ReportMill
SOFTWARE

Key Chains

Table of Contents

Key Chains (Continued)

Basic Keys

The most important part of any reporting system is the data substitution. ReportMill uses a familiar mail-merge paradigm, allowing the user to intermingle keys and static text in any textfield. Keys are delineated by "@" symbols, for example: @Date@ or @Page@. An example of mixed keys and text, might be "@Page@ of @PageMax@", perhaps resulting in the text "1 of 10".

The interesting thing about keys in ReportMill is that they can be @anything@! You can type any string between two "@" symbols and ReportMill will treat it as a key. At run-time, ReportMill will try to evaluate the keys in your template using the objects you supply. ReportMill will match template keys with any public fields or methods in your objects with absolutely no wiring (it does this via Java's reflection API - java.lang.reflect). The syntax for keys follows the rules of Java expressions - almost any basic Java expression should work. Referenced method names or JDBC column names should match exactly (keys are also case-sensitive).

If a given key cannot be evaluated by either ReportMill or your supplied objects, it will simply result in the "Default Substitution String" (set to "<N/A>" by default in the document inspector).

Built-in Keys

ReportMill has a number of keys that are built-in. They may be typed in, but you can also simply drag them in from the Keys Browser.

Date - The current date/time
Row - The current row number (only in tables).
Page - The current page
PageMax - The total number of pages in the generated report
PageBreak - The number of explicit page breaks encountered
PageBreakMax - The total number of explicit page breaks in generated report
PageBreakPage - The number of pages since last explicit page break
PageBreakPageMax - The total number of pages in current explicit page break

KeyChains

In addition to simple keys, ReportMill supports key chains or key paths, which uses standard "." (dot) notation to indicate a reference to a child member. This is important when traversing a rich object graph of entities (as opposed to just core types). For instance, "@getContact.getAddress.getStreet@".

Array Indexing

Additionally, you can referece an individual object in a list using standard array indexing syntax (brackets) like this: "@getContacts[0].getName@".

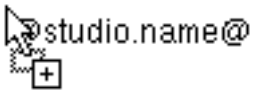
Key Chains (Continued)

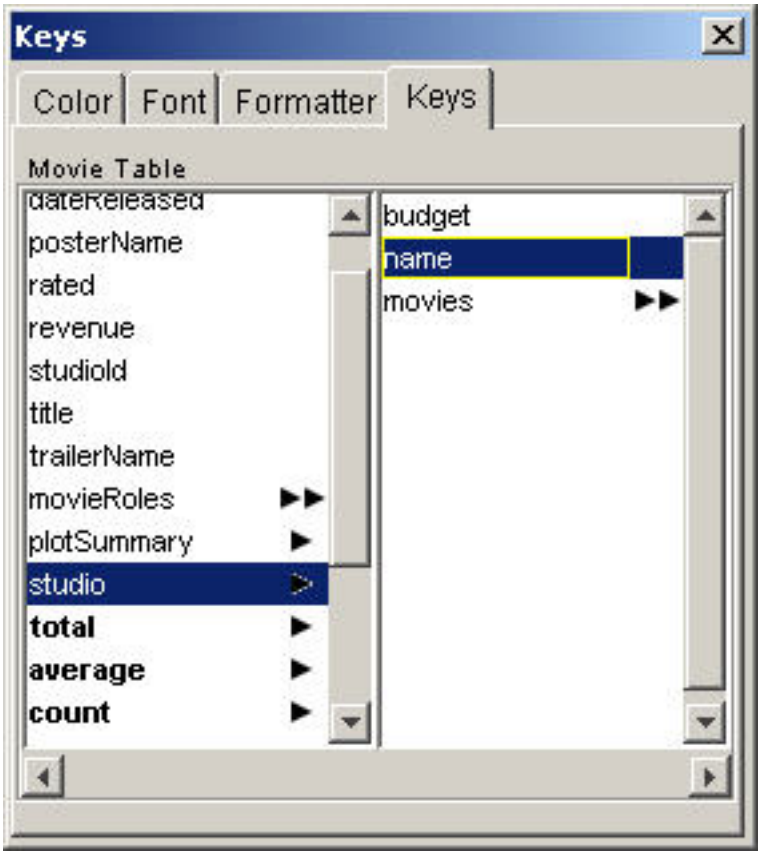
Keys Browser

ReportMill also has a Keys Browser, which is the fourth of the four tabs in the Attributes Panel (choose the menu Tools->Keys Panel to bring it up).

This panel is provided as a convenience (although a very powerful one!). It shows the most relevant keys for the selected shape. You can easily navigate through the browser then click and drag any key at any level to create a Key Chain. In creating a Movie report where the Movie object has a reference to it's studio object, you might drag the key @studio.name@ into your movie table.

Once again it's important to note that since keys can be typed in, the Keys Browser is merely a convenience. This is particularly useful when editing a template not bound to an XML dataset (or bound to an incomplete one).





Key Chains (Continued)

Key Chain Expressions

Keys in ReportMill follow the same basic convention as Java expressions. Here are the list of operators in their order precedence:

Parenthesis	(expr)	Nested expressions
Multiplicative	*, /, %	Mulitply, divide, modulo
Additive	*, -	Add, subtract
Relational	>, <, >=, <=	Greater-than, less-than, greater/less-than-equal
Equality	==, !=	Equal, not-equal
Logical AND	&&	
Logical OR	||	
Conditional	? :	If/then - with form "expr? true_expr : false_expr"
Assignments	=, +=	

Parenthesis
Any portion of a Key Chain can be enclosed with parenthesis to guarantee precedence.

Multiplicative / Additive
These are the most common and intuitive operators. You might want to display @quantity*price@ in an invoice line-item or calculate a percent like this @profit/revenue*100@.

Relational, Equality
These are most useful for conditionals: @amount>=0? "Credit" : "Debit"@ or @name=="this"? "that" : name@.

Logical AND/OR
These operators make it possible to test multiple conditions: @revenue>100 && budget<50? "Winner!"@ or @name=="Jack" || name=="Sam"? "Good Name!"@.

Conditional
This operator is one of the most useful - although you should always consider trying to implement conditionals in your object model (in your actual Java classes) so any real business logic can be re-used in all reports, JSP pages, Swing GUIs, etc. However, these are often convenient, as demonstrated above. ReportMill does add a couple of twists. The first is that the false expression is optional (it will default to null). The second is that RM will evaluate 'null' as false and non-null as true, so you can provide null substitutions like this: @name? name : "(None provided)"@.

Assignments
Only for the brave, RM6 has a new feature that lets you actually create temporary variables for use in a report. Most of the functionality you might use this for is covered in more intuitive ways (such as the Running key, covered later), but it is possible to define a variable in a header row: @revTotal=0@ and update it in details rows @revTotal+=revenue@.

Key Chains (Continued)

Aggregates (totals, min/max, average, count)

The Keys Browser also contains a list of built-in keys at the bottom of any given list: total, average, min, max and count. This allows the user to easily specify aggregate calculations on a set of objects. Each studio object in a studio report may have a relationship to the movies for that studio. We may want to see the @movies.total.revenue@ or the @movies.min.dateReleased@ or perhaps just the @movies.count@. The convention is that the aggregate follows the name of the to-many relationship. The exception is when performing an aggregate calculation on the objects in a group (in a table), where the set of objects is implied (you may just have @total.revenue@).

The "total2" key
An aggregate calculation will result in null if any of the individual values are null (rather than return a value that is technically incorrect). You can work around this by implementing a derived method that returns a default value if the original attribute is null and aggregating using that key/method. Also, most of the aggregates contain a second version ("total2") that assume that null is equal to zero.

The "count" and "countDeep" keys

As you would imagine, the count keys simply tell us how many objects are in a given list or group. This is most commonly used for tables with one or more levels of grouping. If, for instance, you have a table of Movies grouped by their studio and you add the @count@ key to the studio details, it will resolve to the number of movies for each studio. So it might make sense to have a text field with "@studio.name@ has released @count@ movies".

Now the count key only counts the next level of grouping, so if you have multiple levels of grouping, it is occasionally useful to count all the root entities. For instance, if you have Movies grouped by their category and their studio, it might now make sense to have a text field in the top level grouping, studio details, like this: "@studio.name@ has released @countDeep@ movies in @count@ different categories".

Key Chains (Continued)

Heritage Keys (Running Totals, percentage totals)

There is an additional set of keys in the Attributes Browser which are used to access upper level groups: Up, Running, Remaining. @Up.count@ would tell us how many objects are in the current level of grouping. We configure a textfield with "Row @Row@ of @Up.count@" to get "Row 1 of 5".

By doing some simple arithmetic and using the "Up" key, we can calculate a percentage total:

    % Total: @revenue/Up.total.revenue@

The "Running" key actually references a virtual array containing all of the objects processed thus far in a lower level grouping. This is useful to easily get a running total perhaps in a ledger:

    Credit/Debit: @amount@ Current balance: @Running.total.amount@

The "Remaining" key is conceptually the same, but results in a virtual array of remaining objects.

    Credit/Debit: @amount@ Remaining Activity: @Remaining.total.amount@

Key Chains (Continued)

Formatting Keys

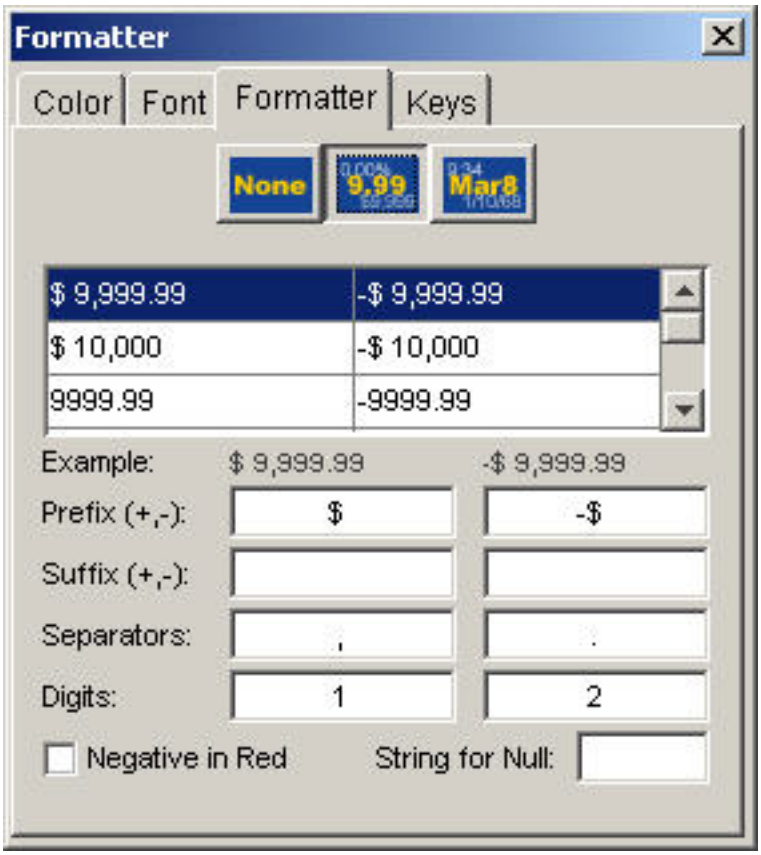
ReportMill has a Formatter Panel, which is the third of the four tabs in the Attributes Panel (choose the menu Tools->Formatter Panel to bring it up).

You can easily format a key to have currency symbols, percent signs or standard date formats by clicking the cursor anywhere between the "@" symbols in a key and bringing up the Formatter Panel (use the menu item Tools->Formatter Inspector).

In addition to the canned formats available in the pick lists, you can create custom formats by configuring the fields in the inspector.

Inspector Fields
Most of the fields in the inspector are self explanatory: prefix/suffix for positive/negative numbers (like $ or ¢) and separators for thousands (like a comma) and decimals (like a period). The final row of text fields is labeled "Digits" and allows you to set the minimum integer digits and minimum fractional digits.

String For Null Object
The inspector also lets you set the string that will appear for that key if the value is undefined (the Java "null"). This defaults to the "Default Substitution String" (in the document inspector), but this is an opportunity to customize it for a specific key.



Key Chains (Continued)

Built-in Functions

There are a number of common top-level functions you can call in any key chain, like this:

```
@floor(getAmount)@
@substring(getName,0,20)@
@rtf(getRTFString)@
```

Common functions

Math:

floor(float)
ceil(float)
round(float)
abs(float)
min(float, float)
max(float, float)
pow(float, float)

String:

substring(string, start)
substring(string, start,end)
startsWith(string,string)
endsWith(string,string)
substring(string,start)
substring(string, start, end)

Misc:

rtf(string): Returns formatted string for rtf encoded text string.
html(string): Returns formatted string for html encoded text string.
RMImage(source): Returns an image for image source (URL, File, String path, bytes, etc.).
list(key,key,key): Returns a list made up of an object for each key.

Key Chains (Continued)

Number Functions

There are a few built-in functions for any key that evaluates to a Number. These can be used just by appending the function, like this:

```
@getAmount.format("$ #,##0.00")@
```

format(string): Returns a string by formatting base number with given format string.
roman(): Returns the given number in roman numerals.

String Functions

There are a number of built-in functions for any key that evaluates to a String (beyond what is available in the Java String class). These can be used just by appending the function, like this:

```
@getName.substring(0,20)@
@getName.pad(20)@
@getName.wrap(30)@
```

substring(start): Returns the string from start index to string end.
substring(start,end): Returns the string from start index to end index.
startsWith(String): Returns whether string starts with given string (boolean).
endsWith(String): Returns whether string ends with given string (boolean).
pad(length): Returns the base string padded by spaces to given length.
pad(pad,length): Returns the base string padded by given string to given length.
padLeft(pad,length): Returns the base string pre-padded by given string to given length.
fix(length): Returns the base string at the given length, padded by spaces.
fix(length,pad): Returns the base string at the given length, padded by given pad string.
wrap(limit): Returns the base string wrapped to limit chars, separated by newlines.

List Functions

This list function can be used to get a string by evaluating a key chain on a list, like this:

```
@getMovies.join("getTitle", ", ")@
@hmtl(getMovies.join("getTitle", "<br>"))@
```

get(index): Returns the individual object at the given index.
join(key, delimiter): Returns a String by evaluating key string on each list item, joined by delimiter string.
filter(key): Returns the sublist of objects that evaluate to true for the given key string.
count(key): Returns the number of objects that evaluate to true for given key string.
countUnique(key): Returns the number of unique results obtained by evaluating given key string against list.