NBOS Character Sheet Designer

Table of Contents

1 Introduction

The NBOS Character Sheet system is designed to provide gamers a game system-independent tool with which they can create customizable Character Sheets.

There are two programs that make up NBOS's Character Sheet system. One's a Character Sheet Designer, and the other a Character Sheet Viewer.

The designer is used to draw out the actual character sheet. That is, the lines, text, and fields on the character sheet itself. With the designer you'll draw out the character sheet, add the fields that the user can fill out, and add in the calculations that you want automated on the character sheet.

The viewer is what players will use to fill out character sheets created by the designer. The viewer allows players to fill out the sheet with their character's information (name, stats, skills, etc), and save/load their characters to/from files.

There are two types of files - one defines the character sheet (what stats can be filled out, skill lists, calculations), and one holds the information about a character (what the actual stats are, which skills the character has).

What this allows is for the character sheet definition to change without requiring the player to copy their character data over into a new character sheet (because the character data is stored in a separate file).

The program is designed for any game system or use. Its very generic. But keep in mind, completely duplicating a system's rules can be a serious undertaking!

2 What's New in v2.0

The big change you'll notice in this new version is that updates after a field is changed are now significantly faster, particularly on very complex character sheets. The reason for this is the Character Sheet Viewer has a new, built-in expression engine to handle the calculations, and will only use JavaScript when told explicitly to run JavaScript code.

Along the same lines, when defining the source for a field, you no longer need to reference the field using dot-notation. You can now use just the field name in formulas, simplifying them. So instead of `DexMod.Value + BaseAttack.Value`, your formula would look like `DexMode + BaseAttack`.

In the interface, there's a new field list on the left side of the designer window to let you easily access fields regardless of the page they are on.

JavaScript runtime errors in the viewer provide more information in the Debug Mode window, including line and position of the error

3 The Process

The character sheet process works this way:

You create a character sheet definition with the designer, save it to a definition file (*.csd) file, and send it to your players.

The players use File-Open to open the .csd file in the viewer. When they do this, the viewer adds the character sheet definition to the list of available character sheets in the viewer. (The process is essentially copying the file over into a predefined directory.)

The player then uses File-New in the viewer. The list of installed character sheet definitions is displayed, and the player picks which type of character they want to make. This provides a blank character sheet for the player to fill out.

The player saves the character sheet. This creates a data file *separate* from the character sheet definition with a *.character extension. This file can be loaded and saved whenever the character is edited. Once a character sheet definition is 'installed', the player can make as many characters as they want, saving each to a separate .character file.

If you update the character sheet definition in the designer, you can send that update to the players (using the same file name as before), and they can install it as above. The next time they open their character, the character sheet is updated with the new functionality.

4 What the program is not

The Character Sheet Designer and Viewer are not intended to be 'game managers'. They aren't tools designed to automate GM'ing or simulate game rules for at table play.   They have a very limited scope - to produce character sheets that players can fill out and print.

The technology used by the Character Sheet Designer and Viewers is also used in The Keep, a Personal Information Manager for gamers, to allow the application to display and print character sheets.

5 Designing Character Sheets

5.1 Fields

What can the designer do?

The designer allows you to draw out a character sheet. Much like Fractal Mapper, it has a set of drawing tools you can use to design your character sheet.

On your character sheet you can add shapes and text. You can also add *fields*, which are areas on the character sheets that your players will use when filling out the character sheet.

Shapes

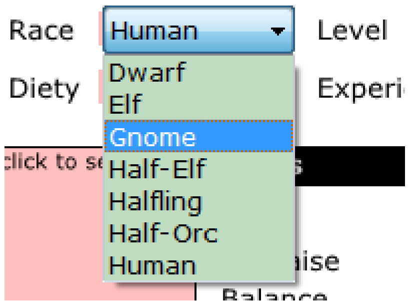
The shapes are straight forward. Lines are just lines. Orthogonal lines are lines that are either perfectly vertical or horizontal. Boxes are rectangles. Orthogonal lines and boxes know what they are, so when you adjust them, they will adjust accordingly.

Text edit fields

Text edits are fields that the user will be able to edit in the viewer. They draw out like a rectangle, and the highlighted area shows how the font size fits into the rectangle. In the viewer, the user will be able to enter text into the text edit field.

Text edits can allow either a single-line of text, or multiple lines of text. By default it will be a single line of text. But to allow multiple lines for such things as item descriptions, select the field in the designer and select the multi-line option ≣ on the toolbar. (If a field does not display in multi-line mode, make sure it is tall enough to contain at least one line of text).

Text edits also can double as a 'drop down' box. If you assign a 'list' to the text edit, then the end user will instead have a pick list to choose from. Use Sheet-List manager from the menu to create and define lists, and the list drop down on the toolbar to assign them to a text edit. Lists can thus be used by more than one text edit.



Label fields

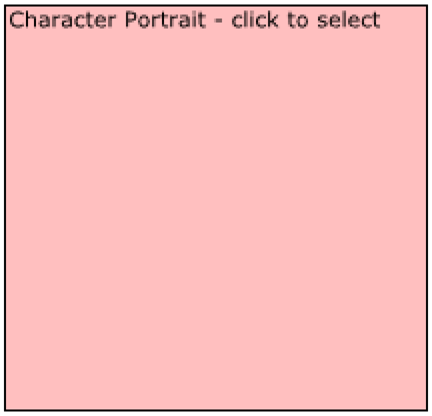
Labels are plain text. Label fields *cannot be directly edited* by the end user in the viewer. They can, though be changed via calculations.

Checkbox Fields

Checkbox fields create a little box the end user can click to toggle an X in the box.

Image Select Fields

Image select fields allow the end user to select an image to place into the character sheet. For example, as a character portrait. In the viewer, the user just clicks on the image select field, and is prompted to select an image. When an image is selected, its placed into the box on the character sheet.



An Image Select Field waiting for the player to pick a picture

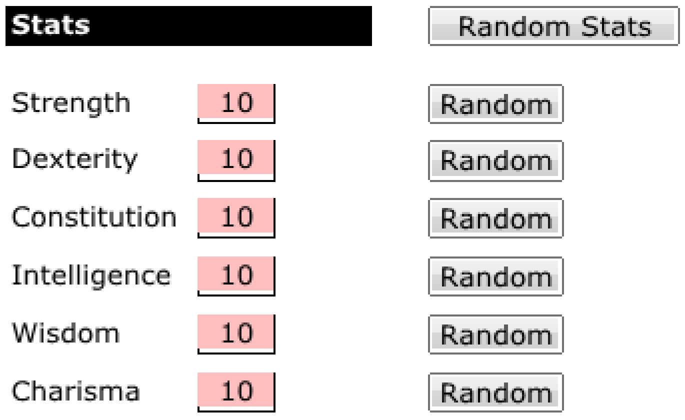
5.2 Images

You can place (non-editable) images into the character sheet by using Object-Insert Image from the menu. You can display the image in grayscale for easier printing by selecting Object-Grayscale Image.

Don't confuse this sort of image with the Image Select Tool, which allows the player to add an image to the character sheet themselves.

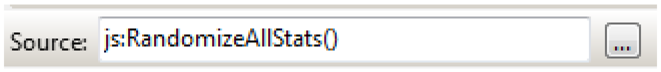
5.3 Buttons

Buttons can also be placed on character sheets. Buttons are a special type of field that can be used to trigger a JavaScript function when clicked. Buttons themselves are visible in the Character Sheet Viewer, but are not printed.



To make a button trigger a JavaScript function when pressed, assign the JavaScript function call as it would be used in code to the button's Field Source, prefixed with a "js:" (as you would for JavaScript calls in other fields). You can also assign expressions and other inline

JavaScript to the field source.


Source: js:RandomizeAllStats()

5.4 Multiple Pages

The designer and viewer support character sheets consisting of multiple pages. To add a new page, use Sheet-New Page in the designer's menu to add a new page.

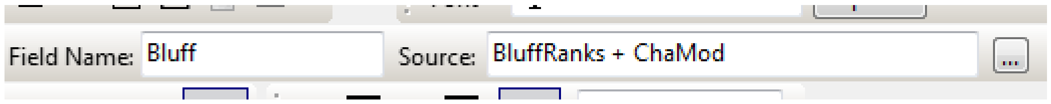
Nothing special needs to be done to reference fields that reside on separate pages in their calculations 5. Just reference the fields as normal.

5.5 Calculations

Calculations, of course, are what make character sheets truly helpful. By assigning calculations to fields you can have the character sheet automatically make the necessary changes to related values when a field changes.

Text Edits and Label fields can have what's called a 'Field Source'. If you've used spreadsheets before, think of the field source as being like a cell formula you'd find in a spreadsheet. Whenever a field changes on your character sheet, the viewer uses the field sources to re-calculate all the dependencies for that value. For example, when a Strength trait changes, a character sheet might then apply bonuses and penalties based on that trait to skills and attributes.

To assign a field source to a field, select the field in the designer and enter your formula into the Source box on the toolbar.


Field Name: Bluff Source: BluffRanks + ChaMod

In formulas you can use the Field Names of other fields as part of the equation. So if you want to calculate a Bluff skill, for example, you might provide a formula that adds values of the BluffRanks field and ChaMod field. Whenever the value for either BluffRanks or ChaMod changes, the total for the Bluff skill would be recalculated.

Lets look at another example. Say you have two text edits for stats. One is named 'Strength', and one 'Constitution'. If you wanted to create a 'hit point' field that was some calculation of the two, you'd create a text edit or label called 'hits', and assign a fields source of 'Strength + Constitution'. Now, whenever the Strength or Constitution field changes, the Hits field is updated with the new sum.

You can also reference fields using dot notation if you want to make distinction between their values being text and numbers. To do so, reference a field as 'Field.Value' for the numeric value of the field, and 'Field.Text' for the text value. Normally you would not have to do this, but there may be case where you want numbers to be treated as text.

Note that any text edit that has a 'field source' defined for it cannot be edited directly by the end user in the viewer. That is, the end user cannot tab into the field and edit it.

5.6 Function List

When creating formulas to use as Field Sources, there are a number of built-in functions you can use in your calculations. These are not JavaScript functions, and are not available within JavaScript code.

5.6.1 Abs

Syntax

abs(n: number)

Description

This returns the *absolute value* of a number. That is, the value of a number without a sign. The value returned is always 0 or greater.

Example:

```
abs( -25)
```

would return

```
25
```

5.6.2 Ceil

Syntax

ceil(n: number)

Description

The ceil function rounds a number *up* towards infinity.

Example:

```
ceil( 5.3)
ceil( -5.3)
```

would return

```
6
-5
```

5.6.3 Floor

Syntax

floor(n: number)

Description

The floor function rounds a number *down* towards negative infinity. That is, the nearest round number below the value.

Example:

```
floor( 5.3)
floor( -5.3)
```

would return

```
5
-6
```

5.6.4 If

Syntax

if(condition: boolean, then: any, else: any)

Description

The if function is used to return different values depending on whether the condition is true or false. The function returns the value *then* if the condition evaluates to true, otherwise it returns *else*.

Example

```
if( Strength > 10, 2, 0)
```

If the field Strength contained a value greater than 10, the function would return

```
2
```

otherwise it would return

```
0
```

5.6.5 Length

Syntax

length(s: text)

Description

Returns the length, in characters, of the text sent as a parameter.

Example:

```
length( "Wizard")
```

would return

```
6
```

5.6.6 Max

Syntax

max(n1, ... nN: number)

Description

The max() function returns the largest value passed as a parameter. This function requires one or more numbers passed as parameters. Any number of parameters can be passed, separated by commas.

Example:

```
max( 1, 3, 5, 7, 9, 7, 5, 3, 1)
```

would return

```
9
```

5.6.7 Min

Syntax

min(n1, ... nN: number)

Description

The min() function returns the smallest number passed as a parameter. This function requires one or more numbers passed as parameters. Any number of parameters can be passed, separated by commas.

Example:

```
max( 1, 3, 5, 7, 9, 7, 5, 3, 1)
```

would return

```
1
```

5.6.8 Round

Syntax

```
round(n: number, decimals: number)
```

Description

The round function rounds a number to the specified number of decimal places. To round to 10's above zero, use a negative for the decimal parameter.

Example:

```
round( 3.14159, 2)
round( 3.14159, 1)
round( 3.14158, 0)
round( 1234567, -2)
```

would return:

```
3.14
3.1
3
1234500
```

5.6.9 Sign

Syntax

```
sign(n: number)
```

Description

The sign function returns 1 if *n* is a positive number, -1 if *n* is a negative number, and 0 if *n* is zero.

Example:

```
sign( 20.2)
sign( -8)
sign( 0)
```

would return

```
1
-1
0
```

5.6.10 Sqrt

Syntax

sqrt(n: number)

Description

The sqrt function returns the *square root* of a number.

Example:

```
sqrt( 144)
Sqrt( 9)
Sqrt( 10)
```

would return

```
12
3
3.1622776
```

5.6.11 Substr

Syntax

substr(s: text, start: number, count: number)

Description

Substr is used to return part of a piece of text, where *s* is the full text, *start* is the offset in characters (starting at 1), and *count* is the number of characters to return.

Example:

```
substr( "chain maile", 3, 3)
substr( "chain maile", 1, 5)
```

would return

```
"ain"
"chain"
```

5.6.12 Trim

Syntax

trim(s: text)

Description

The trim function removes leading and trailing spaces and tabs from text.

Example:

```
trim( "  Chain Maile  ")
```

would return

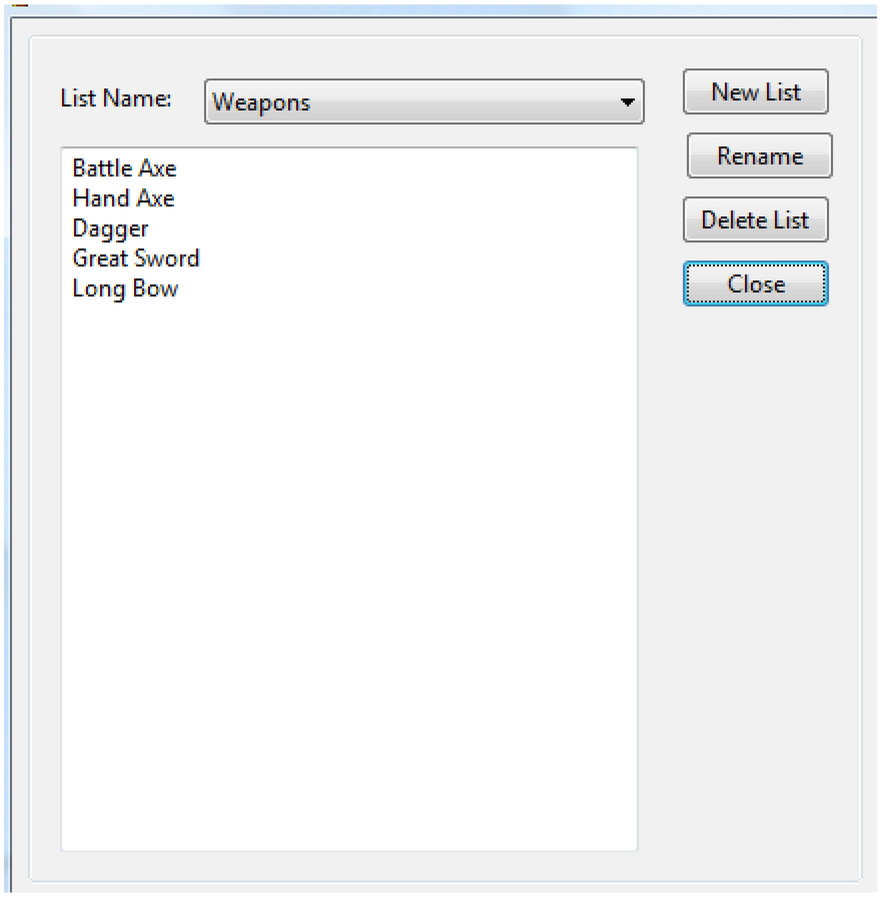
```
 "Chain Mail"
```

5.7 Lists

Most game systems contain many types of lists - races, classes, weapons, spells, etc. The Character Sheet designer allows you to maintain individual lists of these items which can then be assigned to text fields for use as pick-lists.

To create a list of items, open up the List Manager (Sheet - List Manager from the menu). This will display the current lists available within the character sheet definition.

To create a new list, click the New List button, and enter the name of the list to create.

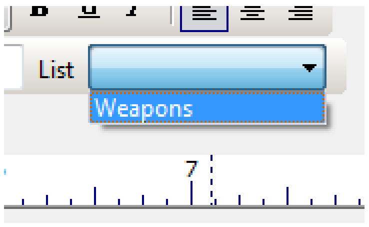


Battle Axe
Hand Axe
Dagger
Great Sword
Long Bow

Enter in, one per line, the items in your list.

Finally, click the Close button to close the list manager.

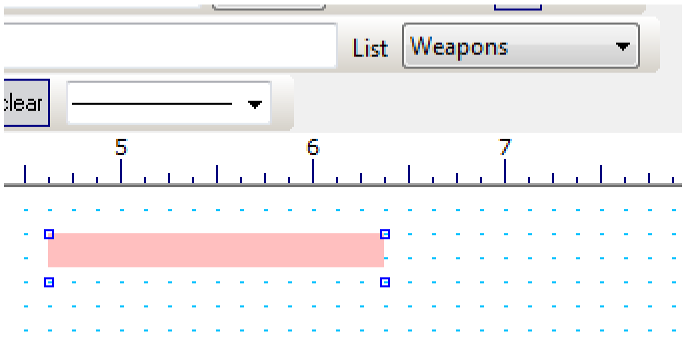
The list you just created will now be available in the List box on the toolbar.



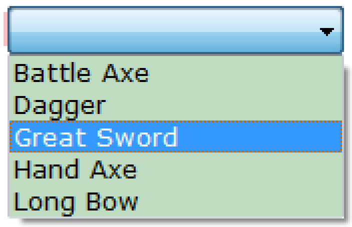
Once the list is created, you can then use it to create a pick-list field.

To do this, first place a text edit field on your character sheet.

Then, with the edit field selected, select the name of your list from the List box.



Now when the character sheet is edited with the viewer, the field will be displayed as a pick-list.



Pick lists can be either 'Open' or 'Fixed', depending on which option on the toolbar you have selected. Fixed pick-lists are limited to the items in the list. Open lists allow for the user to either select an item from the list, or enter their own text.

5.8 Field Options

There's several field options on the toolbar that should be noted:

Numeric Display
The 'Num' button on the toolbar toggles the numeric display for a text edit or label field. What this does is force the field to show only numbers. If the Text of the field is not a number, 0 is displayed.

Read Only fields
The 'Read Only' button makes a text edit field read-only in the viewer. That is, the end user will not be able to edit it, even if it lacks a field source. You won't need this much unless you're doing a lot of javascript in the events, and assigning values to the fields in your scripts.

Hidden Fields
The 'Hidden' button hides/shows an object. The most useful case for this would be to have a field you can reference for some information (some intermediary value used during calculations, for example). But you can also toggle the field's display by setting its Visible attribute to true or false via javascript.

Single / Multi-line Fields
The 'Single Line/Multi-Line'. buttons control how text edits act. If single line is selected, then the end user can only enter a single line of text into the text edit. If multi-line is selected, the end user can enter multiple lines of text, and the field will adjust taller as necessary.

5.9 JavaScript

You can create and access JavaScript functions as part of your character sheet definitions. At the bottom of the designer, where the page tabs are located, there's also Script tab. When selected, a code editor is displayed which allows you to enter JavaScript functions.

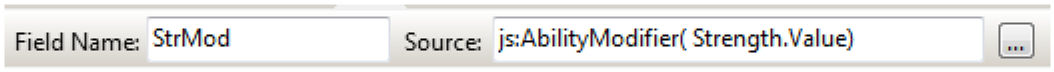
For example you can create a function that calculates a d20 ability modify based on an ability score:

```
function AbilityModifier( n)
{
   return Math.floor( (n-10) /2);
}
```

JavaScript functions created this way can then be used as the Field Source for a field. To do so, prefix the field source with 'js:'.

So you might assign a field source such as:

```
js:AbilityModifier( Strength.Value)
```



Field Name: StrMod Source: js:AbilityModifier(Strength.Value)

When that field is updated, it calls the AbilityModifier() function with the value of the Strength field as a parameter, and then sets the field text to the returned value.

Note that when using JavaScript to reference a field, you must use the dot-notation method. Thus, to reference the numeric value of a field, use Field.Value, and to reference the text value of a field, use Field.Text. So, for example, to reference the numeric value of a field named Strength, you'd use Strength.Value in your JavaScript.

You can also access a field directly in a javascript function, and even assign a value to a field, like:

```
function AdjustHits()
{
   Hits.Text = Strength.Value + Constitution.Value;
}
```

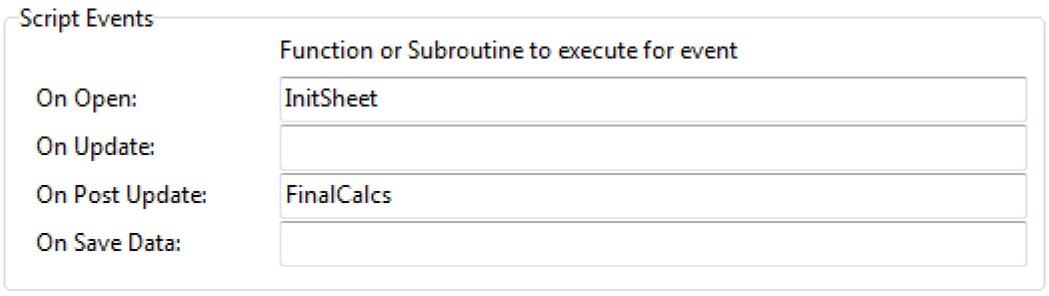
When assigning a value to a field, you can only assign a value to the Text of a field. The Value property is read-only.

External JavaScript files can be used to store oft used functions. See the document setup window for where you set that up.

5.10 JavaScript Events

You can have the viewer program execute Javascript functions when certain events occur, such as when the character sheet is opened, or when the fields are about to be updated.

On the document setup window, on the script tab, you can enter javascript function calls to run on certain events. Enter just the name of the Javascript function. *Do not include braces or parameters*.



On Open:	InitSheet
On Update:	
On Post Update:	FinalCalcs
On Save Data:	

Event Descriptions:

On Open
This occurs when the character sheet is loaded by the viewer.

On Update
This occurs after a field has been changed in the viewer, but before the process of running through all the field formulas/calculations.

On Post Update
This occurs after all the field formulas/calculations are processed.

On Save Data
This occurs right after the user presses 'save' in the viewer, but before the file is actually written to disk.

For large character sheet projects, one optimization trick is to place a lot of the calculations behind the OnUpdate event, rather than assigning field sources to many individual fields.

Example:
In the image above, a JavaScript function named InitSheet is assigned to the On Open event. In the JavaScript editor, that function might look like:

```
function InitSheet()
{
   //set the default class to Fighter
   CharacterClass.Text = 'Fighter';  //references a field name
CharacterClass.
}
```

5.11 Editing Character Sheets

You'll see an item in the menu called Edit Installed Sheet. This will let you quickly access any installed character sheets to edit their definitions. 'Installed' means that the character sheet is located in the proper place so that the definition is available to the viewer.

There's also a Save and Install menu item, which will save the definition currently being edited into the proper directory so that the viewer will consider it installed.

Once a character sheet definition (*.csd) is 'installed' by the viewer, its copied to a location in the the user's Windows Application Data directory.

On XP, this is usually:

```
C:\Documents and Settings\<Name>\My Documents\NBOS Character Sheets
```

On Windows 7, it's

```
C:\Users\<Name>\Documents\NBOS Character Sheets
```

5.12 DataStores (Lookup Tables)

The Character Sheet designer provides a means of accessing tabular data that game rules tend to rely heavily on. By using these *DataStores*, you can populate fields from tabular data

The tabular data for DataStores is stored in comma separated text files (a.k.a., .csv files). You can maintain many different .csv files. When added to your character sheet definition, the data from the .csv files is embedded into the definition file as an available DataStore.

The .csv files are re-embedded every time you save the definition file - so if you change the contents of the .csv file, re-saving the definition will update the DataStores in the character sheet definition.

In addition to being able to access the tabular data in field sources, each .csv DataStore also automatically becomes available as a list for drop downs with the same name as the .csv file. So once a .csv file is added, the items in the key field for that .csv can be accessed in the same way as manually added list items. For example, if you add 'weapons.csv' as a DataStore, a list named 'weapons' is automatically made available.

Here's how you go about using DataStores.

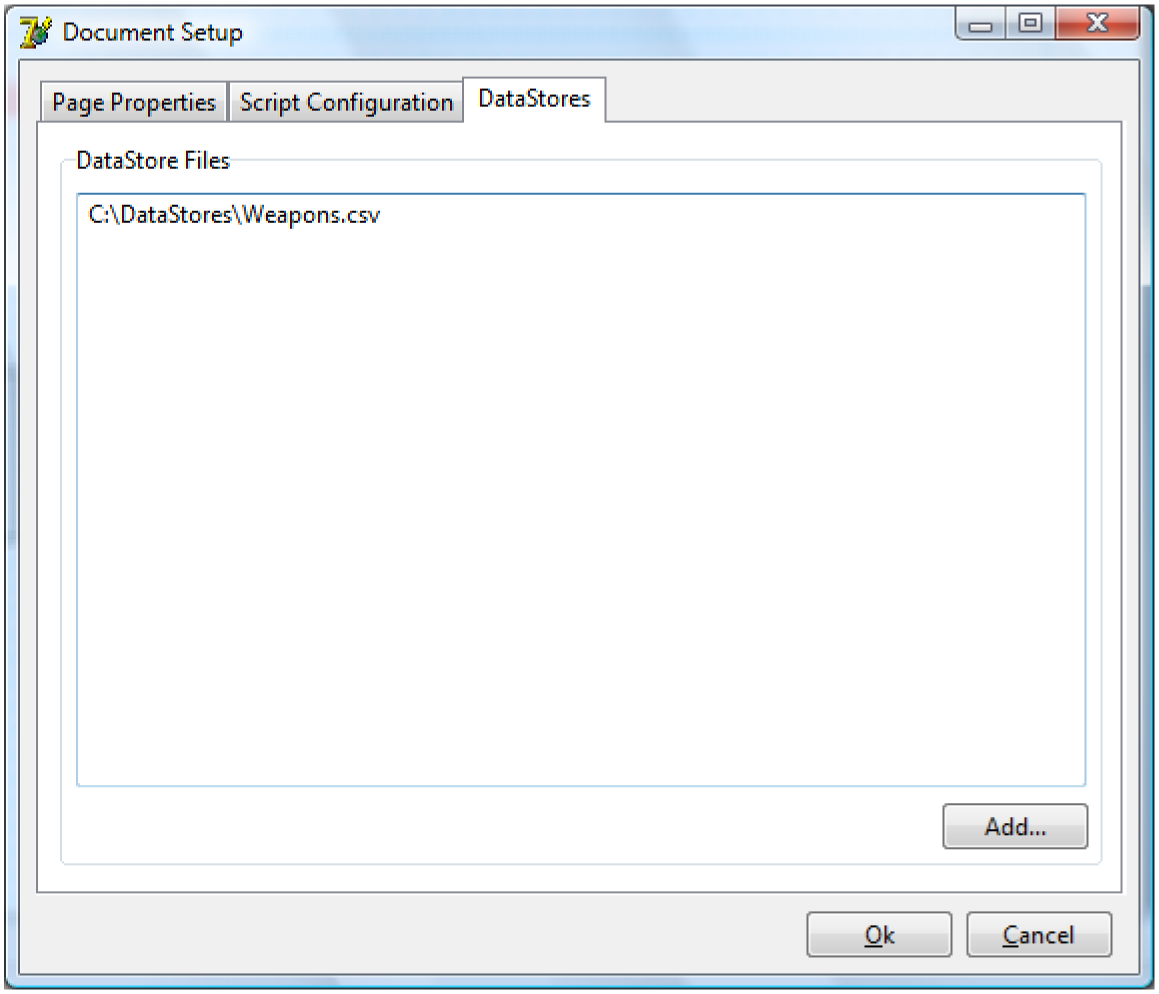
First, create a .csv file of your game rules data. The easiest way to do this is to fire up Excel or another spreadsheet, enter the data in, and save it as a .csv file. Or you can edit them with a text editor if you are comfortable doing that.

The structure of the .csv file is pretty simple. On the first line of the file, list the names of the fields, each separated by commas. Then, on each subsequent line, resides the comma separated data. The first field listed is considered the key.

For example, the .csv file might look like:

```
Weapon,Damage,Weight,Cost
Long Sword,1d8,4,30
Battle Axe,1d10,7,20
Crossbow,1d8,5,50
Dagger,1d4,1,5
```

Now, in the designer open up the Document Setup window (Sheet - Document Setup from the menu).

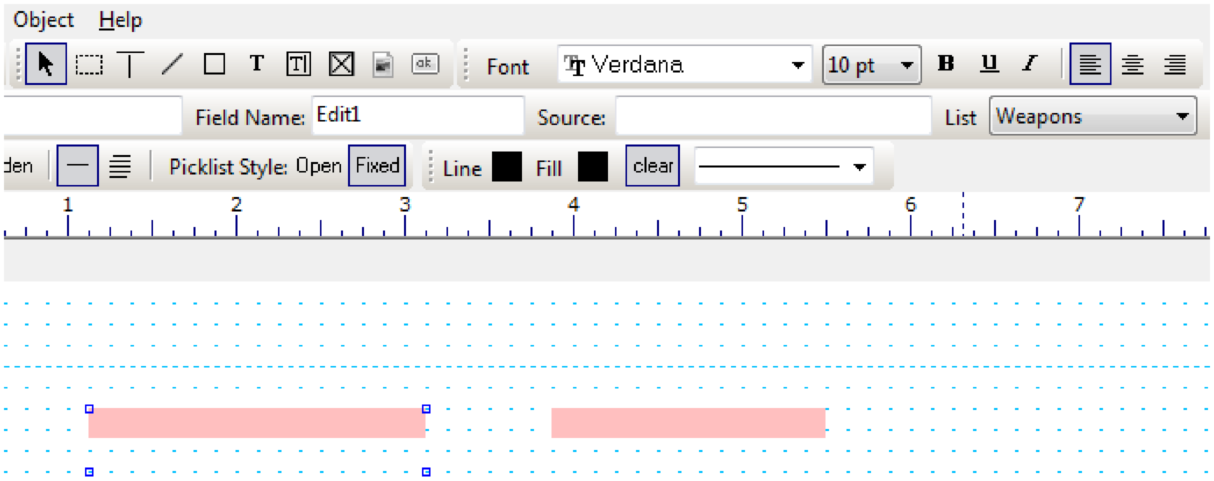


C:\DataStores\Weapons.csv

Use the Add button to select your .csv file. It will be added to the list of DataStore files. Then press Ok. Now the .csv file is added as a DataStore into your Character Sheet definition, and a list named 'Weapons' is available for use with drop down edit boxes.

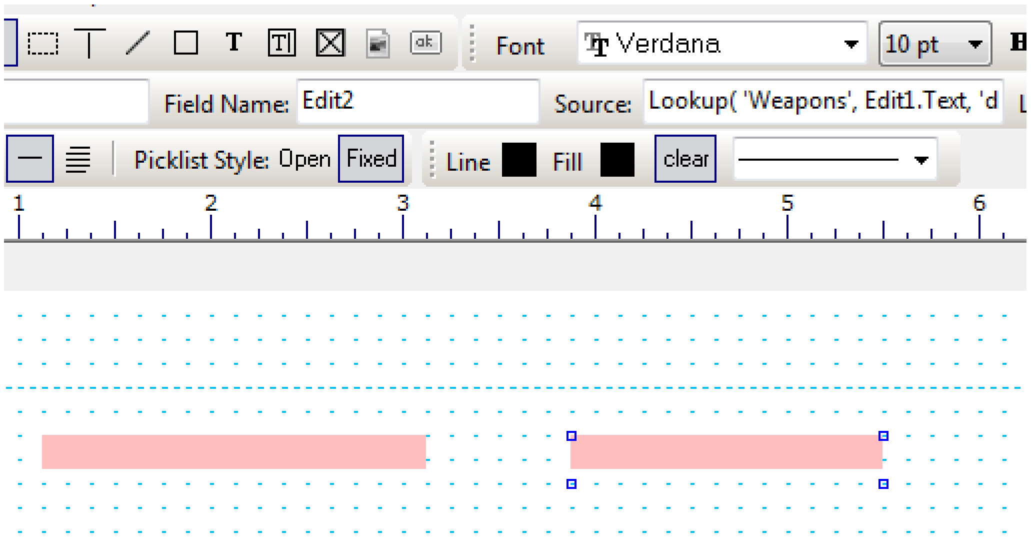
Now place two edit boxes on your Character Sheet definition.

Assign the list named 'Weapons' to the first edit box, Edit1.



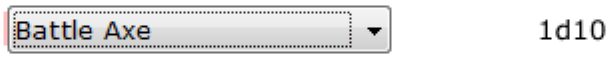
Then assign the following DataStore lookup call as the Field Source for the second edit box, Edit2:

```
Lookup( 'Weapons', Edit1.Text, 'damage')
```



Field Name: Edit2
Source: Lookup('Weapons', Edit1.Text, 'd

What this does is, in the viewer, tell the second edit box, Edit2, to populate itself with the 'damage' field of the Weapon's DataStore for the weapon who's name is in Edit1.



Battle Axe

1d10

The Lookup() function always performs its lookups against the key field of the DataStore.

5.13 JavaScript API

5.13.1 Field Properties

Fields on your character sheet may be referenced within JavaScript functions by name. A field named Edit1, for example, can be referenced as a javascript object named Edit1 in a function.

Field objects have several properties that can be set or read. They include:

Text

This is the text contents of a field. You can both read and set this field. For example:

```
NameEditField.Text = 'Character Name';
```

Value

This is a numeric representation of the text in the field. If the contents of the field can be represented as a number, the number is returned. If not, 0 is returned. For example:

```
TotalHits.Text = Strength.Value + Constitution.Value;
```

This field is *read-only*, and cannot be set.

Checked

True if the checkbox field is checked, false if it is not.

FieldName

This is the name of the field. ex, 'Edit1', 'Button1', etc. This is thus also the name of the corresponding javascript object.

FieldSource

This is the field source/calculation assigned to the field. This should not be changed via script.

isNumeric

Indicates whether or not the field should be displayed as a number only. Value is true or false.

ListStyle

This determines whether or not the field, if a list is assigned, is Fixed or Open. The valid values are 0 for Fixed, 1 for Open.

List

This is the name of the list, if any, assigned to an edit field. If a list is assigned, the edit field is a pick list.

ReadOnly

This determines whether or not an edit field can be edited by the end user. Value is true or false.

Visible

True if the field is visible, false if it is hidden.

LineWidth

This is the width, in Points, of a line or box's line.

LineColor, FillColor

These are the line and fill colors of lines and boxes. Color is a 32bit integer as returned by the rgb() function.

isFilled

True if a box is filled, false if it is only an outline.

FontName

This is the name of the

FontSize

The size, in Points, of the text.

FontBold, FontUnderline, FontItalic

Indicates the style of font. Values may be true or false.

TextAlign

The alignment of a text edit field. The valid values are:

```
0 - Left Align
1 - Center Align
2 - Right Align
```

SingleLine

True if the text edit field is limited to one line, false if it supports multiple lines of text.

ImageGrayscale

True if the image should be displayed in grayscale, false if in full color.

5.13.2 JavaScript Functions

The Character Sheet Viewer's JavaScript interface has a number of built-in functions that you might find useful when putting together your character sheets.

5.13.2.1 Lookup (DataStore)

Syntax

Lookup(*dsName*, *dsKey*, *dsField*)

Description

Returns the value for field *dsField*, in the DataStore named *dsName*, that corresponds to the key field that matches *dsKey*.

Example:

```
Edit2.Text = Lookup( 'Weapons', Edit1.Text, 'damage');
```

Which would look up, in a DataStore named 'Weapons', the 'damage' value for the weapon who's name is stored in the Edit1.Text field.

5.13.2.2 CombineDataStore (DataStore)

Syntax

CombineDataStore(*dsName1, dsName2*)

Description

This function adds the contents of the DataStore named *dsName2* to the DataStore named *dsName1*. If *dsName1* does not exist already, an empty DataStore named *dsName1* is created.

Example:

```
CombineDataStore( 'AllRaces', 'RacesBook1');
CombineDataStore( 'AllRaces', 'RacesBook2');
```

These would add the contents of the DataStore named *RacesBook1* and *RacesBook2* to a DataStore named *AllRaces*.

5.13.2.3 RollDice()

Syntax

RollDice(NumDice, DieType, Modifier)

Description

Roles a specified number of a type of die, and adds Modifier to the result.

Example:

```
RollDice( 3, 6, 4)
```

Would roll 3d6+4

5.13.2.4 RollDiceOpen()

Syntax

RollDiceOpen(NumDice, DieType, Modifier)

Description

Roles a specified number open ended dice, adding *Modifier* to the result. Individual dice rolls are treated as open ended if the result of the individual die is the max possible for the die (6 on a d6, for example), or 95-100 when *DieType* is 100. When a die is open ended, an additional die is rolled and added to the single die's total. If that die is also an open ended result, the process continues.

Example:

```
RollDice( 2, 20, 4)
```

Would roll 2d20+4, with each d20 being treated as an open ended die if a 20 was rolled.

5.13.2.5 EvalDice()

Syntax

EvalDice(DiceExpression)

Description

Evaluates a dice expression contained as a string in the form of nDn+n.   The modifier can be +, -, *, or /.

Example:

```
EvalDice( "3d6+1");
EvalDice( "10d4-10);
EvalDice( "1d10*10);
EvalDice( "1d100/10);
```

5.13.2.6 d2()

Syntax

d2()

Description

Rolls 1d2.

Example:

```
var n = d2() + d4() + d6() + d8() + d10() + d12() + d20() + d30() + d100()
```

Rolls 'wandering damage'... one of each type of die.

5.13.2.7 d4()

Syntax

d4()

Description

Rolls 1d4.

Example:

```
var n = d2() + d4() + d6() + d8() + d10() + d12() + d20() + d30() + d100()
```

Rolls 'wandering damage'... one of each type of die.

5.13.2.8 d6()

Syntax

```
d6()
```

Description

Rolls 1d6.

Example:

```
var n = d2() + d4() + d6() + d8() + d10() + d12() + d20() + d30() + d100()
```

Rolls 'wandering damage'... one of each type of die.

5.13.2.9 d8()

Syntax

```
d8()
```

Description

Rolls 1d8.

Example:

```
var n = d2() + d4() + d6() + d8() + d10() + d12() + d20() + d30() + d100()
```

Rolls 'wandering damage'... one of each type of die.

5.13.2.10 d10()

Syntax

```
d10()
```

Description

Rolls 1d10.

Example:

```
var n = d2() + d4() + d6() + d8() + d10() + d12() + d20() + d30() + d100()
```

Rolls 'wandering damage'... one of each type of die.

5.13.2.11 d12()

Syntax

```
d12()
```

Description

Rolls 1d12.

Example:

```
var n = d2() + d4() + d6() + d8() + d10() + d12() + d20() + d30() + d100()
```

Rolls 'wandering damage'... one of each type of die.

5.13.2.12 d20()

Syntax

```
d20()
```

Description

Rolls 1d20.

Example:

```
var n = d2() + d4() + d6() + d8() + d10() + d12() + d20() + d30() + d100()
```

Rolls 'wandering damage'... one of each type of die.

5.13.2.13 d30()

Syntax

```
d30()
```

Description

Rolls 1d30.

Example:

```
var n = d2() + d4() + d6() + d8() + d10() + d12() + d20() + d30() + d100()
```

Rolls 'wandering damage'... one of each type of die.

5.13.2.14 d100()

Syntax

```
d100()
```

Description

Rolls 1d100.

Example:

```
var n = d2() + d4() + d6() + d8() + d10() + d12() + d20() + d30() + d100()
```

Rolls 'wandering damage'... one of each type of die.

5.13.2.15 DebugMsg()

Syntax

```
DebugMsg( sMessage)
```

Description

Outputs the text *sMessage* to the Character Sheet Viewer's debugging window.

It is important that you remove the DebugMsg() calls prior to distributing your character sheet, or the messages will display when other people try to use the sheet.

This function requires version 2.01b or later of the viewer.

Example:

```
DebugMsg( 'The value is ' + Edit1.Text);
```

Outputs the text of the field named Edit1 to the debugging window.

6 Viewing Character Sheets

6.1 Viewer

In the viewer you can highlight/un-highlight editable fields using the entry on the Sheet menu.