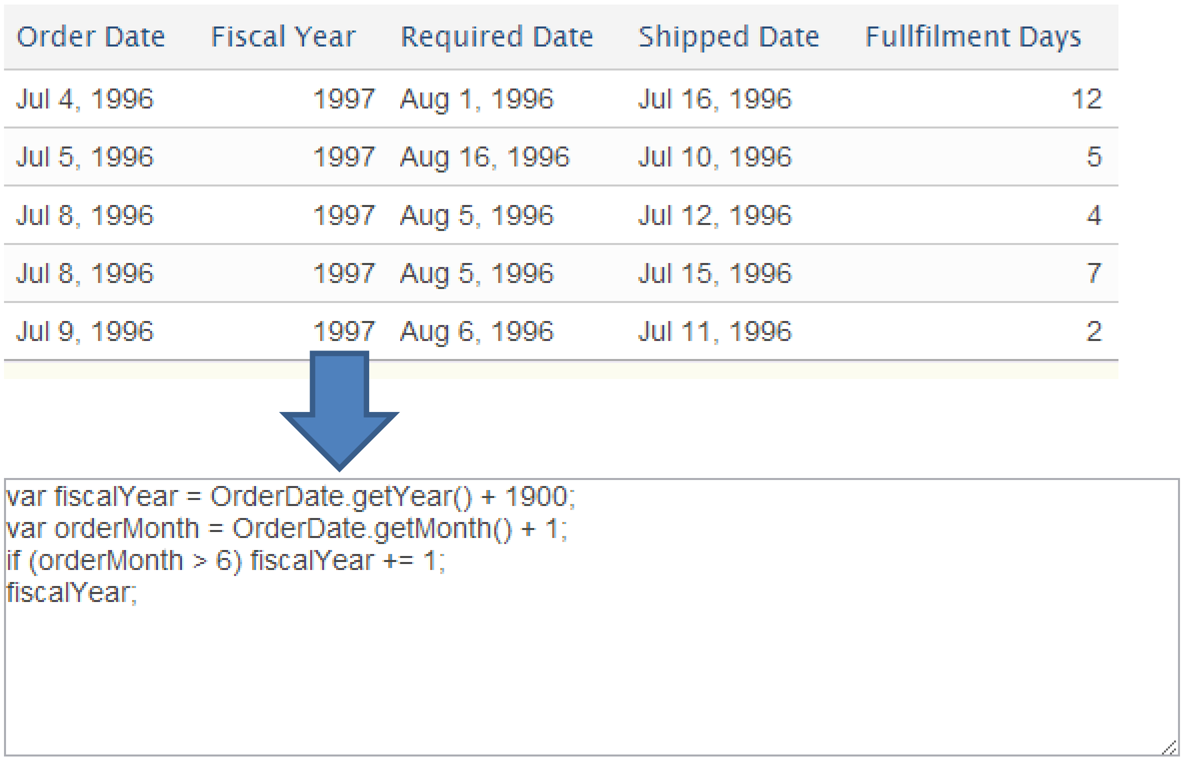
Informer JavaScript Guide

Table of Contents

Chapter 1: Introduction

About Calculated Columns

Informer allows you to extend your reporting capabilities by adding calculated columns. Calculated columns use report data, literal values, or other data to do calculations, such as adding two column values together or concatenating first name and last name together. But that is not all they do. They are very powerful and can make your life much easier when generating reports. This class will help you understand how to write script calculated columns using JavaScript.



Order Date	Fiscal Year	Required Date	Shipped Date	Fullfilment Days
Jul 4, 1996	1997	Aug 1, 1996	Jul 16, 1996	12
Jul 5, 1996	1997	Aug 16, 1996	Jul 10, 1996	5
Jul 8, 1996	1997	Aug 5, 1996	Jul 12, 1996	4
Jul 8, 1996	1997	Aug 5, 1996	Jul 15, 1996	7
Jul 9, 1996	1997	Aug 6, 1996	Jul 11, 1996	2

```
var fiscalYear = OrderDate.getYear() + 1900;
var orderMonth = OrderDate.getMonth() + 1;
if (orderMonth > 6) fiscalYear += 1;
fiscalYear;
```

Template vs. Script

There are two different kinds of calculated columns in Informer: **template** and **script**. Template columns use plain text and HTML to format data. Placeholders, similar to those used in mail merge, allow you to insert column values into the results of the calculation. Template columns only work with single-value columns, and they cannot do computations or conditional statements.

Script columns utilize the JavaScript language to write more complex calculated columns. You can do everything in a script column that you can with a template, as well as those things you can't. These are the columns on which this class will focus.

What Is JavaScript?

JavaScript was developed in 1995 by Netscape for use in their web-browser. The browser did not survive the browser wars, but fortunately, JavaScript is still around. It is primarily used as a client-side scripting language, meaning it is executed on the desktop and not the server. It gained popularity in the programming world when the Ajax development platform was introduced in 2005. It is supported in all major browsers.

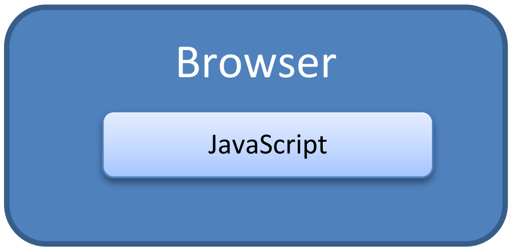
Java vs. JavaScript – Are They The Same Thing?

The simple answer here is "No". They are two totally different languages, although the JavaScript syntax was influenced by Java. But that is where the similarities stop.

Java	JavaScript
Compiled to bytecode	Interpreted at runtime
Strongly Typed	Weakly Typed
Class Based Hierarchy	Prototype-based Hierarchy

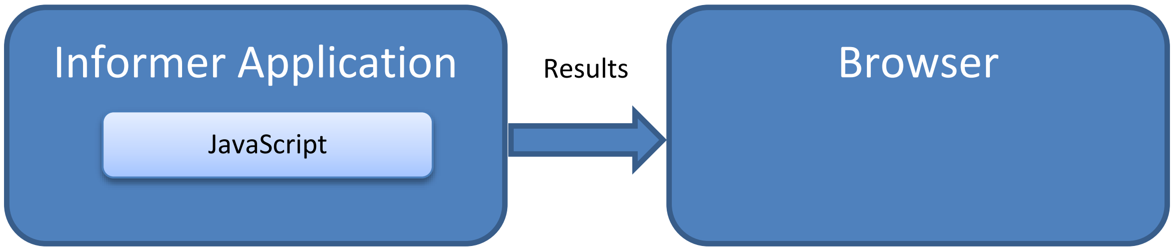
Informer JavaScript

Informer JavaScript is hybridization of JavaScript. Normally, when you request a webpage, the server sends the page to the browser where JavaScript is interpreted and executed.



```
document.write("Hello World");

alert("DANGER Will Robinson!");

window.print();
```

With Informer, however, JavaScript is interpreted within the Informer Java application on the server, and the results are then sent to the browser for rendering.



Because of this, you cannot affect any of the browser objects in the Document Object Model (DOM), but, this also means you have access to standard Java objects:

```
var today = new java.util.GregorianCalendar();
```

Chapter 2: Objects: A Brief Introduction

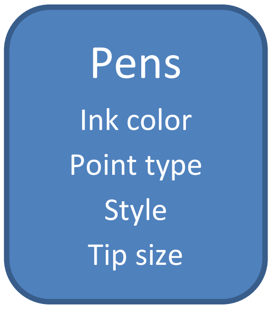
In This Chapter

In this chapter, we will:

- Define objects
- Learn how to interact with objects
- List the different objects in Informer

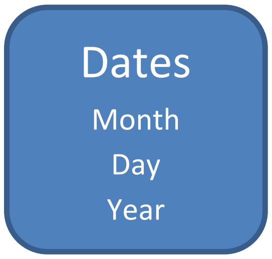
Introduction to Objects

An **object** can be thought of as any thing. Specific objects belong to the general classes of objects. For example, a pen belongs to a class “pens”. Classes of objects have certain common characteristics, or properties.



A specific pen is an **instance** of the class Pen. This is called **instantiation**.

Data can be put in classes as well. Dates, for example, are an example of a class of data. All dates have characteristics as well.



Interacting With Objects

In order to use objects, you need to interact with them. Using our pen example, I may want to know what color of ink is in my pen. Or, I may want to extract the month from my date object, or set the day of the month. Methods and functions are used to interact with objects.

So, what’s the difference between a method and a function? Technically speaking, a **method** takes no arguments, where a **function** does. For example, if I want to get the pen color, I do not need to pass anything to determine that value. The color is what it is. That would be a method. If, on the other hand, I want to add a certain number of days to my date, I would need to pass the number of days to add. That would be a function. Basically the two are synonymous. Both perform actions on an object.

For the purposes of this class, we will use the term method for both methods and functions.

To use (or call) a method, you simply state the object name, followed by a period, and then the name of the method.

```
myDate.getMonth()
```

The empty parentheses are necessary if the method takes no arguments.

Objects also have **properties** as we mentioned earlier. These are values associated with the particular object. For example, a string object has a length property that tells how many characters are in the object. Properties are referenced in essentially the same way as methods, except they do not take any arguments.

```
firstName.length
```

Informer Objects

In Informer, **columns** are objects. The specific type object class depends on the data type of the column.

Informer Data Type	Object Type	Class Name
Text	JavaScript	String Object
Date (From U2)	Java	java.util.Date
Date (From SQL)	Java	java.sql.Timestamp or java.sql.Date
Boolean	JavaScript	Boolean Object
Numeric	JavaScript	Number Object
Multi-value	JavaScript	Array Object

We will go into each of these objects in Chapter 4.

If you want to find what methods or properties are available for a specific type of object, refer to either the w3schools.com website for JavaScript objects, or the Java documentation found at the Oracle.com website.

Think of something you own as an object (a car, pet, etc.). What would its class be? What kind of properties would it have?

Informer™
Quick and Easy Business Intelligence

Chapter 3: Informer JavaScript

In This Chapter

In this chapter, we will:

- Show the operators used in JavaScript statements
- Discuss the rules for JavaScript statements
- Introduce variables
- Write a simple script column
- Discuss conditional statements
- Discuss controlled loops

Operators

The following list shows the operators that are used when writing your script columns.

JavaScript Operators	
+	Addition or concatenation
-	Substraction
*****	Multiply
/	Divide
%	Modulus (Remainder)
=	Assign value
==	Is equal to
!=	Is not equal to
>	Is greater than
<	Is less than
>=	Is greater than or equal to
<=	Is less than or equal to
&&	And
||	Or
!	Not

JavaScript Syntax

Like all programming languages, JavaScript follows specific rules and syntax. Just a couple of things to note:

- Case sensitivity – case matters in JavaScript. “A” is not the same as “a”.
- Stand-alone statements must end in a semi-colon.
- Comments are indicated by //, or a block of comments can be noted using /* to start the comment block and */ to end the block.

```
//This is a single line comment
/* This is a block
of comments */
```

Variables

When writing your calculated columns, you may find a need to create variables to store literal values or results of expressions or values of other variables.

Variables must be declared in order to be used in your script. To declare a variable, use the **var** keyword.

```
var myVariable;
```

Variable Values

In the example above, myVariable is declared but not assigned any value. This is called an **undefined** variable. Usually it is a good idea to assign a default value to a variable. To do that, you would use the = operator in the declaration.

```
var myVariable = ”Hello”;
```

This assigns a default value of Hello to the variable. The spacing is optional before and after the equal sign.

When assigning a value, the use of quotes can determine the type of data stored in the variable. Values enclosed in quotes, regardless of whether they are numbers, will be treated as text. Do not use quotes if you wish the variable to be treated as a number.

`var x = 5;`	x is treated as a number
`var x = “5”;`	x is treated as text, since the value is enclosed in quotes.

Variable Names

Variable names must start with a letter. After that, you can use letters, numbers, and certain symbols in the variable name. Note the casing of the letters in myVariable. It is fairly common to lower-case the first letter of the variable name, and upper-case the beginning of each subsequent word. This is called **camel case**. It is not required that you follow this convention. Variable names can be all lower case or all upper case or any combination of the two.

Variable names are typically descriptive of the values they hold. For example, if you wanted to store first name in a variable, the variable name might be firstName. Variable names can also be short, like single letters. Commonly used letters in programming are i, j, x, and y. These are used a lot in the mathematical world and have carried over into programming.

Context Variables

Context variables are special variables that allow you to reference specific information about the report being executed.

_context.report.name returns the title of the report.

_context.report.mapping returns the name of the mapping used on the report.

_context.user returns the user principal for the current logged-in user. A principal is a special Informer object. You can get the name of the user with the getName() method.

```
_context.user.getName()
```

_context.arguments allows you to access the values from prompted parameters. Use the get method to retrieve a particular prompt's value.

```
_context.arguments.get("Prompt text")
```

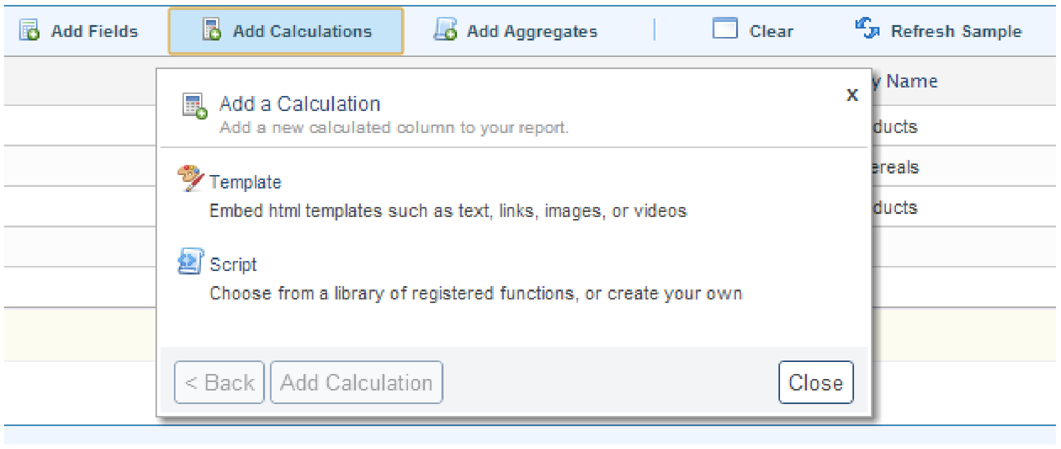
If a report has a prompted parameter for order date, and the prompt text is "Please Enter Order Date", the get method would be

```
_context.arguments.get("Please Enter Order Date")
```

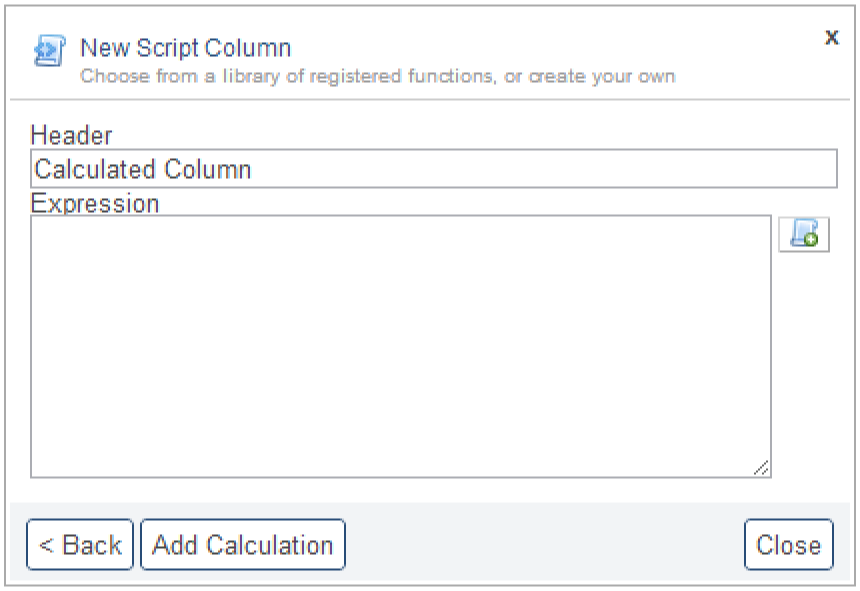
_context.suites returns the table set from which the record was selected.

Creating Script Columns

To create a script calculated column, go to the Column Editor in Informer and click "Add Calculations".


Add Fields
Add Calculations
Add Aggregates
Clear
Refresh Sample
y Name
ducts
ereals
ducts
x
Add a Calculation
Add a new calculated column to your report.
Template
Embed html templates such as text, links, images, or videos
Script
Choose from a library of registered functions, or create your own
< Back
Add Calculation
Close

Click Script to create a script calculated column. This will open the script editor.



In the Header area, key in the name of the column header. This will also used to create the **alias** of the calculated column. Your script statements go in the Expression block.

A column's alias is used to reference the column in a statement. It is the column's variable name.

Statements

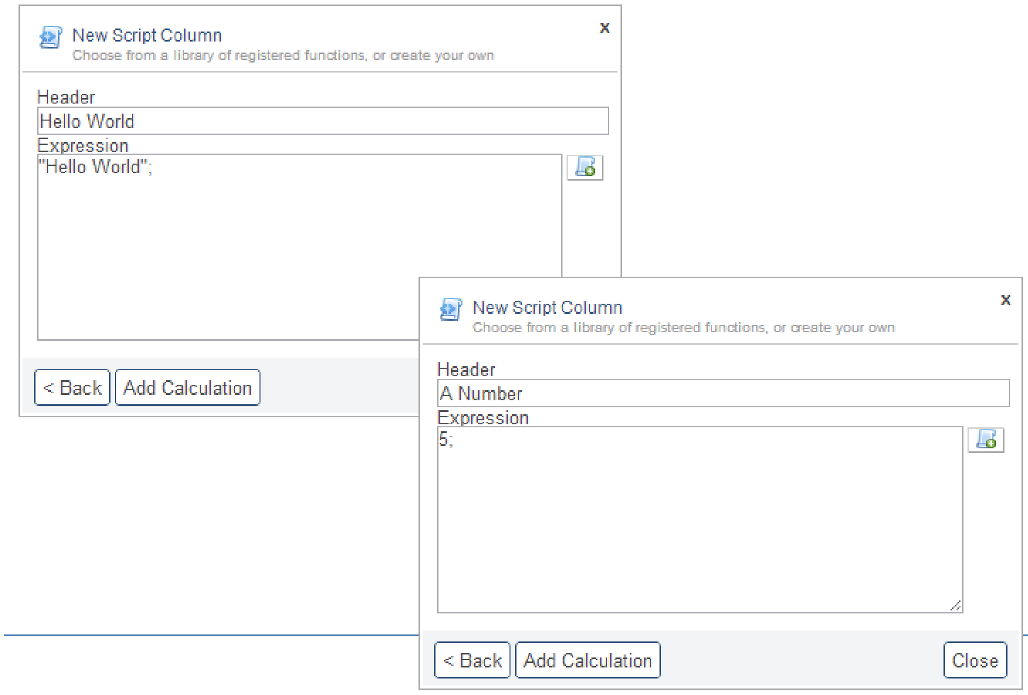
A script calculated column consists of one or more JavaScript statements.

Simple Columns

A simple script column may only have one or two statements. It can be a literal value, the value in a column or variable, or the result of a calculation.

Literal Values

If you want to display a literal value, simply place the value in the expression window.

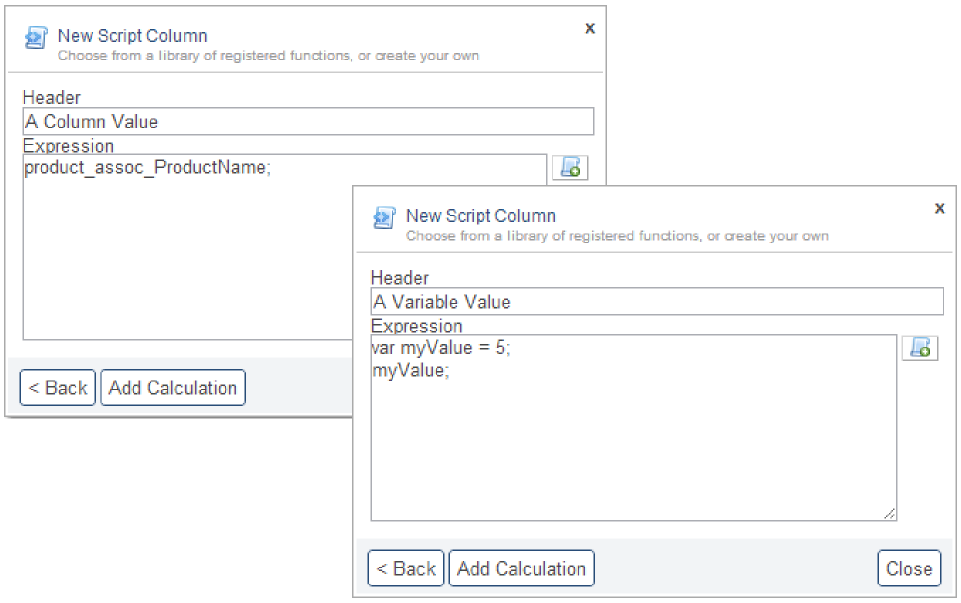


```
"Hello World";
```

```
5;
```

Column or Variable Value

If you want to display the value of a column, enter the column's alias in the expression window. To display the value in a variable, enter the variable name on a line by itself.


New Script Column
Choose from a library of registered functions, or create your own
Header
Expression
< Back
Add Calculation
Close

A Column Value

```
product_assoc_ProductName;
```

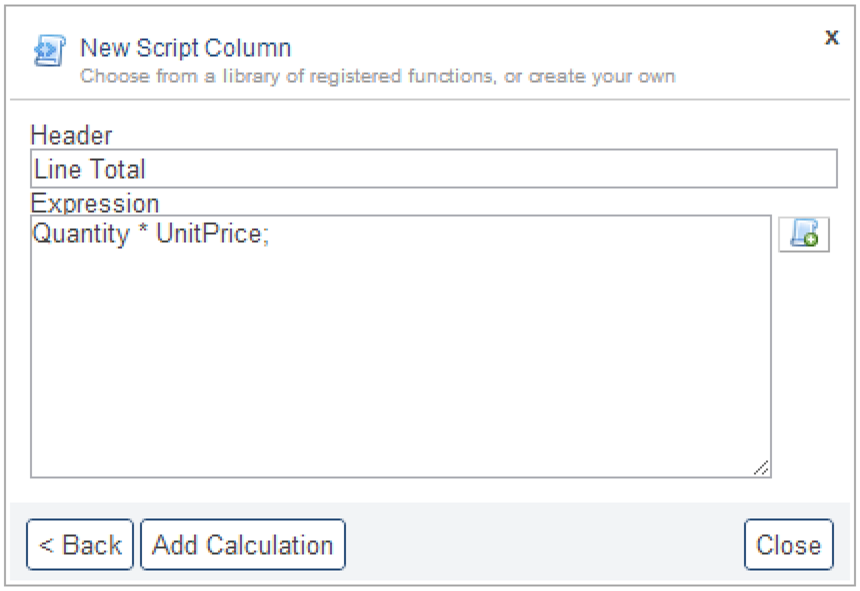
A Variable Value

```
var myValue = 5;
myValue;
```

Hint: Instead of typing the name of the column alias, you can drag the column header from the report sample into the expression window.

Calculation Result

Your expression can also be a calculation.


New Script Column

Choose from a library of registered functions, or create your own

Header

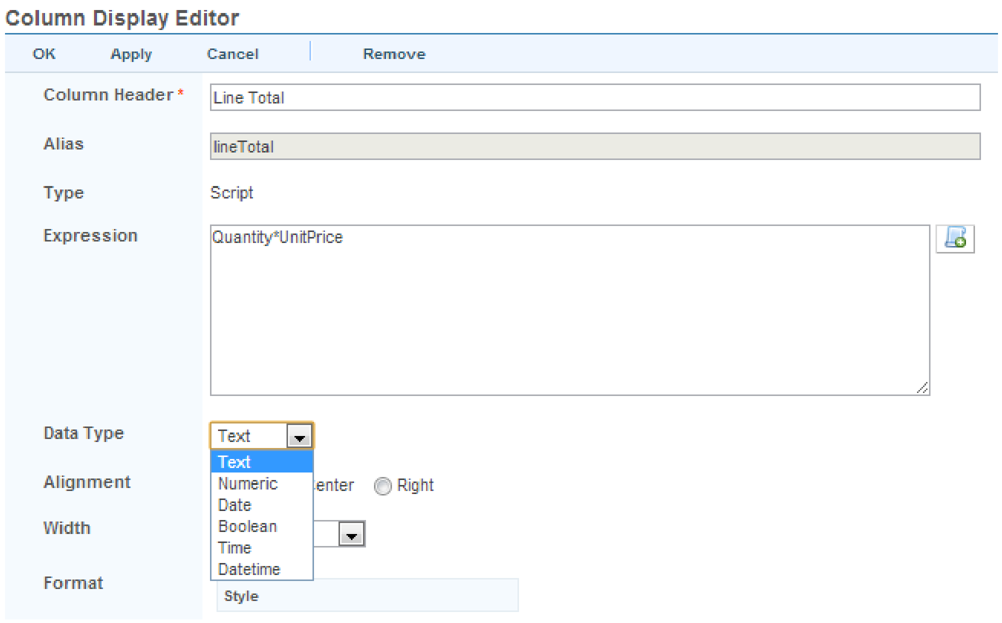
Line Total

Expression

Quantity * UnitPrice;

Column Type

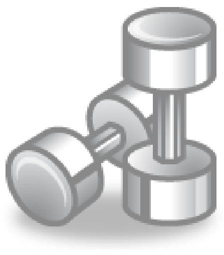
When you create a calculated column in Informer, it defaults to a data type of text. To change the data type, click the calculated column's header, and in the Column Display Editor, change the Data Type field just below the expression.


Column Display Editor

Column Header *	Line Total
Alias	lineTotal
Type	Script
Expression	Quantity*UnitPrice
Data Type	Text

Exercise

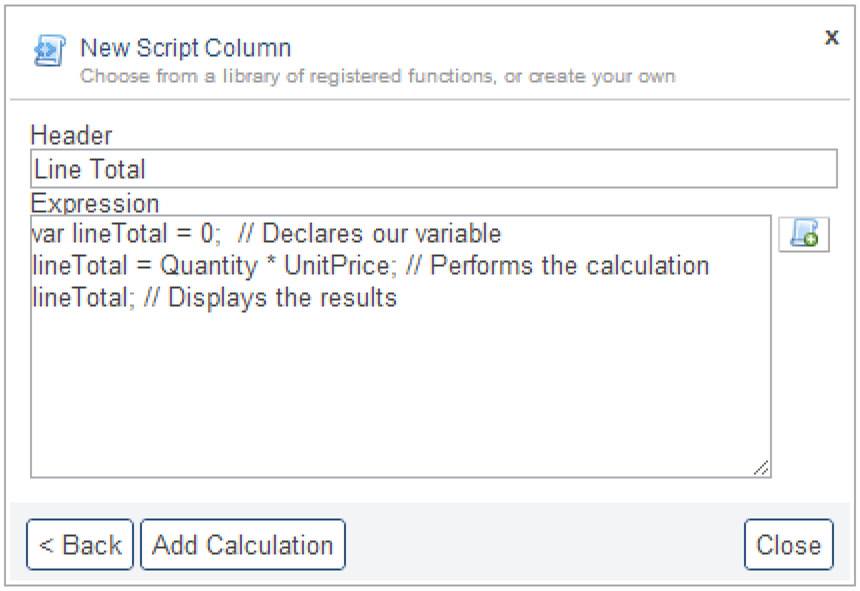
1. Open the Order Details Report
2. Go to the Columns page
3. Add a calculated column that displays "Hello World"



4. Add a calculated column that displays the value in the Company Name column
5. Add a calculated column that calculates the order total for each line by multiplying Quantity and Price

Complex Columns

A more complex calculated column will have many statements. You may need to do several steps in order to generate the results you wish to have. For example, you may want a column that declares a variable, performs a calculation, and then displays the results.


New Script Column
Choose from a library of registered functions, or create your own
< Back
Add Calculation
Close

Header

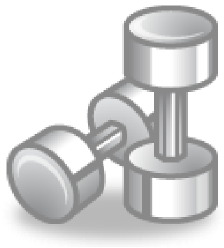
Line Total

Expression

```
var lineTotal = 0;  // Declares our variable
lineTotal = Quantity * UnitPrice; // Performs the calculation
lineTotal; // Displays the results
```

Note: A script must eventually display either a literal value or the value of a variable or column. If the script does not display a value, you will see [object Object] displayed in the column.

Exercise



1. In the Order Details Report, create the calculated column shown above to calculate the line item total.

Conditional Statements

There may be situations in your script where you only want to do something when a certain condition is true (or not true). This is where conditional statements come in to play. There are two different conditional statements that you can use in JavaScript: if and switch.

If Statements

If-statements perform one or more statements when the condition or conditions are true. The basic syntax is:

```
if (condition) statement;
```

You can perform more than one statement if the condition is true by enclosing the statements in curly braces ({}). This is called a **code block**. The statements are indented strictly for readability.

```
if (condition)
{
     Statement1;
     Statement2;
     .
     .
     .
}
```

The curly braces do not need to go on their own line.

```
if (condition){
     Statement1;
     Statement2;
     .
     .
     .
     StatementN;}
```

If you have one or more additional statements that you want to execute when the condition is not true, use an else-statement.

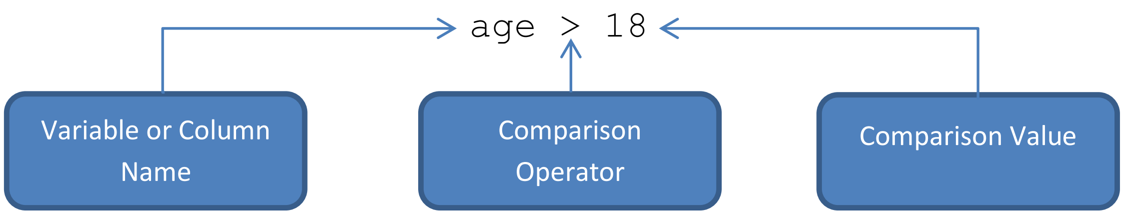
```
if (condition)
{
     Statement1;
     Statement2;
}
else
{
     Statement3;
     Statement4;
}
```

If-statements can also be embedded.

```
if (condition1)
{
     Statement1;
     Statement2;
}
else
{
     if (condition2)
     {
          Statement3;
          Statement4;
     }
}
```

Conditions

Conditions use comparisons to determine if something is true. A comparison typically consists of a variable or column alias, a condition operator, and a comparison value. The comparison value can be either a literal or another variable or column alias.



```
  firstName == "John"
qtyOnHand <= qtyOrdered
```

Refer to the operators list at the beginning of this chapter for a list of comparison operators.

So to display a message if the first name is "John"…

```
if (firstName == "John")
     "Hello John!";
else
     "";
```

The else statement is necessary here because our script MUST display something. If we left the else off and the first name isn't John, then we would get the [object Object] message.

Switch Statements

Switch statements work much in the same way as if statements. Switch statements can take the place of multiple if/else statements. It uses a series of case statements that compare an expression *n* to a given value.

```
switch (n)
{
case value1:
     statement block;
     break;
case value2:
     statement block;
     break;
.
.
.
default:
     statement(s) to execute if none of the values match
}
```

The case statement block is not enclosed in curly braces.

The switch statement evaluates the value of *n* one time and compares the value to each of the case values. If the values match, the statement block is executed (note the statement block does not have to be enclosed in curly braces). The break statement is necessary to prevent subsequent case statements from being evaluated, thus causing unwanted results. The default statement does not need a break since all the case statements have been exhausted.

Example:

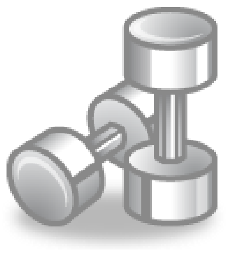
```
var myVariable = 2;
switch (myVariable)
{
case 0:
     "Value is 0";
     break;
case 1:
     "Value is 1";
     break;
case 2:
     "Value is 2";
     break;
default:
     "No match found!";
}
```

You are not limited to literal values. You can use the following example to do relative conditional statements as well:

```
var myVariable = 2;
switch (true)
{
case (myVariable < 2):
     "Is less than 2";
     break;
case (myVariable > 2):
     "Is more than 2";
     break;
default:
     "Must be equal to 2";
}
```

Exercise

1. Using the Order Details Report, create a calculated column that displays a message "Backordered" if the Quantity exceeds the Units in Stock, or no message if not.



Controlled Loops

Controlled loops allow you to execute code repeatedly with a different value each time.   There are 3 types of controlled loops: for, while, and do.

For Loops

For-loops execute a block of code a given number of times. It has the following syntax:

```
for (statement 1; statement 2; statement 3)
{
     Code block to execute
}
```

Statement 1 – the statement executed the first time into the loop. Typically this declares the variable that is used as the loop counter and sets the initial value.

Statement 2 – this is the condition statement which controls how long the loop is to run.

Statement 3 – the statement that is executed at the end of each loop. Typically this increments or decrements the loop counter defined in statement 1.

In this example, we will create a loop that adds together the numbers 1 through 5.

```
var x = 0;
for (var i = 1; i <= 5; i++)
{
     x = x + i;
}
x;
```

JavaScript has a shortened notation for adding 1 to a variable. Simply state the variable and put two plus signs after it – i++ in this example.

The first time through the loop, i is set to 1 and x is 0, so 0 + 1 is 1. At the end of the loop, i is incremented by 1. Since 2 is less than 5, the loop will continue to the next iteration. x becomes 3 (1 + 2), i is incremented by 1 to 3, 3 is less than 5, etc. When i becomes 5 on the last time through, x becomes 15 (10 + 5), and i is incremented by 1 to 6, which is greater than 5. At that point, the loop execution stops.

It is possible to create a never-ending loop, which will lock up your browser, and possibly the Informer server. Be careful when creating loop structures to ensure the loop eventually stops.

For-loops are especially useful when processing arrays, which we will discuss in the next chapter.

While Loops

While-loops execute as long as a given condition is true. It has the following syntax:

```
while (condition)
{
     Code block to execute
}
```

The condition statement is the same as the ones used in if statements discussed earlier in this chapter. It is checked before each loop is executed.

This is how our code to add the numbers 1 to 5 together would look using a while- instead of a for-loop.

```
var x = 0;
var i = 1;
while (i <= 5)
{
   x += i;
   i++;
}
x;
```

Here is another example of JavaScript's shortened notation. x+=i is the same as x = x + i.

This code does essentially the same thing as the for-loop.

Do/While Loops

Do/While loops are a variation of the while-loop, except they execute the code block first and then check the condition. This means that the code block is always executed at least once.

```
do
{
code block to execute
}
while (condition);
```

So our earlier example would look like this using a do-while loop:

```
var x = 0;
var i = 1;
do
{
   x += 1;
   i++;
}
while (i <= 5);
x;
```

Loop Controls

Loop controls allow you to affect the flow of the loop. For example, you want to completely stop the loop cycle, or you want to skip the remaining logic in the current loop iteration and go to the next iteration.

break will stop the loop completely.

continue will stop the current loop iteration and skip to the next.

In this example, we use break to stop the loop completely after the count reaches 3.

```
var x = 0;
for (var i = 0; i <= 5; i++)
{
   if (i == 3) break;
   x += i;
}
x;
```

The result would be 3.

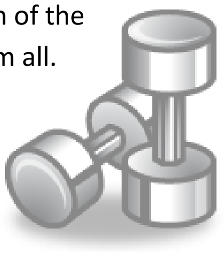
In this example, we use continue to skip the iteration where the variable i is 3.

```
var x = 0;
for (var i = 0; i <= 5; i++)
{
   if (i == 3) continue;
   x += i;
}
x;
```

The result would be 12.

Exercise

1. Using the order details report, create a calculated column that gives the sum of the numbers 1 through 50. Use any of the loop statements you wish, or try them all.
2. Now create a calculated column that adds all the numbers from 1 to 50, but stops after the sum total reaches 50.
3. Now create a calculated column that gives the sum of all the odd numbers (Hint: there are two ways of doing this – can you find them?)



Informer™
Quick and Easy Business Intelligence

Chapter 4: Data Types

In This Chapter

In this chapter, we will:

- Review the different data types used in Informer
- Review Numeric data type and its methods
- Review String data type and its methods
- Review Date data type and its methods
- Review Array data type and its methods

Informer Data Types

This is the table we reviewed earlier in chapter 2.

Informer Data Type	Object Type	Class Name
Text	JavaScript	String Object
Date (From U2)	Java	java.util.Date
Date (From SQL)	Java	java.sql.Timestamp or java.sql.Date
Boolean	JavaScript	Boolean Object
Numeric	JavaScript	Number Object
Multi-value	JavaScript	Array Object

Numeric Data Type

As its name implies, numeric data types store numbers. They can be whole numbers (integers) or real numbers (decimal values). You can perform any mathematical operation on numeric values.

```
var x = 34;
var pi=3.1415927;
```

When declaring a numeric variable, do not use quotes, as this will cause the variable to be treated as a string. You can convert a numeric value stored as a string to a number using either the parseInt or the parseFloat functions.

```
var xString = “34”;
var xInt = parseInt(xString);
var piString = “3.1415927”;
var piFloat = parseFloat(piString);
```

String Data Type

String data types allow you to store text and numbers. String literal values can be enclosed in either single or double quotes.

```
var firstName = “John”;
var password = ‘pa$$w0rd123’;
```

String Properties, Methods & Operations

Length

To determine the length of a string, use the length property.

```
var txt = "Hello World";
txt.length;  // 11
```

Concatenation

Concatenation combines two or more strings into one. There are two ways to do concatenation: using the + symbol and using the concat() method.

```
var firstName = "John";
var lastName = "Doe";
var fullName = firstName + " " + lastName; // John Doe
var concatName = firstName.concat(lastName); // JohnDoe
```

Substrings

You can reference each character in a string using its position (or index). The position numbering starts at 0 for the first character. The syntax is

String[*n*]

where *n* is the position of the character to extract.

```
var txt = "Hello World";
var firstChar = txt[0];  // H
var thirdChar = txt[2];  // l
```

To extract a portion of a string, use either the substr or substring method. The syntax for substr is

String.substr(*Start*, *Length*)

Start is the position in the string where the extraction starts. The first character is at position 0. *Length* is the number of characters to extract. This is an optional field. If omitted, the result is the remaining portion of the string from the starting position.

```
var txt = "Hello World";
var first7 = txt.substr(0,7);  // Hello W
var middle3 = txt.substr(2,3); // llo
var last5 = txt.substr(6);     // World
```

The substring method works a little differently. Its syntax is

```
String.substring(Start, End)
```

Start is the position in the string where the extraction starts. The first character is at position 0.
End is the position where to stop the extraction. If omitted, the result is the remaining portion of the string from the starting position.

```
var txt = "Hello World";
var first7 = txt.substring(0,7);   // Hello Wo
var middle3 = txt.substring(2,3);  // ll
var last5 = txt.substring(6);      // World
```

Finding a String in a String

Use the indexOf method to search for a string within a string.

```
String1.indexOf(String2)
```

The method searches for the first occurrence of *String2* in *String1* and returns the position of the occurrence. If *String2* does not occur in *String1*, the result is -1;

```
var txt = "Hello World";
var firstL = txt.indexOf("l");     // 2
var rldPos = txt.indexOf("rld");   // 8
var n = txt.indexOf("John");       // -1
```

Breaking Apart a String

The split method splits a string into an array of substrings.

```
String.split(delimiter, limit)
```

The string is split on each occurrence of *delimiter*, up to the optional *limit* value. If an empty string ("") is used as the delimiter, then the split occurs between each character.

The value of *String* is not changed.

```
var txt = "Hello World";
var arr1 = txt.split(" ");   // {Hello,World}
var arr2 = txt.split(""); // {H,e,l,l,o, ,W,o,r,l,d}
var arr3 = txt.split("",3); // {H,e,l}
```

We will talk more about arrays later in this chapter.

Changing Case

To change the case of characters within a string, use the methods toUpperCase and toLowerCase.

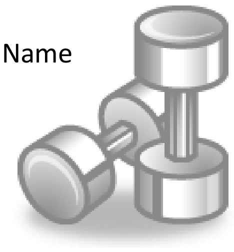
```
String.toUpperCase();
String.toLowerCase();
```

The entire string is cased depending on which method is used.

```
var txt = "Hello World";
var txt1 = txt.toUpperCase(); // HELLO WORLD
var txt2 = txt.toLowerCase(); // hello world
```

Exercises

These exercises will use the Northwind Orders report.



1. Create a calculated column that displays the number of characters in the Last Name field.
2. Create a calculated column that concatenates the first and last name together. The format should be last, first.
3. Create a calculated column that displays the first initial, a period, a space, and the first three characters of the last name. ex: J. Smi for John Smith.
4. Create a calculated column that forces the customer name to all upper case.

CHALLENGE: Using the last name field, create a calculated column that upper cases all of the characters in odd-numbered positions. (for Smith, it would display SmItH).

Date Data Type

As shown in the data type chart, dates are one of the exceptions to the use of JavaScript objects in Informer. The specific Java date object used depends on the reporting database type. U2-based dates use the java.util.Date object. SQL-based dates use either the java.sql.Timestamp or java.sql.Date object. If you are creating a date variable, you can simply use java.util.Date.

*You may also see references to the java.util.Calendar object in our forums. This is the object that will eventually replace the java.util.Date object at some point in the future, but Java still retains the functionality of the java.util.Date object. The java.util.Date object is said to be **deprecated.***

Date Methods and Operations

Most of the same methods used with JavaScript dates are also available with Java dates, as well as some methods that aren't available with JavaScript dates.

Creating a Date Variable

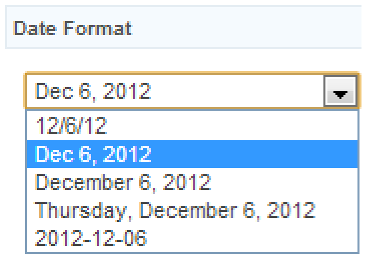
When creating a date variable, we want to create a new Java Date object.

```
var today = new java.util.Date();  // 12/6/12
```

This creates a new instance of the java.util.Date object and sets it to the current date. The following examples assume the current date is 12/6/2012.

Changing the Date Format

Informer allows you to specify certain date formats in the Column Display Editor.



Be sure to change your calculated column's data-type to Date in order to set the formatting options and to use your calculated column as a date elsewhere.

However, there may be times where you need a format not listed here. You can use a Java SimpleDateFormat object to change the format also.

Creating the SimpleDateFormat Object

The SimpleDateFormat constructor syntax is:

SimpleDateFormat(*String*)

String is a format pattern which determines how the date is formatted. Here are some common pattern letters that are used:

Letter	Date or Time Component	Examples
y	Year	2012; 12
M	Month in Year	December; Dec; 12
d	Day in Month	25
E	Day name in week	Wednesday; Wed
u	Day number of week (1 = Monday … 7 = Sunday)	1
a	AM/PM marker	PM
H	Hour in day (0-23)	15
k	Hour in day (1-24)	16
K	Hour in am/pm (0-11)	3
h	Hour in am/pm (1-12)	4
m	Minute in hour	30
s	Second in minute	25
S	Millisecond	999

The pattern letters are repeated to indicate format. For year, yyyy would display 4 digits in the year, and yy just 2 digits. For month, if the pattern is 3 or more characters, the textual representation of the month is presented; otherwise the numeric value is presented zero-padded to the number of pattern letters. For other numeric fields like day or hour, the value is presented zero-padded to the number of pattern letters. For day name, if the number of pattern letters is 4 or more, the full form of the day name is presented; otherwise, an abbreviated version is displayed.

```
// Create a formatter to format a date as MM-DD-YYYY
// Example: 12-06-2012
var sdf = new java.text.SimpleDateFormat("MM-dd-yyyy");
```

Formatting a Date

Once you have a formatter object created, you simply call its format method, passing the appropriate date or time value.

```
SimpleDateFormat.format(Date);
```

If we have a date called myDate, we can pass it to the formatter to format it appropriately.

```
sdf.format(myDate);
```

Getting the Components of a Date

To extract the month, day, or year of a date, use one of the following methods.

```
Date.getMonth()
Date.getDate();
Date.getYear();
```

getMonth

The getMonth method returns the month number of *Date*. January is month 0, February is 1, etc.

```
today.getMonth();  // 11 (December)
```

getDate

The getDate method returns the day of the month, 1-31.

```
today.getDate();  // 6
```

getYear

The getYear method returns the year portion of *Date*. Prior to the year 2000, this method returns the 2-digit year. For year 2000 and beyond, this method returns a 3-digit date, with the first digit being a 1 and the last 2 digits the actual year. Year 2000 would return 100. In order to get the 4-digit year, you need to add 1900.

```
today.getYear();  // 112
today.getYear() + 1900;   // 2012
```

JavaScript has a date method called getFullYear, which returns the 4-digit year. Since we aren't using JavaScript date objects, that method is not available. The java.util.Calendar object has a get method that will also return the 4-digit year. Refer to the online documentation.

Setting the Date

If you need to set a date variable to a specific date, you can either do so at the time you create the new date object, or you can use the appropriate set methods.

Setting the Date at Creation Time

To set the date when the date object is created, simply pass the date as an argument to the object. The format of the date argument can be a string, or the date components themselves.

```
java.util.Date(String)
java.util.Date(Year, Month, Date)
```

Month must be the 0-based value of month. *Year* must be the appropriate 2- or 3-digit year, not the 4-digit year. If the year is in 4-digit format, subtract 1900.

So, to create a date that is set to 12/13/08:

```
var date1 = new java.util.Date("12/13/08");
var date2 = new java.util.Date(108, 11, 08);
var date3 = new java.util.Date(2008-1900, 11, 08);
```

Setting the Date Using Methods

There are three methods for setting the date components: setMonth, setDate, and setYear.

```
Date.setMonth(Month)
Date.setDate(Day)
Date.setYear(Year)
```

The same stipulations apply for *Month* and *Year* as above.

```
var theDate = new java.util.Date(); // Today's date
theDate.setMonth(10);    // Sets the month to November
theDate.setYear(108);    // Sets the year to 2008
```

Comparing Dates

The Java Date object has methods that compare dates. Each will tell you whether a date is after, before, or equal to another date.

After

To see if one date comes after another, use the after method.

```
Date1.after(Date2)
```

After returns true if *Date1* falls after *Date2*, or false if it does not.

```
var date1 = new java.util.Date("12/1/12");
var date2 = new java.util.Date("11/30/12");
if (date1.after(date2))
   "Date 1 is after Date 2";  // Displays
else
   "Date 1 is on or before Date 2";
```

Before

To see if one date comes before another, use the before method.

```
Date1.before(Date2)
```

Before returns true if *Date1* comes before *Date2*, or false if it does not.

```
var date1 = new java.util.Date("12/1/12");
var date2 = new java.util.Date("11/30/12");
if (date1.before(date2))
   "Date 1 is before Date 2";
else
   "Date 1 is on or after Date 2";  // Displays
```

Equals

To see if two dates are equals, use the equals method.

```
Date1.equals(Date2)
```

Before returns true if *Date1* is the same as *Date2*, or false if it is not.

```
var date1 = new java.util.Date("12/1/12");
var date2 = new java.util.Date("12/1/12");
if (date1.equals(date2))
   "Date 1 is equal to Date 2"; // Displays
else
   "Date 1 is not equal to Date 2";
```

Using the compareTo method

The compareTo method allows you to see how one date relates to another, whether before, on, or after.

```
Date1.compareTo(Date2)
```

The result of the method depends on how Date1 relates to Date2.

Condition	Result
Date1 equals Date2	0
Date1 before Date2	-1
Date1 after Date2	1

```
var date1 = new java.util.Date("12/1/12");
var date2 = new java.util.Date("12/1/12");
var xResult = date1.compareTo(date2); // 0
```

Date Math

Using date math, you can add and subtract days to a date, or calculate the amount of time that has spanned between two dates. Unfortunately the java.util.Date object does not have any methods that do date math. This is where the Calendar object can be useful.

Creating a Calendar Object

As mentioned earlier, the Calendar object (java.util.Calendar) is set to replace the Date object as the standard way to manage dates. It comes with similar methods to the Date object, as well as a method for doing date math.

```
var myCalendar = java.util.Calendar.getInstance();
```

Setting The Calendar's Date

As with the Date object, the Calendar is created with the current date set. To change the date on a calendar object to that of an existing Date object, use the setTime method.

```
myCalendar.setTime(dateFromReport);
```

Accessing Calendar Components

And like the Date object, you can also set specific components of the date. It has a single set method, which takes a variable list of arguments.

```
Calendar.set(Year, Month, Date)
Calendar.set(field, value)
```

The first method format works the same as the Date object constructor, except *Year* is in 4-digit format.

The second method format takes a numeric representation of the *field* being set (e.g. month, date, year, etc.) and the actual value to which *field* is set. There are constant names that can be used in place of *field* and *value*. Here is a list of the more commonly used calendar constants that can be referenced in Calendar get or set methods:

JANUARY	DECEMBER
FEBRUARY	MONTH
MARCH	DATE
APRIL	YEAR
MAY	HOUR_OF_DAY
JUNE	MINUTE
JULY	SECOND
AUGUST	MILLISECOND
SEPTEMBER	DAY_OF_WEEK
OCTOBER	
NOVEMBER	

When using these constants, they will need to be fully qualified. So, the MONTH constant would be referenced as java.util.Calendar.MONTH. These statements would set the month to December and get the day of week.

```
myCalendar.set(java.util.Calendar.MONTH, java.util.Calendar.DECEMBER);
myCalendar.get(java.util.Calendar.DAY_OF_WEEK);
```

Advancing a Calendar Date

Once you have a date represented in a Calendar object, you can use the add method to add to and subtract from any of the date components.

```
Calendar.add(field, amount)
```

The add method adds *amount* to the *field*. This statement adds 30 days to the date:

```
myCalendar.add(java.util.Calendar.DATE, 30);
```

To subtract, make *amount* negative. This statement subtracts 30 days from the date:

```
myCalendar.add(java.util.Calendar.DATE, -30);
```

Calculating Time Between Two Dates

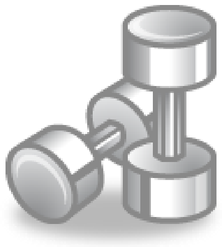
This is more involved than simply saying Date1 – Date2. Since dates are not actual numeric values, we have to get them into some sort of numeric format. The getTimeInMillis method returns the number of milliseconds since the Epoch, which is defined as midnight, January 1, 1970. If we call the getTimeInMillis method using today's date, we would get the number of milliseconds since 1/1/70. If we use the method on another date, we would get the number of milliseconds since 1/1/70 for that date. The difference of the two numbers would be the number of milliseconds between the two dates. All that would be left is to convert the value to the appropriate value – seconds, minutes, hours, days, weeks, years, etc.

Lost? OK, here is how that would look in code. Let's look at a calculation to determine how many days are left until Christmas.

```
// First we need to create a calendar object for today's
// date.
var today = java.util.Calendar.getInstance();

// Because we want to make sure the time of day isn't taken into
// account, we'll set the time to midnight.
today.set(java.util.Calendar.HOUR_OF_DAY, 0);
today.set(java.util.Calendar.MINUTE, 0);
today.set(java.util.Calendar.SECOND, 0);
today.set(java.util.Calendar.MILLISECOND, 0);

// Now let's create a calendar object for Christmas
var xmas = java.util.Calendar.getInstance();

// We have to set the date to December 25
// We will assume that Christmas hasn't already passed this year
xmas.set(java.util.Calendar.MONTH, java.util.Calendar.DECEMBER);
xmas.set(java.util.Calendar.DATE, 25);

// Just like with today's date, we need to zero out the time
xmas.set(java.util.Calendar.HOUR_OF_DAY, 0);
xmas.set(java.util.Calendar.MINUTE, 0);
xmas.set(java.util.Calendar.SECOND, 0);
xmas.set(java.util.Calendar.MILLISECOND, 0);

// Now to do the math...
// 1000 milliseconds in a second
// 60 seconds in a minute
// 60 minutes in an hour
// 24 hours in a day
(xmas.getTimeInMillis() - today.getTimeInMillis) / (1000 * 60 * 60 * 24)
```

Exercises

We will continue using the Northwind Orders report for these exercises.



1. Write calculated columns to pull out the month, day, and 4-digit year of the order date.
2. Create a calculated column that displays the current date. Format it as 12062012 (month, day, and year, without a delimiter).
3. Create a calculated column that displays a message "Late Shipment" if the order was shipped after the required date.
4. Create a calculated column that determines the number of days between the order date and the ship date.

Challenge: Write a calculated column that indicates whether the order was shipped within 30 days, another for over 30 days, another for over 60, and one for over 90. The columns should display a 1 to indicate if the order shipped within the timeframe, or blank if not. You should end up with 4 separate columns.

Array Data Type

Arrays are special objects that allow you to store more than one value in a single variable. For example, you could use an array variable to store the list of names for each student in this class. Multivalued fields from U2 datasources are treated as arrays on reports.

Treat multivalued fields as arrays. Also remember computed columns

Creating Array Variables

Depending on which version of Java you are running on your Informer server, you can create array variables in one of two ways. For Java version 6, use the java.lang.reflect.Array object. Starting with Java version 7, you can use the JavaScript Array object.

Java Version 6 Arrays

Creating arrays with the java.lang.reflect.Array object is more involved than with Java 7 and JavaScript Arrays. The construct requires that you specify the data type being stored in the array, as well as the size of the array (the maximum number of values). Use the newInstance method to create a new array.

```
java.lang.reflect.Array.newInstance(DataType, Size)
```

DataType is one of the standard Java data types and must be fully qualified. Commonly, the data types used are java.lang.String, java.util.Date, java.lang.Integer, and java.lang.Float.

Size specifies the maximum number of values that can be stored in the array.

To create the list of student names we mentioned earlier:

```
var students = java.lang.reflect.Array.newInstance(java.lang.String, 20);
```

The students array is a list of up to 20 String objects.

Java Version 7 Arrays

Starting with Java version 7, you can create arrays using the JavaScript Array object.

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

Note that you do not have to specify the data type being stored or the maximum length.

Referencing Values in the Array

Refer to a specific element value in an array by using its index.

```
Array[index]
```

To set the values in an array, use the index to assign the value to the appropriate array element.

```
students[0] = "John Doe";
students[1] = "Joe Smith";
students[2] = "Beverly Jones";
```

Conversely, you can retrieve the value of a specific element in the same fashion.

```
var secondStudent = students[1];
```

Determining the Size of the Array

To get the number of elements in an array, use the length property.

```
Array.length
```

```
students.length; // returns 3
```

Stepping Through an Array

Using the controlled loop statements covered in 3, you can step through (or iterate over) an array and do something with each of the elements.

Summing Up the Values in an Array

In this example, we will sum up the values in the following array:

```
amount[0] = 100;
amount[1] = 300;
amount[2] = 500;
```

You can use a for-loop to iterate over the element values in the array.

```
var total = 0;
for (var i = 0; i < amount.length; i++)
{
   total += amount[i];
}
```

Note: another format you can use is:

```
for (var i in amount)
{
   total += amount[i];
}
```

Creating a New Array

In this example, we will create a new array that combines the values from two other arrays. We will be using the Java 7 convention for creating arrays.

```
firstName[0] = “John”;
firstName[1] = “Joe”;
firstName[2] = “Beverly”;

lastName[0] = “Doe”;
lastName[1] = “Smith”;
lastName[2] = “Jones”;

var studentNames = new Array();
for (var i = 0; i < firstName.length; i++)
{
   studentNames[i] = firstName[i] + “ ” + lastName[i];
}
```

Sorting Arrays

JavaScript arrays can be sorted using the sort method. Unfortunately, Java arrays do not have a sort method.

The sort method returns the sorted array.

```
lastName[0] = “Doe”;
lastName[1] = “Smith”;
lastName[2] = “Jones”;

lastName.sort();
```

The new array will be:

```
lastName[0] = “Doe”;
lastName[1] = “Jones”;
lastName[2] = “Smith”;
```

The sort performed is alphabetic, so a list of numbers will not be sorted properly (“40” comes before “5”). In order to perform a numeric sort, you must pass a function as an argument to the sort method.

```
amount[0] = 100;
amount[1] = 300;
amount[2] = 500;
```

```
amount.sort(function (a,b) {return a-b}); // ascending
amount.sort(function (a,b) {return b-a}); // descending
```

Finding the Minimum and Maximum Value

Using sorts, you can easily determine the minimum or maximum numeric value in an array. For minimum, sort the array in ascending order and retrieve the first value of the array. For maximum, sort the array in descending order and retrieve the first value of the array.

Converting Arrays to Strings

You can convert an array to a string using the join method.

Array`.join(`*delimiter*`)`

The delimiter is used to separate the values in the string.

```
lastName[0] = "Doe";
lastName[1] = "Jones";
lastName[2] = "Smith";
lastName.join(",");
```

This will result in a string of "Doe,Jones,Smith".

Multi-dimensional Arrays

Arrays can also contain other arrays as elements. This is called a multi-dimensional array. Instead of a single index, you have multiple.

Array`[`*index1*`][`*index2*`]...[`*indexN*`]`

An array of an array is said to be two-dimensional. An example of this might be a list of employee wages. An employee may have had more than one position in a company over time. Each of those positions would likely have multiple wage records (hopefully they would get a raise at some point). So if we wanted a list of all wages for all employees, the result would be in a two-dimensional array.

```
employeeWage[0][0]; // The 1st employee's first wage
employeeWage[0][1]; // The 1st employee's second wage
employeeWage[1][0]; // The 2nd employee's first wage
```

For clients using U2 datasources, multi-dimensional arrays can occur when you use a link built on a multi-valued pointer to pull a multi-valued field.

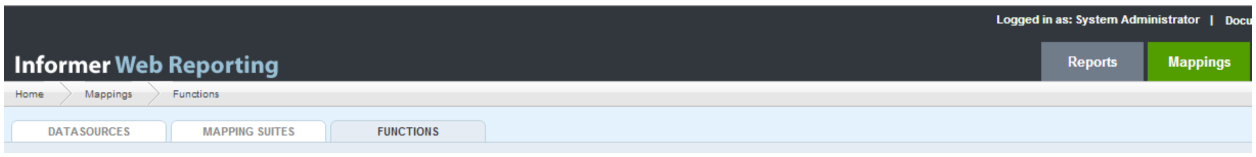
Exercises

We will continue using the Northwind Orders report for these examples.

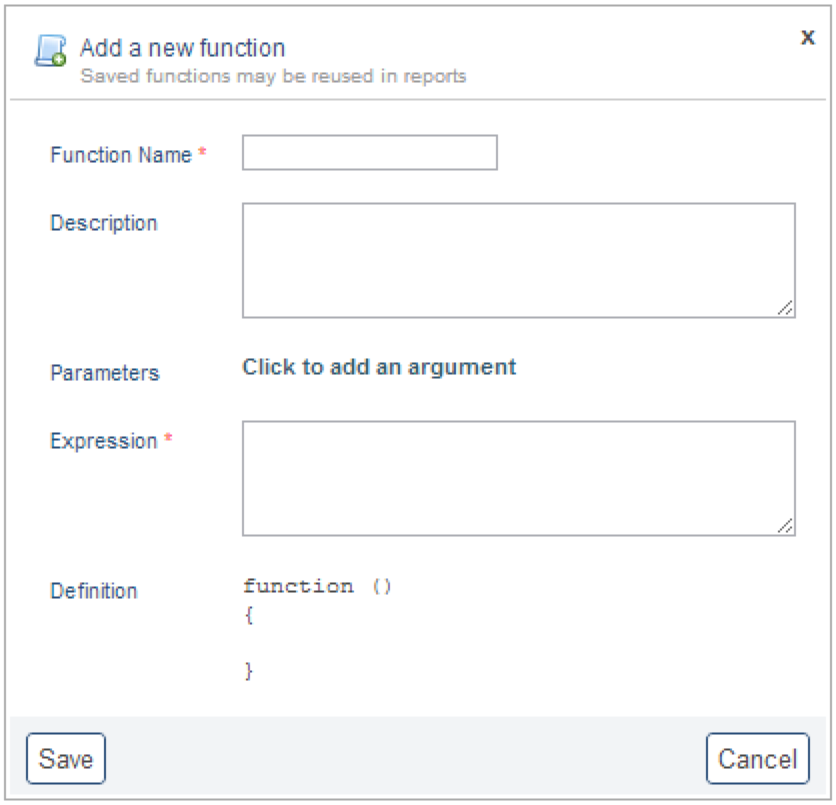
1. Write a calculated column that returns the number of items on the order.
2. Write a calculated column that displays a message if one of the quantities ordered is above 70.
3. Write a calculated column that creates an array of the extended price on each line items (price * qty).
4. Write a calculated column that adds the line extended prices to get the order totals.

Functions

Functions are ways of "saving" your calculated columns for future use. Just like object functions, your functions can take optional parameters as input and return the resulting value. Functions can be defined in the functions area under the Mappings tab.



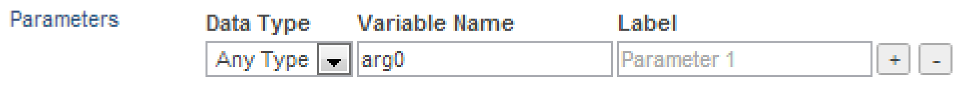
To create a new function, click the "Add Function" icon in the upper right corner of the browser. The Add a new function dialog appears.


Add a new function
Saved functions may be reused in reports

Function Name *	
Description	
Parameters	Click to add an argument
Expression *	
Definition	function () { }

The Function Name must adhere to the same naming conventions as variable names, in that they must begin with a letter.

Parameters are values passed into the function (but they are not passed back, or returned). Choose "Click to add an argument" to add a parameter.



You can specify the data type expected for this parameter, or use Any Type to mean any type of data can be passed. When adding the function to the report, each column is listed in the drop-down. Those with the same data-type as the parameter will be sorted to the top of the list and bolded.

To add additional parameters click the plus icon to the right of the parameter.

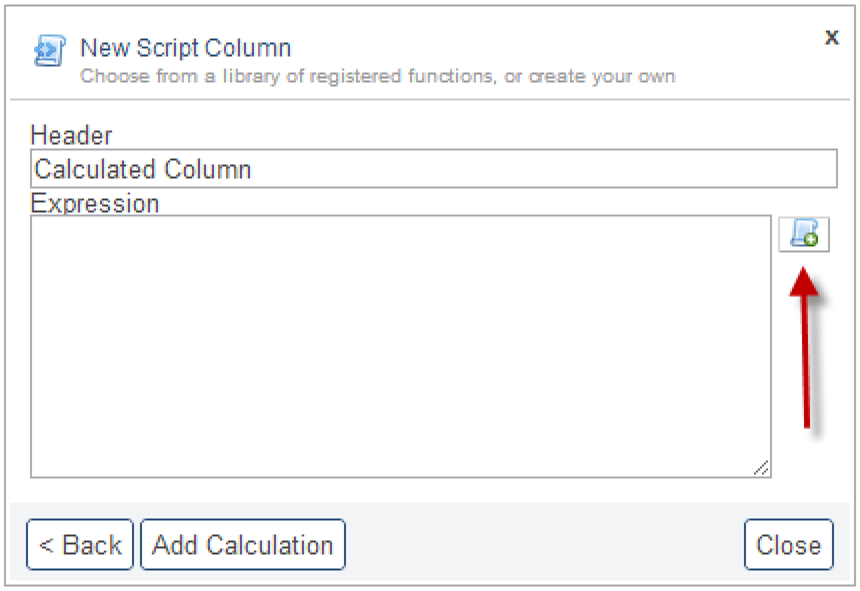
The expression is nothing but a series of statements similar to what we have been working with already. The only difference is that JavaScript Functions do not display the results. They do, however, have to return a value.

```
return value;
```

This is the function code to determine the minimum value in an array. Note that it is good practice to check for null values in one or more of the arguments before proceeding.

```
var sortedArray = new Array();
if (arg0 != null)
   var sortedArray = arg0.sort(function (a,b) {return a-b});
return sortedArray[0];
```

To reference an existing function in a calculated column, click on the Saved Function icon to the right of the expression window.

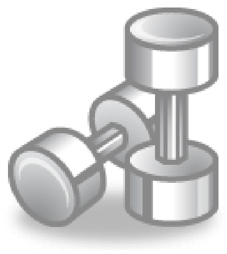


Choose your function on the next window. Then, define the arguments and from where you are getting them. Click Add Function.

You will notice Informer adds the code to call the function,   You can also create a variable to store the results of the function.

Exercises

We will continue using the Northwind Orders report for these examples.



1. Write a function that concatenates two strings together and returns the results.
2. Use the calculated column in a report.

Informer™
Quick and Easy Business Intelligence

Appendix A: JavaScript Exercise Solutions

JavaScript Exercise Solutions

Chapter 3

1. Calculated Column to display "Hello World"

```
"Hello World"
```

2. Calculated Column to display value of Company Name

-drag column alias into Script area

```
product_assoc_supplier_assoc_CompanyName
```

3. Calculated Column of order total for each line

```
var lineTotal = 0; // Declares our variable
lineTotal = Quantity*UnitPrice; //Performs the calculation
lineTotal; //Displays the results
```

Switch Statements

1. "Backordered" if Quantity exceeds Units in Stock

```
if(Quantity>product_assoc_UnitsInStock)
  "Backordered";
else
  "";
```

Loops

1. Sum of numbers 1 to 50

```
var x=0;
for (var i=0; i <=50; i++)
{
  x = x+i;
}
x;
```

2. Sum of numbers 1 to 50, but stops when the sum reaches 50

```
var x =0
for (var i=0; x<= 50; i++)
{
    x +=i;
}
x;
```

Do While Statement

```
var x = 0;
var i =1;
do
{
   x+=i;
   i++;
}
while (x <= 50);
x;
```

While Statement

```
var x = 0;
var i = 1;
while (x <=50)
{
   x += i;
   i++;
}
x;
```

3. Calculated Column of the Odd numbers (2 Solutions)

```
var x=0;
for (var i=1; i <=50; i++)
if (i%2 != 0)
{
  x = x+i;
}
x;
```

2nd Solution

```
var x=0;
for (var i=1; i <=50; i+=2)
  x = x+i;
x;
```

Chapter 4

1. Number of Characters in the Last Name field

```
employee_assoc_LastName.length;
```

2. Concatenate first and last name together in lastname, firstname format

```
employee_assoc_LastName + "," + employee_assoc_FirstName;
```

3. Calculated column that displays the first initial, a period, a space, and first 3 of the last name

```
employee_assoc_FirstName.substr(0,1) + ". "+ employee_assoc_LastName.substr(0,3)
```

4. Customer name to all upper case

```
customer_assoc_CompanyName.toUpperCase();
```

Challenge - Last name field in upper case for odd numbered positions

```
var xResult = "";
for (var i=0; i<lengthOfLastName; i++)
{
  if((i%2) ==0)
  xResult += employee_assoc_LastName[i].toUpperCase();
else
  xResult += employee_assoc_LastName[i].toLowerCase();
}
xResult;
```

Dates

1. Pull Month, Day and 4-Digit Year out of date

```
OrderDate.getMonth();
OrderDate.getDate();
OrderDate.getYear() + 1900; //add 1900 to get the 4 digit year
```

2. Use the Simple Date Format to format date as 12062012 (month,day, year without delimiter)

```
var sdf = new java.text.SimpleDateFormat("MMddyyyy");
sdf.format(today);
```

3. Calculated column that displays "Late Shipment" or "On Time"

```
if (ShippedDate.after(RequiredDate))
   "Late Shipment";
else
   "On time";
```

4. Calculated column to display number of days between order date and ship date (fulfillment date)

```
var orderDate = java.util.Calendar.getInstance();
orderDate.setTime(OrderDate);
orderDate.set(java.util.Calendar.HOUR, 0);
orderDate.set(java.util.Calendar.MINUTE, 0);
orderDate.set(java.util.Calendar.SECOND, 0);
orderDate.set(java.util.Calendar.MILLISECOND, 0);

var shipDate = java.util.Calendar.getInstance();
shipDate.setTime(ShippedDate);
shipDate.set(java.util.Calendar.HOUR, 0);
shipDate.set(java.util.Calendar.MINUTE, 0);
shipDate.set(java.util.Calendar.SECOND, 0);
shipDate.set(java.util.Calendar.MILLISECOND, 0);

(shipDate.getTimeInMillis() - orderDate.getTimeInMillis()) / (1000*60*60*24);
```

Arrays

1. Number of items in an order (or number of items in an array)

```
orderDetailsRemote_assoc_ProductID.length
```

2. Calculated column that displays a message if one of the quantities is above 70

```
var message = ""
for (var i = 0; i < orderDetailsRemote_assoc_Quantity.length; i++)
{
  if (orderDetailsRemote_assoc_Quantity[i] > 70)
  { message = "Yes";
  break;
  }
}
message;
```

3. Calculated Column that creates an array of the extended price of each line item

Using Java Version 1.6

```
var extPrice = java.lang.reflect.Array.newInstance(java.lang.String,
orderDetailsRemote_assoc_Quantity.length);
for (var i = 0; i < orderDetailsRemote_assoc_Quantity.length; i++)
{
  extPrice[i] =  orderDetailsRemote_assoc_Quantity[i] * orderDetailsRemote_assoc_UnitPrice[i];
}
extPrice;
```

Using Java Version 1.7

```
var extPrice = new Array();
for (var i = 0; i < orderDetailsRemote_assoc_Quantity.length; i++)
{
  extPrice[i] =  orderDetailsRemote_assoc_Quantity[i] * orderDetailsRemote_assoc_UnitPrice[i];
}
extPrice;
```

4. Calculated column that adds the line extended price to give an order total

```
var orderTotal = 0;
for (var i=0; i< extendedPrice.length; i++)
{
   orderTotal += extendedPrice[i];
}
orderTotal;
```