Custom Functions

by John Mark Osborne

If you have FileMaker Advanced, you can create your own FileMaker calculation functions complete with function name and parameters using Custom Functions. The great thing about Custom Functions is you don't need to know C++. You simply use the FileMaker functions you are already familiar with to create your own named function. While you need FileMaker Advanced to create, edit or delete custom functions, anyone with a regular copy of FileMaker Pro can take advantage of them. But, what advantages do Custom Functions have over standard calculation formulas?

In general, there are four distinct advantages to be gained by using Custom Functions. The first and most common reason is to reuse code. Reusing code allows you easily to access a complex formula with a simple function name call much like the existing calculation functions that ship with FileMaker. If you need to change a Custom Function formula, the code is also stored in a central location so the changes flow out to every place you have referenced the Custom Function. That means you can update a Custom Function and any Script, Calculation Field, Conditional Formatting or Auto-Enter Calculation that references it will be updated with the new code.

A second reason to use Custom Functions is for system constants. For example, you could define sales tax for your state and when it changes, make the change in a single location. Wherever the Custom Function is referenced in your FileMaker solution, it will be updated with the new sales tax. This is very similar to reusing code except that there is no formula, just a constant value.

One of my favorite reasons for using Custom Functions is recursion. You can compare recursion to a looping script inside of a calculation. The calculation formula keeps calling itself over and over until it exits, all along building a complex result. Understanding recursion is probably the most difficult Custom Function concept so we'll explain by example in the upcoming sections.

The last advantage of Custom Functions is the ability to hide proprietary formulas from junior programmers. Concealing the family jewels inside a Custom Function will allow another programmer to access the function but not see the actual code. As long as you control what version of FileMaker your employees use, you can hide the genius behind your Custom Functions.

Since system constants and hiding proprietary code are self-evident in the descriptions above, we are going to concentrate on complex code reuse and recursion in the following examples.

A Simple Custom Function

Let's say you want a Custom Function for calculating the commission for sales people. and the formula looks like the following:

```
Round(MyTotalField * .08; 2)
```

While this is not a very complicated formula, it is possible you might need to use this formula in several different areas of your solution. If the commission rate changes, you will need to remember everywhere you used the formula and update it manually. If you are like me, I tend to forget every single place I used a piece of code. If you use a Custom Function instead, the code is stored in a central location so you can update it in a single place.

To define a Custom Function, all you need to do is copy and paste your existing code into a new Custom Function or type it in from scratch as you would any calculation formula. In addition, you will need to give your Custom Function a unique name and possibly define some parameters. In the example above, one parameter will need to be defined to substitute for the field reference. You might decide to name your Custom Function "Commission" and your parameter "Amount". Once this is done, your function call to the Custom Function will look like the following in your calculation dialog:

```
Commission(Amount)
```

Determining Age

While updating code is probably one of the most important advantages of Custom Functions, simplifying complex formulas is surely close behind. I like to use an example of an Age calculation that returns the age in years, months and days based on a date of birth field:

```
GetAsNumber(Year(Get(CurrentDate)) -
Year(DOB) - Case(Get(CurrentDate) <
Date(Month(DOB); Day(DOB); Year(Get(CurrentDate))); 1; 0)) & "
Years, "

&

GetAsNumber(Mod(Month(Get(CurrentDate)) -
Month(DOB) + 12 - Case(Day(Get(CurrentDate)) <
Day(DOB); 1; 0); 12)) & " Months, "

&

GetAsNumber(Day(Get(CurrentDate)) -
Day(DOB) + Case(Day(Get(CurrentDate)) >=
Day(DOB); 0; Day(Get(CurrentDate) -
Day(Get(CurrentDate))) <
Day(DOB); Day(DOB); Day(Get(CurrentDate) -
Day(Get(CurrentDate))))) & " Days"
```

Let's start by simplifying this formula with the Let function:

```
Let(

[@CurrentDate = Get(CurrentDate);
@DOB = DOB];

GetAsNumber(Year(@CurrentDate) -
Year(@DOB) - Case(@CurrentDate <
Date(Month(@DOB); Day(@DOB); Year(@CurrentDate)); 1; 0)) & "
Years, "

&

GetAsNumber(Mod(Month(@CurrentDate) -
Month(@DOB) + 12 - Case(Day(@CurrentDate) <
Day(@DOB); 1; 0); 12)) & " Months, "

&

GetAsNumber(Day(@CurrentDate) -
Day(@DOB) + Case(Day(@CurrentDate) >=
Day(@DOB); 0; Day(@CurrentDate -
Day(@CurrentDate)) <
Day(@DOB); Day(@DOB); Day(@CurrentDate -
Day(@CurrentDate)))) & " Days"

)
```

You can see how much easier the formula is to update a formula with the Let function if you move it to another solution where the date of birth field might have a different name. But, what if you have to change some of the code because you found a mistake. It would be much easier to reuse the code in a Custom Function so you can update it in a central location rather than trying to locate every instance of the code.

In addition, a Custom Function can make it easier to reference a large piece of code. The concept is similar to modular scripting where long scripts are divided into modules and connected with the Perform Script step. You can read the Custom Function name inside the calculation formula and get a pretty good idea what it does.

Once you place the Age calculation in a Custom Function, the call to the formula might look like this:

```
Age(BirthDate)
```

The Custom Function is almost like a chapter title for a reference book. You can decide if you want to see the code by visiting the Custom Function dialog or resign yourself to just looking at the reference to the code. If you have good names for your Custom Functions, you will likely never need to look at the actual code again. And, with FileMaker 11 Advanced, you can now import or copy/paste Custom Functions from one file to another!

It's important to remember when updating a Custom Function that it will update the live calculations on existing records like Calculation Fields, Record Level Access formulas and Conditional Formatting conditions, just to name a few. If you want existing records to keep their original values, make sure Custom Function calls are placed in an Auto-Enter calculation or a Script.

Basic Recursion

Let's start with a simple example of recursion that is unlikely to inspire you. However, it is important not to skip this simple example because the code will be used as the basis for all other recursive Custom Functions in this article. So, here's what we are going to do. We want to create a Custom Function that repeats the contents of a field as many times as we specify. For example, if the name "John" is contained in a field and we ask it to repeat five times, the result will look like this:

```
John John John John John
```

Start by designing a new Custom Function called "RepeatText" with two parameters called "Text" and "Repeat". The Custom Function will contain the following formula:

```
Case(

Repeat > 0;

Text & " " & RepeatText(Text; Repeat - 1)

)
```

The call to the Custom Function to repeat the contents of the "Name" field five times will look like this:

```
RepeatText(Name; 5)
```

So, how does this recursion stuff work? It's actually quite simple. All you need is a conditional statement like Case or If. Inside the conditional statement, you test a scenario. When the test is true, call the Custom Function again. If the test is false, exit the Custom Function and return the result.

In our example, the test decides if the Repeat parameter is greater than zero. The first time the Custom Function loops, the value of the Repeat parameter is the number five. Since five is greater than zero, the condition results in a call back to itself but with one minor change; the Repeat parameter is reduced by one. Think of it as a counter in a looping script. Once the counter reduces enough to make the conditional statement false, the recursion ends and a result is returned.

As the recursive Custom Function is looping, it builds up a result in the memory stack. Each time the Custom Function loops, it concatenates the contents of the Text parameter with a space. It keeps adding to the result stack each time it loops, resulting in my name being repeated five times, separated by a space.

If you test this Custom Function in a FileMaker solution and take a careful look at the result, you'll notice a space at the end of the string. Think about it. That's what you asked it to do. But, that space could screw up your solution. Here's how you workaround the issue:

```
Case(

Repeat > 1;

Text & " " & RepeatText(Text; Repeat - 1);

Text

)
```

All we did was change the comparison test from a zero to a one and include a default or false result. So, the recursion happens one less time but the default result allows for the last concatenation of the text to be repeated minus the space.

Phone Formatting

Now that you understand the basics of recursion, let's do something useful with recursive Custom Functions. Formatting a phone number field has always been a common solution in order for the formatting to be consistent across all records. Back in the FileMaker 6 days, developers used a calculation field layered on top of a data entry field to format a field. With FileMaker 7, you could eliminate the extra field using an auto-enter calculation with the option to "do not replace existing value of field (if any)" unchecked. Even better is a recursive Custom Function since it can handle any length of phone number whether it is from the United States or the Netherlands.

Let's start right off with the formula:

```
Let(
[@NumbersOnly = Filter(Phone; "0123456789");
@NewNumber = Right(@NumbersOnly; Length(@NumbersOnly) - 1);
@NewFormat = Right(Format; Length(Format) - 1)];

Case(

not IsEmpty(@NumbersOnly);

Case(
Left(Format; 1) = "#";
Left(@NumbersOnly; 1) & PhoneFormat(@NewNumber; @NewFormat);
Left(Format; 1) & PhoneFormat(@NumbersOnly; @NewFormat)
)

)

)
```

Although this formula is quite a bit more complicated than the basic example covered earlier, it shares the same basic technique of a conditional statement to control the recursion. The name of the function is "PhoneFormat" and the two parameters are "Phone" and "Format". The first parameter holds the phone number that was entered into a field and the second parameter holds the format for the phone number. Phone formats are passed along in the parameter in the following fashion:

```
(###) ###-####
```

Pound signs represent where the digits are substituted and anything else is a separator which stays the same. So, a call to this Custom Function might look like the following:

```
PhoneFormat(Self; "(###) ###-####")
```

The first declaration in the Let function removes all non-number values using the Filter function. The second declaration removes the leftmost value from the Phone parameter and the third declaration removes the leftmost value from the the Format parameter. When we get down to the conditional statement, you'll understand why this is necessary.

The Case statement starts by testing whether the @NumbersOnly declaration is empty or not. Each time the formula recurses, one value is removed from the phone number till it finally becomes empty. In other words, the formula works through the phone number one digit at a time until there are none left. Actually, it works through the format value one character at a time, as you will see, but both are reduced in length each time the recursion occurs.

If the phone number still contains values, another Case statement is invoked, determining whether the value in the format is a pound sign or other character. If the value is a pound sign, the leftmost number is grabbed and placed in the memory stack to await the final result and the PhoneFormat function calls itself again minus one value from the phone number and the format string. If the leftmost value contains a character other than a pound sign, the leftmost formatting character is placed in the memory stack and a similar recursive call is made which removes one formatting character but keeps the phone number the same.

By the way, you should enter the call to the Custom Function on the field you want to format as an auto-enter calculation with the option to "do not replace existing value of field (if any)" unchecked. You might also want to add a validation calculation that checks the length of the numbers in the phone number against the number of pound signs in the format string to make sure the Custom Function can properly format the phone number.

Custom Function Gotchas

If you create an endless loop in a script, you have the option of canceling that script if you haven't turned off Allow User Abort. If you create an endless loop in a recursive Custom Function, FileMaker will automatically timeout after 10,000 recursions to prevent an endless loop. While this is a good thing in general, it can also be a significant limit in some recursive scenarios such as collecting a return-separated list of serial numbers for use in a multi-key relationship. In this case, you can increase the limit to 50,000 recursions by using tail recursion which avoids using the memory stack to hold the result. Here's an example of tail recursion using our original basic recursion example:

```
Case(

Repeat > 1;

RepeatTextTail(
Text;
Repeat - 1;
Text & " " & Result
);

Result & Text

)
```

The call to this Custom Function might look like the following:

```
RepeatTextTail(Name; 5; "")
```

The first thing you want to notice is that there is a third parameter. This is where the result is built in order to avoid the memory stack limits. The reason tail recursion is limited to 50,000 recursions is because of the memory stack. While one recursive call is limited to 10,000 recursions, you can have up to five separate recursive calls for a total of 50,000 recursions.

Conclusion

The Custom Function examples in this article only brushed the surface of this topic but hopefully gave you a good idea of how important a copy of FileMaker Advanced can be to the efficiency of the development process. While no additional examples are likely needed to understand standard Custom Functions and reusing code, recursive Custom Functions open up a whole new way of working with calculations. For more examples of Custom Functions and especially recursive Custom Functions, take a look at the Custom Function library at:

http://www.briandunning.com/

John Mark Osborne is president and owner of Database Pros <www.databasepros.com>, offering the largest free FileMaker resource on the internet, training classes, commercial solutions and development services. John Mark is internationally recognized as the co-author of Scriptology, a speaker at the FileMaker Developer Conference and MacWorld conferences and an authorized trainer for the FileMaker Training Series developed by FileMaker, Inc. John Mark is certified for FileMaker 7, 8, 9, 10 and 11, having passed all five rigorous tests.