers to the current point in the report.

Synopsis

`runningSum( expr [, filter [, group ]] )`

Arguments

`expr`

The expression to sum. The expression should reference at least one data row column. See additional description above. The result must be a number.

`filter`

An optional filter condition. See description above.

`group`

An optional group identifier. See description above.

Returns

The running total of the given expression.

Description

Computes the sum resulting adding up a value for each row in the group. The value for each row is computed using the expression given in the `expr` argument. The sum is obtained by adding all these values together.

This aggregate differs from sum in how the total is calculated. Sum returns the total for the current grouping level. RunningSum provides the total so far within the group level.

Example

The following returns the running total for order amounts for a customer:

```
Total.runningSum( row.OrderAmt )
```

See Also

`Total.count` aggregate

`Total.sum` aggregate

8.9 `Total.count` Aggregate

Counts the rows.

Synopsis

```
Total.count( [ filter [, group ]] )
```

Arguments

```
filter
```

An optional filter condition. See description above.

```
group
```

An optional group identifier. See description above.

Returns

The number of matching rows within the given group. Returns zero if no rows were available.

Description

This function counts the number of rows within the group.

Example

To count the number of male and female students in a class, create two data items. Set the first to:

```
Total.sum( row.sex == 'M' );
```

Set the second to:

```
Total.sum( row.sex == 'F' );
```

See Also

`Total.countDistinct` aggregate

8.10 `Total.countDistinct` Aggregate

Computes the count of distinct values within a group.

Synopsis

```
Total.countDistinct ( expr [, filter [, group ]] )
```

Arguments

```
expr
```

The expression that identifies the unique values. The expression should reference at least one data row column. See additional description above. The data type can be number, string or date.

```
filter
```

An optional filter condition. See description above.

```
group
```

An optional group identifier. See description above.

Returns

The number of distinct values within the group or data set. Returns zero if no rows were available.

Description

Computes the number of distinct values within the group or data set. The expr argument gives an expression used to group the values. The expression refers to a data row column. Null values are counted as one distinct value.

Example

Suppose we want to know the number of different countries represented by a group of students. We can define a data item that uses the following expression:

```
Total.countDistinct( row.Country )
```

Where `row.Country` is a column that contains the name (or code) for the student's home country. Suppose that some rows contain null, meaning that we don't know the home country. We can exclude such rows from our count:

```
Total.countDistinct( row.Country, row.Country != null )
```

See Also

`Total.count` aggregate

8.11 `Total.max` Aggregate

Computes the maximum value of a sequence of numbers, strings or dates.

Synopsis

```
Total.max( expr [, filter [, group ]] )
```

Arguments

`expr`

The expression to maximize. The expression should reference at least one data row column. See additional description above. The data type can be number, string or date.

`filter`

An optional filter condition. See description above.

`group`

An optional group identifier. See description above.

Returns

The maximum value of the given expression. Returns `null` if no rows were available.

Description

Computes the maximum value of the given expression. The expression is evaluated for each row, and the maximum value is retained. This function can work with any simple type: number, date or string.

Example

To find the oldest student within a class:

```
Total.max( row.Age )
```

See Also

`Total.min` aggregate

`Total.first` aggregate

`Total.last` aggregate

8.12 `Total.min` Aggregate

Computes the minimum value of a sequence of numbers, strings or dates.

`Total.min( expr [, filter [, group ]] )`

Arguments

`expr`

The expression to minimize. The expression should reference at least one data row column. See additional description above. The data type can be number, string or date.

`filter`

An optional filter condition. See description above.

`group`

An optional group identifier. See description above.

Returns

The minimum value of the given expression. Returns `null` if no rows were available.

Description

Computes the minimum value of the given expression. The expression is evaluated for each row, and the minimum value is retained. This function can work with any simple type: number, date or string.

Example

To find the youngest student within a class:

    Total.min(row.Age)

See Also

`Total.max` aggregate

`Total.first` aggregate

`Total.last` aggregate

8.13 `Total.ave` Aggregate

Computes the average value of a sequence of numbers or dates.

`Total.ave( expr [, filter [, group ]] )`

Arguments

`expr`

The expression to average. The expression should reference at least one data row column. See additional description above. The data type can be number or date.

```
filter
```

An optional filter condition. See description above.

```
group
```

An optional group identifier. See description above.

Returns

The average value of the given expression. Returns `null` if no rows were available.

Description

Computes the mathematical mean value. If the expression evaluates to a number, then this function returns the average of those numbers. If the expression evaluates to a date, then the function returns the average date.

Example

To return the average age of students in a class:

```
Total.ave( row.Age )
```

And, to return the average birthday of students in a class:

```
Total.ave( row.BirthDate )
```

See Also

`Total.weightedAve` aggregate

`Total.movingAve` aggregate

`Total.median` aggregate

`Total.mode` aggregate

`Total.stdDev` aggregate

`Total.variance` aggregate

8.14 `Total.weightedAve` Aggregate

The weighted average of a sequence of numbers.

Synopsis

```
Total.weightedAve( expr, weight [, filter [, group ]] )
```

Arguments

```
expr
```

The expression to average. The expression should reference at least one data row column. See additional description above. The result must be a number.

```
weight
```

An expression giving the weight of each row. The result must be a number.

```
filter
```

An optional filter condition. See description above.

```
group
```

An optional group identifier. See description above.

Returns

The weighted average value of the given expression. Returns `null` if no rows were available.

Description

Computes the mathematical weighted mean value. If either the expr or weight arguments evaluate to null, then the row is excluded from the average.

Example

Suppose that a finance application tracks batches of a given stock purchased at different times. Each batch has a different purchase price, and a number of shares purchased at that price. The following computes the weighted average purchase price:

```
Total.weighted( row.purchasePrice, row.shareCount )
```

See Also

`Total.ave` aggregate

`Total.movingAve` aggregate

`Total.median` aggregate

`Total.mode` aggregate

`Total.stdDev` aggregate

`Total.variance` aggregate

8.15 `Total.movingAve` Aggregate

A running average of a sequence of numbers in which the user specifies the number of values to consider when computing the average.

Synopsis

```
Total.movingAve( expr, window [, filter [, group ]] )
```

Arguments

`expr`

The expression to average. The expression should reference at least one data row column. See additional description above. The result must be a number.

`window`

The number of rows to consider when computing the aggregate. Must evaluate to a number. Evaluated once when the aggregate starts.

`filter`

An optional filter condition. See description above.

`group`

An optional group identifier. See description above.

Returns

The moving average value of the given expression. Returns `null` if no rows were available.

Description

Computes a moving average. The expr argument gives the value to average, and the window argument gives the number of rows to consider. The rows are averaged in the order determined by the context, usually the order specified by a sort for a List or Table element.

Example

Suppose a report lists the daily price for a stock. Suppose the report wants to display the moving average of that stock over the last five days:

```
Total.movingAve( row.price, 5 );
```

See Also

`Total.ave` aggregate

`Total.weightedAve` aggregate

`Total.median` aggregate

`Total.mode` aggregate

`Total.stdDev` aggregate

`Total.variance` aggregate

8.16 `Total.median` Aggregate

Computes mathematical median of a sequence of numbers.

```
Total.median( expr [, filter [, group ]] )
```

Arguments

`expr`

The expression to average. The expression should reference at least one data row column. See additional description above. The data type can be number or date.

`filter`

An optional filter condition. See description above.

`group`

An optional group identifier. See description above.

Returns

The median value of the given expression. Returns `null` if no rows were available.

Description

Computes the mathematical median value. The median is selected so that half the values fall above the median, and half below.

Example

To return the median age of students in a class:

```
Total.median( row.Age )
```

And, to return the median birthday of students in a class:

```
Total.median( row.BirthDate )
```

See Also

`Total.ave` aggregate

`Total.weightedAve` aggregate

`Total.movingAve` aggregate

`Total.mode` aggregate

`Total.stdDev` aggregate

`Total.variance` aggregate

8.17 `Total.mode` Aggregate

The mathematical mode of a sequence of values.

Synopsis

`Total.mode( expr [, filter [, group ]] )`

Arguments

`expr`

>The expression to average. The expression should reference at least one data row column. See additional description above. The data type can be number, date or string.

`filter`

>An optional filter condition. See description above.

`group`

>An optional group identifier. See description above.

Returns

The mode value of the given expression. Returns `null` if no rows were available, or if the data has more than one mode.

Description

Computes the mathematical mode value. The mode is the value that occurs most frequently in the data. For example, in the sequence {1, 2, 3, 2, 4, 7}, 2 is the mode because it appears twice, while all other numbers appear only once. A data set may have multiple modes: {1, 2, 3, 2, 3, 4}. In this case, 2 and 3 both appear twice while the other numbers appear once. The mode aggregate returns null in this case.

Example

To return the mode (most frequently occurring) age of students in a class:

```
Total.mode( row.Age )
```

See Also

`Total.ave` aggregate

`Total.weightedAve` aggregate

`Total.movingAve` aggregate

`Total.median` aggregate

`Total.stdDev` aggregate

`Total.variance` aggregate

8.18 `Total.stdDev` Aggregate

Computes the mathematical standard deviation of a sequence of numbers.

Synopsis

`Total.stdDev( ` *expr* ` [, ` *filter* ` [, ` *group* ` ]] )`

Arguments

`expr`

> The expression to analyze. The expression should reference at least one data row column. See additional description above. The result must be a number.

`filter`

> An optional filter condition. See description above.

`group`

> An optional group identifier. See description above.

Returns

The standard deviation of the given expression. Returns `null` if no rows were available.

Description

This aggregate computes the statistical standard deviation of a sequence of numbers. The standard deviation is a measure of the spread of a set of values.

Example

See Also

`Total.ave` aggregate

`Total.weightedAve` aggregate

`Total.movingAve` aggregate

`Total.median` aggregate

`Total.mode` aggregate

`Total.variance` aggregate

8.19 `Total.variance` Aggregate

The mathematical variance of a sequence of numbers.

Synopsis

`Total.variance( ` *expr* ` [, ` *filter* ` [, ` *group* ` ]] )`

Arguments

`expr`

> The expression to analyze. The expression should reference at least one data row column. See additional description above. The result must be a number.

```
filter
```

An optional filter condition. See description above.

```
group
```

An optional group identifier. See description above.

Returns

The variance of the given expression. Returns `null` if no rows were available.

Description

This aggregate computes the statistical variance of a sequence of numbers. The variance is a measure of the spread of a set of values.

Example

See Also

`Total.ave` aggregate

`Total.weightedAve` aggregate

`Total.movingAve` aggregate

`Total.median` aggregate

`Total.mode` aggregate

`Total.stdDev` aggregate

8.20 `Total.first` Aggregate

Computes the first value in a sequence.

Synopsis

```
Total.first( expr [, filter [, group ]] )
```

Arguments

```
expr
```

The expression to evaluate. The expression should reference at least one data row column. See additional description above. The data type can be number, date or string.

```
filter
```

An optional filter condition. See description above.

```
group
```

An optional group identifier. See description above.

Returns

The first value that appears in the sequence of rows, or null if the data set contains no rows.

Description

Returns the first value that appears in a data set. This is the first value fetched from the data set when fetching rows using the sort order defined for the Table or List.

Example

Suppose that a report lists transactions for a given stock over a period of time. The following displays the earliest purchase of the stock:

```
Total.first( row.TransDate, row.Action == 'Buy' );
```

See Also

`Total.last` aggregate

`Total.max` aggregate

`Total.min` aggregate

8.21 `Total.last` Aggregate

Computes the last value in a sequence.

Synopsis

```
Total.last( expr [, filter [, group ]] )
```

Arguments

`expr`

> The expression to evaluate. The expression should reference at least one data row column. See additional description above. The data type can be number, date or string.

`filter`

> An optional filter condition. See description above.

`group`

> An optional group identifier. See description above.

Returns

The last value that appears in the sequence of rows, or null if the data set contains no rows.

Description

Returns the last value that appears in a data set. This is the last value fetched from the data set when fetching rows using the sort order defined for the Table or List.

Example

Suppose that a report lists transactions for a given stock over a period of time. The following displays the most recent sale of the stock:

```
Total.last( row.TransDate, row.Action == 'Sell' );
```

See Also

`Total.first` aggregate

`Total.max` aggregate

`Total.min` aggregate

8.22 `Total.irr` Aggregate

Computes the internal rate of return for a series of periodic cash flows. See Section on Finace.irr function for description of arguments and return values.

Synopsis

```
Total.irr( expr, startingGuess [, filter [, group ]])
```

See Also

Finance.irr Function

8.23 `Total.mirr` Aggregate

Computes the modified internal rate of return for a series of periodic cash flows. See Section on Finace.mirr function for description of arguments and return values.

Synopsis

```
Total.mirr( expr, financeRate, reinvestmentRate[, filter [, group ]] )
```

See Also

Finance.mirr Function

8.24 `Total.npv` Aggregate

Computes the net present value of a varying series of periodic cash. See Section on Finace.npv function for description of arguments and return values.

Synopsis

```
Total.npv( expr, rate, [, filter [, group ]] )
```

See Also

Finance.npv Function

8.25 `Total.runningNpv` Aggregate

Computes the running net present value of a varying series of periodic cash. See Section on Finace.npv function for description of arguments and return values.

Synopsis

```
Total. runningnpv( expr, rate, [, filter [, group ]] )
```

See Also

Finace.npv Function

Total.npv Function

9. The Finance Class

BIRT provides a class that provides a set of static methods that provide a wide range of financial functions.

9.1 `Finance` Class

Provides a set of static financial functions.

Constructor

The application cannot create an instance of this class.

Static Methods

The BIRT-provided Finance class provides the following functions:

ddb(*initialCost*, *salvageValue*, *assetLifespan*, *singlePeriod*)

The depreciation of an asset for a given, single period using the double-declining balance method.

sln(*initialCost*, *salvageValue*, *assetLifespan*)

Straight-line depreciation of an asset for a single period.

syd(*initialCost*, *salvageValue*, *assetLifespan*, *singlePeriod*)

Sum-of-years'-digits depreciation of an asset for a specified period.

fv(*ratePerPeriod*,*numberPayPeriods*, *eachPmt*, *presentValue*, *whenDue*)

Future value of an annuity based on periodic, constant payments, and on an unvarying interest rate.

ipmt(*ratePerPeriod*, *singlePeriod*, *numberPayPeriods*, *presentValue*, *futureValue*, *whenDue*)

interest payment for a given period of an annuity, based on periodic, constant payments, and on an unvarying interest rate.

nper(*ratePerPeriod*, *eachPmt*, *presentValue*, *futureValue*, *whenDue*)

Number of periods for an annuity based on periodic, constant payments, and on an unvarying interest rate.

pmt(*ratePerPeriod*, *numberPayPeriods*, *presentValue*, *futureValue*, *whenDue*)

Payment for an annuity, based on periodic, constant payments, and on an unvarying interest rate.

ppmt(*ratePerPeriod*, *singlePeriod*, *numberPayPeriods*, *presentValue*, *futureValue*, *whenDue*)

Principal payment for a given period of an annuity, based on periodic, constant payments, and on an unvarying interest rate.

`pv( ` *ratePerPeriod*`, ` *numberPayPeriods*`, ` *eachPmt*`, ` *futureValue*`, ` *whenDue* ` )`

> Present value of an annuity based on periodic, constant payments to be paid in the future, and on an unvarying interest rate.

`rate( ` *numberPayPeriods*`, ` *eachPmt*`, ` *presentValue*`, ` *futureValue*`, ` *whenDue*`,`
    startingGuess ` )`

> Interest rate per period for an annuity.

`irr( ` *cashArray*`, ` *startingGuess* ` )`

> Internal rate of return for a series of periodic cash flows, payments and receipts, in an existing array.

`npv( ` *rate*`, ` `cashArray` ` )`

> The net present value of a varying series of periodic cash flows, both positive and negative, at a given interest rate.

`mirr( ` *cashArray*`, ` *financeRate*`, ` *reinvestmentRate* ` )`

> The modified internal rate of return for a series of periodic cash flows (payments and receipts) in an existing array.

`percent( ` *denom*`, ` *num*`, ` *valueIfZero* ` )`

> Computes the percentage of two numbers.

Description

Financial values can be represented as either a float or decimal value. (Only float values will be supported in the first release.)

This class is simply a container for the financial functions; the application cannot create instances of this class.

9.2 `Finance.ddb` Function

Returns the depreciation of an asset for a given, single period using the double-declining balance method.

Synopsis

`ddb( ` *initialCost*`, ` *salvageValue*`, ` *assetLifespan*`, ` *singlePeriod* ` )`

Arguments

`initialCost`

> Numeric expression that specifies the initial cost of the asset.

`salvageValue`

> Numeric expression that specifies the value of the asset at the end of its useful life.

`assetLifespan`

> Numeric expression that specifies the length of the useful life of the asset.

> Rule: Must be given in the same units of measure as `singlePeriod`. For example, if `singlePeriod` represents a month, then `assetLifespan` must be expressed in months.

`singlePeriod`

Numeric expression that specifies the period for which you want DDB to calculate the depreciation.

Rule: Must be given in the same units of measure as `assetLifespan`. For example, if `assetLifespan` is expressed in months, then `singlePeriod` must represent a period of one month.

Description

Double-declining balance depreciation is an accelerated method of depreciation that results in higher depreciation charges and greater tax savings in the earlier years of the useful life of a fixed asset than are given by the straight-line depreciation method (SLN), where charges are uniform throughout.

The method uses the following formula:

Depreciation over `singlePeriod` = ((`initialCost` - total depreciation from prior periods) * 2) / `assetLifespan`.

Rules:

- `assetLifespan` and `singlePeriod` must both be expressed in terms of the same units of time.

- All parameters must be positive numbers.

Example

The following example calculates the depreciation for the first year under the double-declining balance method for a new machine purchased at $1400, with a salvage value of $200, and a useful life estimated at 10 years. The result ($280) is assigned to the variable Year1Deprec:

```
Year1Deprec = Finance.ddb(1400, 200, 10, 1)
```

See Also

`Finance.sln` function

`Finance.syd` function

9.3 `Finance.sln` Function

Returns the straight-line depreciation of an asset for a single period.

Synopsis

`sln( initialCost, salvageValue, assetLifespan )`

Arguments

`initialCost`

Numeric expression that specifies the initial cost of the asset.

`salvageValue`

Numeric expression that specifies the value of the asset at the end of its useful life. You can type a salvage value to view the straight line depreciation offset by the salvage value, or return straight line depreciation without salvage value by entering 0 in salvage value.

`assetLifespan`

Numeric expression that specifies the length of the useful life of the asset.

Rule: Must be given in the same units of measure you want the function to return. For example, if you want SLN to determine the annual depreciation of the asset, `assetLifespan` must be given in years.

Description

Straight-line depreciation is the oldest and simplest method of depreciating a fixed asset. It uses the book value of the asset less its estimated residual value, and allocates the difference equally to each period of the asset's life. Such procedures are used to arrive at a uniform annual depreciation expense that is charged against income before calculating income taxes.

All arguments must be positive numbers.

Example

The following example calculates the depreciation under the straight-line method for a new machine purchased at $1400, with a salvage value of $200, and a useful life estimated at 10 years. The result ($120 annually) is assigned to AnnualDeprec:

```
AnnualDeprec = Finance.sln(1400, 200, 10)
```

See Also

`Finance.ddb` function

`Finance.syd` function

9.4 `Finance.syd` Function

Returns sum-of-years'-digits depreciation of an asset for a specified period.

Synopsis

`syd( initialCost, salvageValue, assetLifespan, singlePeriod )`

Arguments

`initialCost`

Numeric expression that specifies the initial cost of the asset.

`salvageValue`

Numeric expression that specifies the value of the asset at the end of its useful life.

`assetLifespan`

Numeric expression that specifies the length of the useful life of the asset.

Rule: Must be given in the same units of measure as `singlePeriod`. For example, if `singlePeriod` represents a month, then `assetLifespan` must be expressed in months.

`singlePeriod`

Numeric expression that specifies the period for which you want syd to calculate the depreciation.

Rule: Must be given in the same units of measure as `assetLifespan`. For example, if `assetLifespan` is expressed in months, then `singlePeriod` must represent a period of one month.

Description

Sum-of-years'-digits is an accelerated method of depreciation that results in higher depreciation charges and greater tax savings in the earlier years of the useful life of a fixed asset than are given by the straight-line depreciation method (SLN), where charges are uniform throughout.

The method bases depreciation on an inverted scale of the total of digits for the years of useful life. For instance, if the asset's useful life is 4 years, the digits 4, 3, 2, and 1 are added together to produce 10. SYD for the first year then becomes 4/10ths of the depreciable cost of the asset (cost less salvage value). The rate for the second year becomes 3/10ths, and so on.

Rules:

- `singlePeriod` and `singlePeriod` must both be expressed in terms of the same units of time.

- All arguments must be positive numbers.

Example

The following example calculates the depreciation for the first year under the sum-of-years'-digits method for a new machine purchased at $1400, with a salvage value of $200, and a useful life estimated at 10 years. The result, $218.18, is assigned to Year1Deprec. You may wish to note (a) that this result is equivalent to 10/55 * $1,200; (b) that 55 = 10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1; and (c) that 10 is the 1st (Year 1) term in this series of digits:

```
Year1Deprec = Finance.syd(1400, 200, 10, 1)
```

The following example calculates the depreciation of the same asset for the second year of its useful life. The result, $196.36, is assigned to Year2Deprec. You may wish to note (a) that this result is equivalent to 9/55 * $1,200; (b) that 55 = 10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1; and (c) that 9 is the 2nd (Year 2) term in this series of digits:

```
Year2Deprec = Finance.syd(1400, 200, 10, 2)
```

See Also

`Finance.ddb` function

`Finance.sln` function

9.5 `Finance.fv` Function

Returns the future value of an annuity based on periodic, constant payments, and on an unvarying interest rate.

Synopsis

fv(*ratePerPeriod*,*numberPayPeriods*, *eachPmt*, *presentValue*, *whenDue*)

Arguments

`ratePerPeriod`

Numeric expression that specifies the interest rate that accrues per period.

Rule: Must be given in the same units of measure as `numberPayPeriods`. For instance, if `numberPayPeriods` is expressed in months, then `ratePerPeriod` must be expressed as a monthly rate.

`numberPayPeriods`

Numeric expression that specifies the total number of payment periods in the annuity.

Rule: Must be given in the same units of measure as `ratePerPeriod`. For instance, if `ratePerPeriod` is expressed as a monthly rate, then `numberPayPeriods` must be expressed in months.

`eachPmt`

Numeric expression that specifies the amount of each payment.

Rule: Must be given in the same units of measure as `ratePerPeriod`. For instance, if `ratePerPeriod` is expressed in months, then `eachPmt` must be expressed as a monthly payment.

`presentValue`

Numeric expression that specifies the value today of a future payment, or stream of payments.

Example: If you put $23.94 in the bank today and leave it there for 15 years at an interest rate of 10% compounded annually, you end up with about $100. The present value of $100 is approximately $23.94.

`whenDue`

Numeric expression that specifies whether each payment is made at the beginning (1), or at the end (0) of each period.

Rule: Must be 0 or 1.

Description

An annuity is a series of cash payments, constant in value, made over a period of time. An annuity can be an investment, such as a monthly savings plan, or a loan, such as a home mortgage. The future value of an annuity is the cash balance you want after you have made your final payment.

Examples:

- You set up a savings plan with a goal of having $75,000 in 18 years to pay for your child's education. For this plan, the future value is $75,000.

- You take out a loan for $11,000. The future value is $0.00, as it is for any typical loan.

Rules:

- `ratePerPeriod`, `numberPayPeriods`, and `eachPmt` must all be expressed in terms of the same units, weekly/weeks, monthly/months, yearly/years, and so on.

- You must express cash paid out, such as deposits to savings, using negative numbers, and cash received, such as dividend checks, using positive numbers.

Example

The following example assumes you deposit $10,000 in a savings account for your daughter when she is born. If the account pays 5.7% compounded daily, how much will she have for college in 18 years? The answer, $27,896.60, is assigned to the variable TotalValue:

```
TotalValue = Finance.fv(0.057/365, 18*365, 0, -10000, 1)
```

The following example is almost the same as the previous one. In this one, however, assume that the interest is compounded monthly instead of daily, and that you have decided to make an additional monthly deposit of $55 into the account. The future value assigned to TotalValue in this case is $48,575.82:

```
TotalValue = Finance.fv(0.057/12, 18*12, -55, -10000, 1)
```

See Also

`Finance.ipmt` function

`Finance.nper` function

`Finance.pmt` function

`Finance.ppmt` function

`Finance.pv` function

`Finance.rate` function

9.6 `Finance.ipmt` Function

Returns the interest payment for a given period of an annuity, based on periodic, constant payments, and on an unvarying interest rate.

Synopsis

```
ipmt( ratePerPeriod, singlePeriod, numberPayPeriods, presentValue,
      futureValue, whenDue )
```

Arguments

`ratePerPeriod`

Numeric expression that specifies the interest rate that accrues per period.

Rule: Must be given in the same units of measure as `numberPayPeriods`. For instance, if `numberPayPeriods` is expressed in months, then `ratePerPeriod` must be expressed as a monthly rate.

`singlePeriod`

Numeric expression that specifies the particular period for which you want to determine how much of the payment for that period represents interest.

Rule: Must be in the range 1 through `numberPayPeriods`.

`numberPayPeriods`

Numeric expression that specifies the total number of payment periods in the annuity.

Rule: Must be given in the same units of measure `ratePerPeriod` For instance, if `ratePerPeriod` is expressed as a monthly rate, then `numberPayPeriods` must be expressed in months.

`presentValue`

Numeric expression that specifies the value today of a future payment or stream of payments.

Example: If you put $23.94 in the bank today and leave it there for 15 years at an interest rate of 10% compounded annually, you end up with about $100. In this case, the present value of $100 is approximately $23.94.

`futureValue`

Numeric expression that specifies the cash balance you want after you have made your final payment.

Examples:

- You set up a savings plan with a goal of having $75,000 in 18 years to pay for your child's education. For this plan, the future value is $75,000.

- You take out a loan for $11,000. The future value is $0.00, as it is for any typical loan.

whenDue

Numeric expression that specifies whether each payment is made at the beginning (1) or at the end (0) of each period.

Rule: Must be 0 or 1.

Description

An annuity is a series of cash payments, constant in value, made over a period of time. An annuity can be an investment, such as a monthly savings plan, or a loan, such as a home mortgage. Each payment consists of two components, principal and interest. IPmt returns the interest component of the payment.

Rules:

- `ratePerPeriod` and `numberPayPeriods` must be expressed in terms of the same units (weekly/weeks, monthly/months, yearly/years).

- You must express cash paid out, such as deposits to savings, using negative numbers, and cash received, such as dividend checks, using positive numbers.

Example

The following example assumes you are making monthly payments the first of each month on a loan of $20,000, over 36 months, at an APR of 11.5%. How much of your 5th payment represents interest? The answer, $171.83, is assigned to Interest5:

```
Interest5 = Finance.ipmt(.115/12, 5, 36, -20000, 0, 1)
```

See Also

`Finance.fv` function

`Finance.nper` function

`Finance.pmt` function

`Finance.ppmt` function

`Finance.pv` function

Finance.rate function

9.7 `Finance.nper` Function

Returns the number of periods for an annuity based on periodic, constant payments, and on an unvarying interest rate.

Synopsis

nper(*ratePerPeriod*, *eachPmt*, *presentValue*, *futureValue*, *whenDue*)

Arguments

ratePerPeriod

Numeric expression that specifies the interest rate that accrues per period.

Rule: Must be given in the same units of measure as eachPmt . For instance, if eachPmt is expressed as a monthly payment, then ratePerPeriod must be expressed as the monthly interest rate.

eachPmt

Numeric expression that specifies the amount of each payment.

Rule: Must be given in the same units of measure as ratePerPeriod. For instance, if ratePerPeriod is expressed in months, then eachPmt must be expressed as a monthly payment.

presentValue

Numeric expression that specifies the value today of a future payment or of a stream of payments.

Example: If you put $23.94 in the bank today and leave it there for 15 years at an interest rate of 10% compounded annually, you will end up with about $100. In this case, the present value of $100 is approximately $23.94.

futureValue

Numeric expression that specifies the cash balance you want after you have made your final payment.

Examples:

- You set up a savings plan with a goal of having $75,000 in 18 years to pay for your child's education. For this plan, the future value is $75,000.

- You take out a loan for $11,000. The future value is $0.00, as it is for any typical loan.

whenDue

Numeric expression that specifies whether each payment is made at the beginning (1), or at the end (0) of each period.

Rule: Must be 0 or 1.

Description

An annuity is a series of cash payments, constant in value, made over a period of time. An annuity can be an investment, such as a monthly savings plan, or a loan, such as a home mortgage.

Rules:

- `ratePerPeriod` and `eachPmt` must be expressed in terms of the same units (weekly/monthly/yearly, and so on).

- You must express cash paid out (such as deposits to savings) using negative numbers, and cash received (such as dividend checks) using positive numbers.

Example

The following example assumes you are making monthly payments at the first of each month on a loan of $20,000, at an APR of 11.5%. If each payment is $653.26, how many payments will you have to make to finish paying off the loan? The answer (36) is assigned to the variable NumPeriods.

```
NumPeriods = Finance.nper(.115/12, -653.26, 20000, 0, 1)
```

See Also

`Finance.fv` function

`Finance.ipmt` function

`Finance.pmt` function

`Finance.ppmt` function

`Finance.pv` function

`Finance.rate` function

9.8 `Finance.pmt` Function

Returns the payment for an annuity, based on periodic, constant payments, and on an unvarying interest rate.

Synopsis

```
pmt( ratePerPeriod, numberPayPeriods, presentValue, futureValue,
     whenDue )
```

Arguments

`ratePerPeriod`

Numeric expression that specifies the interest rate that accrues per period.

Rule: Must be given in the same units of measure as `ratePerPeriod`. For instance, if `numberPayPeriods` is expressed in months, then `ratePerPeriod` must be expressed as a monthly rate.

`numberPayPeriods`

Numeric expression that specifies the total number of payment periods in the annuity.

Rule: Must be given in the same units of measure as `ratePerPeriod`. For instance, if `ratePerPeriod` is expressed as a monthly rate, then `numberPayPeriods` must be expressed in months.

`presentValue`

Numeric expression that specifies the value in today's dollars of a future payment, or stream of payments.

Example: If you put $23.94 in the bank today and leave it there for 15 years at an interest rate of 10% compounded annually, you end up with about $100. In this case, the present value of $100 is approximately $23.94.

`futureValue`

Numeric expression that specifies the cash balance you want after you have made your final payment.

Examples:

- You set up a savings plan with a goal of having $75,000 in 18 years to pay for your child's education. For this plan, the future value is $75,000.

- You take out a loan for $11,000. The future value is $0.00, as it is for any typical loan.

`whenDue`

Numeric expression that specifies whether each payment is made at the beginning (1), or at the end (0) of each period.

Rule: Must be 0 or 1.

Description

An annuity is a series of cash payments, constant in value, made over a period of time. An annuity can be an investment, such as a monthly savings plan, or a loan, such as a home mortgage.

Rules:

- `ratePerPeriod` and `numberPayPeriods` must be expressed in terms of the same units (weekly/weeks, monthly/months, yearly/years).

- You must express cash paid out (such as deposits to savings) using negative numbers, and cash received (such as dividend checks) using positive numbers.

Example

The following example assumes you are making monthly payments the first of each month on a loan of $20,000, over 36 months, at an APR of 11.5%. How much will each of your payments be? The answer ($653.26) is assigned to PaymentAmt.

```
PaymentAmt = Finance.pmt(.115/12, 36, -20000, 0, 1)
```

See Also

`Finance.fv` function

`Finance.ipmt` function

`Finance.nper` function

`Finance.ppmt` function

`Finance.pv` function

`Finance.rate` function

9.9 `Finance.ppmt` Function

Returns the principal payment for a given period of an annuity, based on periodic, constant payments, and on an unvarying interest rate.

Synopsis

```
ppmt( ratePerPeriod, singlePeriod, numberPayPeriods, presentValue,
      futureValue, whenDue )
```

Arguments

`ratePerPeriod`

Numeric expression that specifies the interest rate that accrues per period.

Rule: Must be given in the same units of measure as `numberPayPeriods`. For instance, if `numberPayPeriods` is expressed in months, then `ratePerPeriod` must be expressed as a monthly rate.

`singlePeriod`

Numeric expression that specifies the particular period for which you want to determine how much of the payment for that period represents interest.

Rule: Must be in the range 1 through <number pay periods>.

`numberPayPeriods`

Numeric expression that specifies the total number of payment periods in the annuity.

Rule: Must be given in the same units of measure as `ratePerPeriod`. For instance, if `ratePerPeriod` is expressed as a monthly rate, then `numberPayPeriods` must be expressed in months.

`presentValue`

Numeric expression that specifies the value today of a future payment, or stream of payments.

Example: If you put $23.94 in the bank today and leave it there for 15 years at an interest rate of 10% compounded annually, you end up with about $100. In this case, the present value of $100 is approximately $23.94.

`futureValue`

Numeric expression that specifies the cash balance you want after you have made your final payment.

Examples:

- You set up a savings plan with a goal of having $75,000 in 18 years to pay for your child's education. For this plan, the future value is $75,000.

- You take out a loan for $11,000. The future value is $0.00, as it is for any typical loan.

whenDue

Numeric expression that specifies whether each payment is made at the beginning (1), or at the end (0) of each period.

Rule: Must be 0 or 1.

Description

An annuity is a series of cash payments, constant in value, made over a period of time. An annuity can be an investment, such as a monthly savings plan, or a loan, such as a home mortgage.

Each payment in an annuity consists of two components: principal and interest. PPmt returns the principal component of the payment.

Rules:

- `ratePerPeriod` and `numberPayPeriods` must be expressed in terms of the same units such as weeks, months or years.

- You must express cash paid out (such as deposits to savings) using negative numbers, and cash received (such as dividend checks) using positive numbers.

Example

The following example assumes you are making monthly payments at the first of each month on a loan of $20,000, over 36 months, at an APR of 11.5%. How much of your 5th payment represents principal? The answer ($481.43) is assigned to Principal5:

```
Principal5 = Finance.ppmt(.115/12, 5, 36, -20000, 0, 1)
```

See Also

`Finance.fv` function

`Finance.ipmt` function

`Finance.nper` function

`Finance.pmt` function

`Finance.pv` function

`Finance.rate` function

9.10 `Finance.pv` Function

Returns the present value of an annuity based on periodic, constant payments to be paid in the future, and on an unvarying interest rate.

Synopsis

pv(*ratePerPeriod*, *numberPayPeriods*, *eachPmt*, *futureValue*, *whenDue*)

Arguments

`ratePerPeriod`

Numeric expression that specifies the interest rate that accrues per period.

Rule: Must be given in the same units of measure as `numberPayPeriods`. For instance, if `numberPayPeriods` is expressed in months, then `ratePerPeriod` must be expressed as a monthly rate.

`numberPayPeriods`

Numeric expression that specifies the total number of payment periods in the annuity.

Rule: Must be given in the same units of measure as `ratePerPeriod`. For instance, if `ratePerPeriod` is expressed as a monthly rate, then `numberPayPeriods` must be expressed in months.

`eachPmt`

Numeric expression that specifies the amount of each payment.

Rule: Must be given in the same units of measure as `ratePerPeriod`. For instance, if `ratePerPeriod` is expressed in months, then `eachPmt` must be expressed as a monthly payment.

`futureValue`

Numeric expression. Specifies the cash balance you want after you have made your final payment.

Examples:

- You set up a savings plan with a goal of having $75,000 in 18 years to pay for your child's education. For this plan, the future value is $75,000.

- You take out a loan for $11,000. The future value is $0.00, as it is for any typical loan.

`whenDue`

Numeric expression that specifies whether each payment is made at the beginning (1), or at the end (0) of each period.

Rule: Must be 0 or 1.

Description

An annuity is a series of cash payments, constant in value, made over a period of time. An annuity can be an investment, such as a monthly savings plan, or a loan, such as a home mortgage. The present value is the value today of a future payment, or of a stream of payments structured as an annuity.

Example:

If you put $23.94 in the bank today and leave it there for 15 years at an interest rate of 10% compounded annually, you end up with about $100. So, the present value of $100 is approximately $23.94.

Rules

- `ratePerPeriod` and `numberPayPeriods` must be expressed in terms of the same units (if weekly/then weeks, monthly/months, yearly/years, and so on).

- You must express cash paid out (such as deposits to savings) using negative numbers, and cash received (such as dividend checks) using positive numbers.

Example

The following example assumes you are considering the purchase of a corporate bond with a $1000 face value. The bond pays an annual coupon of $100, matures in 15 years, and the next coupon is paid at the end of one year. The yield to maturity on similar bonds is 12.5%. What is a fair price for this bond (its present value)? The answer, $834.18, is assigned to the variable PresentValue:

```
PresentValue = Finance.pv(.125, 15, 100, 1000, 0)
```

The following examples assumes you have won the lottery. The jackpot is $10 million, which you receive in yearly installments of $500,000 per year for 20 years, beginning one year from today. If the interest rate is 9.5% compounded annually, how much is the lottery worth today? The answer, $4,406,191.06, is assigned to PresentValue:

```
PresentValue = Finance.pv(.095, 20, 50000,10000000, 0)
```

The following example assumes you want to save $11,000 over the course of 3 years. If the APR is 10.5% and you plan to save $325 monthly, and if you make your payments at the beginning of each month, how much would you need to start off with in your account to achieve your goal? The answer, $2,048.06, is assigned to StartValue. Note that <each pmt> is expressed as a negative number because it represents cash paid out:

```
StartValue = Finance.pv(.105/12, 3*12, -325, 11000, 1)
```

See Also

`Finance.fv` function

`Finance.ipmt` function

`Finance.nper` function

`Finance.pmt` function

`Finance.ppmt` function

`Finance.rate` function

9.11 `Finance.rate` Function

Returns the interest rate per period for an annuity.

Synopsis

```
rate( numberPayPeriods, eachPmt, presentValue, futureValue, whenDue,
    startingGuess )
```

Arguments

`numberPayPeriods`

Numeric expression that specifies the total number of payment periods in the annuity.

Rule: Must be given in the same units of measure as `eachPmt`. For instance, if `eachPmt` is expressed as a monthly payment, then `numberPayPeriods` must be expressed in months.

`eachPmt`

Numeric expression that specifies the amount of each payment.

Rule: Must be given in the same units of measure as `numberPayPeriods`. For instance, if `numberPayPeriods` is expressed in months, then `eachPmt` must be expressed as a monthly payment.

`presentValue`

Numeric expression that specifies the value today of a future payment, or of a stream of payments.

Example: If you put $23.94 in the bank today and leave it there for 15 years at an interest rate of 10% compounded annually, you will end up with about $100. So in this case, the present value of $100 is approximately $23.94.

`futureValue`

Numeric expression that specifies the cash balance you want after you have made your final payment.

Examples:

- You set up a savings plan with a goal of having $75,000 in 18 years to pay for your child's education. For this plan, the future value is $75,000.

- You take out a loan for $11,000. The future value is $0.00, as it is for any typical loan.

`whenDue`

Numeric expression that specifies whether each payment is made at the beginning (1), or at the end (0) of each period.

Rule: Must be 0 or 1.

`startingGuess`

Numeric expression that specifies the value you estimate Rate will return. In most cases, this is 0.1 (10 percent).

Description

An annuity is a series of cash payments, constant in value, made over a period of time. An annuity can be an investment, such as a monthly savings plan, or a loan, such as a home mortgage.

Rate calculates the interest rate on an annuity iteratively. Starting with the value of `startingGuess`, it repeats the calculation until the result is accurate to within 0.00001 percent. If it cannot determine a result after 20 iterations, the function fails.

Rules:

- `numberPayPeriods`, and `eachPmt` must be expressed in terms of the same units (weekly/weeks, monthly/months, yearly/years, and so on).

- You must express cash paid out, such as deposits to savings, using negative numbers and cash received, such as dividend checks, using positive numbers.

Tips:

- Because Rate uses the order of values within the array to interpret the order of payments and receipts, be sure to enter your payment and receipt values in the correct sequence.

- If Rate fails, try a different value `startingGuess`.

Example

The following example assumes you have taken out a loan for $20,000, that you are paying off over the course of 3 years. If your payments are $653.26 per month, and you make them at the beginning of each month, what interest rate (APR) are you paying? The answer, .115 or 11.5%, is assigned to the variable InterestRate. Note that the return value of Rate must be multiplied by 12 to yield an annual rate:

```
     InterestRate = Finance.rate(3*12, -653.26, 20000, 0, 1, .1) * 12
```

See Also

`Finance.fv` function

`Finance.ipmt` function

`Finance.nper` function

`Finance.pmt` function

`Finance.ppmt` function

`Finance.pv` function

9.12 `Finance.irr` Function

Returns the internal rate of return for a series of periodic cash flows, payments and receipts, in an existing array.

Synopsis

`irr( `*`cashArray`*`, `*`startingGuess`*` )`

Arguments

`cashArray`

Specifies the name of an existing array of Doubles representing cash flow values.

Rule: Array must contain at least one positive value (receipt) and one negative value (payment).

`startingGuess`

Numeric expression. Specifies the value you estimate IRR will return. In most cases, this is 0.1 (10 percent).

Description

The internal rate of return is the interest rate for an investment consisting of payments and receipts that occur at regular intervals. The cash flow for each period does not need to be constant, as it does for an annuity.

IRR is closely related to the net present value function, NPV, because the rate of return calculated by IRR is the interest rate corresponding to a net present value of zero. IRR calculates by iteration. Starting with the value of <starting guess>, it repeats the calculation until the result is accurate to within 0.00001 percent. If it cannot determine a result after 20 iterations, the function fails.

Rules:

- You must express cash paid out, such as deposits to savings, using negative numbers, and cash received, such as dividend checks, using positive numbers.

- `cashArray` must contain at least one negative and one positive number.

- In cases where you have both a positive cash flow (income) and a negative one (payment) for the same period, use the net flow for that period.

- If no cash flow or net cash flow occurs for a particular period, you must type 0 (zero) as the value for that period.

Tips:

- Because IRR uses the order of values within the array to interpret the order of payments and receipts, be sure to type your payment and receipt values in the correct sequence.

- If IRR fails, try a different value for `startingGuess`.

Example

The following example assumes you have filled the array `myArray` with a series of cash flow values. The internal rate of return is assigned to the variable IRRValue:

```
IRRValue = Finance.irr( myArray, .1 )
```

See Also

`Finance.mirr` function

`Finance.npv` function

`Finance.rate` function

9.13 `Finance.npv` Function

Returns the net present value of a varying series of periodic cash flows, both positive and negative, at a given interest rate.

Synopsis

```
npv( rate, cashArray )
```

Arguments

`rate`

Numeric expression that specifies the discount rate over the length of the period.

Rule: Must be expressed as a decimal.

`cashArray`

Array of doubles that specifies the name of an existing array of cash flow values.

Rule: Array must contain at least one positive value (receipt) and one negative value (payment).

Description

While PV determines the present value of a series of constant payments, NPV does the same for a series of varying payments. Net present value is the value in today's dollars of all future cash flows associated with an investment minus any initial cost. In other words, it is that lump sum of money that would return the same profit or loss as the series of cash flows in question, if the lump sum were deposited in a bank today and left untouched to accrue interest at the rate given by <rate> for the same period of time contemplated by the cash flow stream.

Rules:

- The NPV investment begins one period before the date of the first cash flow value and ends with the last cash flow value in the array.