⇑

AcroTEX.Net

AcroTEX PDF Blog

Processing Acrobat Forms using JavaScript

External Processing of a Field

Part 4: Button Fields, the Check Box

D. P. Story

AcroTEX PDF Blog #15, Revision #2

http://www.acrotex.net
Published: December 9, 2008

Table of Contents

6. Button Fields: The Check Box

A button field is an interactive control the user manipulates with the mouse. Button fields do not take keyboard input from the user. There are three types of buttons fields,

1. A *Push button* is a purely interactive control that has not value.
2. A *check box* toggles between two states, on and off.
3. *Radio button fields* contain a set of related buttons that can each be on or off. Normally, when one radio button is on, the others in the same field are off.

In this article, we shall concentrate all our efforts on the *check box*.

6.1. Getting and Setting the Value of a Check Box

A check box can be either 'on' and 'off'. Within the PDF file itself, 'off' is actually the key word `Off` and 'on' is the key word `Yes`. The values of `Off` and `Yes` are descriptions of the 'off' and 'on' appearances of the check box.

The string used to reference the 'on' value can be changed through the user interface and through JavaScript, to any text string; the string to reference the 'off' state (`Off`) cannot be changed, however. `Yes` is the default 'on' state for English based PDF viewers (Acrobat/AR). When a check box is created with the user interface in a language localized version of Acrobat (French, German, etc.) the default 'on' state is not `Yes`, but a localized string such as `Oui`, `Ja`, etc.

In the Options tab, the user interface references the *export value* of a check box; this is where the 'on' state can be redefined from `Yes`, to whatever text string you wish.

As was just mentioned, a check box has two states, 'on' (`Yes`, which may be redefined) and 'off' (`Off`, which cannot be redefined). We shall discuss how to redefine the 'on' state in the section on the 'The Options Tab' on page 8. In this section we shall explore getting and setting the current state of the check box, and how to programmatically check or clear the box.

Unlike the push button, the check box has a value, accessed with `Field.value`. The value of the check box is retained in the PDF file as the value of the `V` key. `Field.value` can be used to query the state of the check box, and to get and set the state as well.

Normally, you should set the `Field.value` to either `"Off"` or `"Yes"` (the export value, or 'on' value), but other values are not prohibited. Other values do not correspond, however, to any appearance state of the check box, and would be rendered with the 'off' appearance.

Before continuing with the example, let me elaborate on the difference between *`Field`*`.value` versus the export value. When *`Field`*`.value` is used to assign a value, this value is compared to the export value, if they match, the check box is given the ‘on’ appearance; if they do not match, the ‘off’ appearance is rendered (the check box is cleared).

Example 6.1. Get the value of a check box. Check, or clear, the check box, then press the push button. ☛☐ Get It!

```
1  var f = this.getField("ck6-1");
2  msgStr= "The check box is " + ((f.value=="Off") ? "off" : "on")
3          + ", the export value is \"" + f.value + "\".";
4  app.alert(msgStr);
```

Comments. We get the Field object (1), construct a message based on the value of `f.value` (2), and display an alert box with the message (3).

`Field``.value` is used primarily to get the export value of the check box, the ‘on’ value, but it can also be used to check the box, as the next example demonstrates.

Example 6.2. Toggle the readonly check box using the push button provided.

☐ Toggle It!

```
1  var f = this.getField("ck6-2");
2  f.value = ( f.value=="Off" ) ? "Yes" : "Off";
```

Comments. We get the Field object (1); set the state to the opposite of what it currently is (2), using the value of `f.value`.

Trying to assign a value to a check box that is neither `"Off"` nor `"Yes"` really does nothing, and may have some slight side effects. In the above example, does nothing. Open the JavaScript Debugger Console window (Ctrl+J) and enter the following code:

```
var f = this.getField("ck6-2");
f.value = "dps";
```

If the check box is ‘on’, the box will change to the ‘off’ state; if the box is in the ‘off’, it remains that way. After executing `f.value="dps"`, press the **Toggle It!** button, there is some slight disruption of the toggling. The behavior I seems to obey this logic:

- Suppose the check box is clear and has a value of `"Off"`. Now execute `f.value = "dps"`. Click on the **Toggle It!** button, `f.value` is not equal to `"Off"`, so line (2) assigns `f.value` a value of `"Off"`, the check box remains clear. Click **Toggle It!** as it toggles on as it should.
- Suppose the check box is clear and has a value of `"Yes"`. Now execute `f.value = "dps"`. The new value `"dps"` is not equal the the export value `"Yes"`, so the check box is cleared (is rendered as the ‘off’ appearance). Now click **Toggle It!**, since `f.value="dps"`, `f.value` is not "Off", so line (2) assigns value of `"Off"`, and the check box remains off after that click. Click again, and everything is copacetic.

Important: When we set `f.value = "dps"` in the above example, this pretty well messes up the techniques illustrated in Example 6.1, page 4. In that example, we built a message string like so,

```
msgStr= "The check box is " + ((f.value=="Off") ? "off" : "on")
        + ", the export value is \"" + f.value + "\".";
```

Now, if `f.value="dps"`—dps is always causing trouble—the string would say that the check box has an 'on' appearance, which it may not have.

The weakness of the approach in Examples 6.1 and 6.2 is that you need to know the name of the *export value* of the check box, and you need to know that the value of the field has not been changed to some bogus value, like dps did. Conceivably, there are situations where the 'on' value is not known, unless, of course, the check box is 'on'. `:-{)` There are two methods, *Field*.`isBoxChecked` and *Field*.`checkThisBox`, that are used to determine the state of a check box and for checking (or clearing) a check box. The export value is not needed with these two methods.

The next example uses *Field*.`isBoxChecked` to query the state of a check box. According to *JavaScript for Acrobat API Reference*, *Field*.`isBoxChecked` takes one parameter, `nWidget`, the 0-based index of the check box widget for the field. In the example that follows, we take this parameter to have a value of 0. The return value of *Field*.`isBoxChecked` is `true`, if the specified check box is currently checked, `false`, otherwise.[1]

Example 6.3. Use *Field*.`checkThisBox` to query the state of a check box. Check, or clear, the check box, then press the push button. ☛☐ Get It!

```
var f = this.getField("ck6-3");
var state=f.isBoxChecked(0);
msgStr= "The check box is " + ( (state) ? "checked." : "cleared.");
app.alert(msgStr);
```

Comments. After getting the Field object of the target check box, line (1), we save the value of `f.isBoxChecked(0)` in line (2) in the variable `state` (this step is really not needed), and use the value of `state` to build a message (3). Finally, we broadcast our message as an alert box. ☐

We can modify Example 6.2 to use the *Field*.`checkThisBox` method to check or clear a check box.

Field.`checkThisBox` has two parameters, and returns nothing. The first parameter is `nWidget`, which has the same description as in *Field*.`isBoxChecked`, and the second parameter is `bCheckIt`, a Boolean value which if true, the check box is checked, and if false,

[1]This method actually return 1 for `true` and 0 for `false`.

the check box is cleared. The default if `bCheckIt` is true. See a later section, Widgets–Multiple Fields with the same Name on page 14, for a more detailed discussion of the notion of a widget; you'll find another example of the use of *`Field`*`.checkThisBox` and the role the `nWidget` parameter plays in Example 7.12, page 18.

Example 6.4. Use *`Field`*`.checkThisBox` to toggle the readonly check box using the push button provided. push button.

☐ Toggle It!

```
1 var f = this.getField("ck6-4");
2 f.checkThisBox(0,!f.isBoxChecked(0));
```

Comments. We get the Field object (1); set the state to the opposite of what it currently is, in line (2). The first argument of `f.checkThisBox` set to 0, and `f.isBoxChecked(0)` is used to set the value of the second parameter to the opposite (`!f.isBoxChecked(0)`) of what it currently is.

Beginning with Acrobat 5.0, the parameters of most methods in the JavaScript for Acrobat API may be entered as a literal object. For example, line (2) could also be written as

```
f.checkThisBox({ nWidget: 0, bCheckIt: !f.isBoxChecked(0) });
```

I say most of the API because some of the multimedia methods (now called legacy multimedia) that was created back in Acrobat 6.0, *does not use* this notation. For more details on this object literal notation, see the section named Syntax of the *Acrobat for JavaScript API Reference* □

I think we've beat the drum long enough, yet's move on.

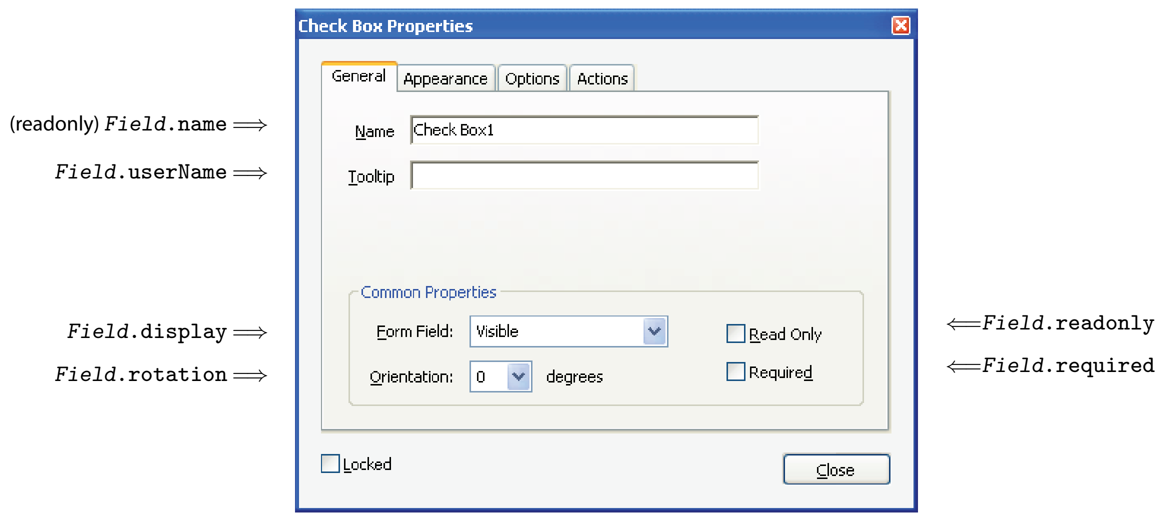
6.2. Changing the Appearance and the Properties of the Button Field

Until such time arrives that we begin to examine the Actions tab—where JavaScript actions are defined—we shall concentrate on the appearance and other properties of the check box.

• The General Tab

Below, see Figure 1 on page 7, is a screen shot of the General tab of the Check Box Properties, parallel to the dialog box, are the JavaScript properties that correspond to the elements on the General tab of the Check Box Properties dialog box. The properties of the General tab, are the same as those of the other fields.

We have pretty well illustrated these properties in the previous blogs, AcroTEX Blog #13 and AcroTEX Blog #14, so no examples will be presented here.



Name: Check Box1
Tooltip:
Form Field: Visible
Orientation: 0 degrees

Figure 1: The General Tab of Check Box Properties

• The Appearance Tab

Let's do the same now for the appearance tab, see Figure 2 on page 8

As with the General tab, use these properties, listed in Figure 2, by first acquiring the Field object of the target field, and assigning values to the JavaScript properties. In general, some properties and methods are only available in Acrobat, and not available on Adobe Reader. See the *JavaScript for Acrobat API Reference* for full documentation, pay attention to the quick keys that accompany the properties and methods in the *JavaScript for Acrobat API Reference*.

Border Color: Determines the color of the border; `Field.strokeColor` is its counter-part in JavaScript.

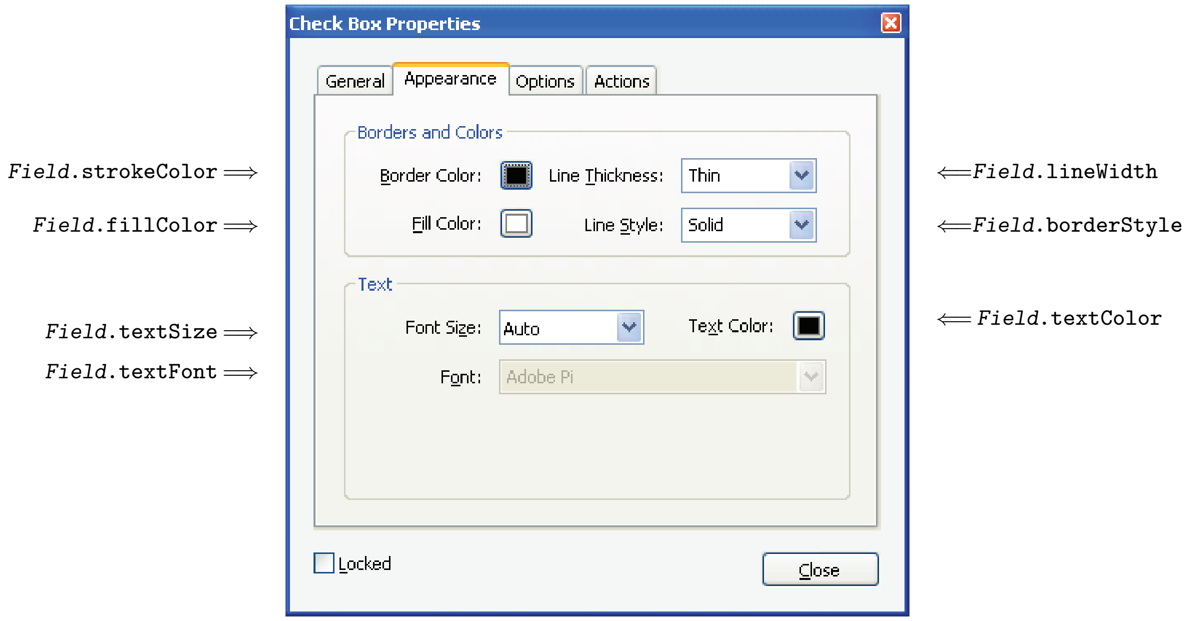
Fill Color: The background color of the check box.

Font Size: The size of the check. Setting the `Field.textSize=0` is equivalent to selecting Auto from the drop-down list in the user interface.

Font: The default value of the Font property in Figure 2 is Adobe PI; consequently, the JavaScript property `Field.textFont` is readonly.

Thickness: Use `Field.lineWidth` to set the width of the border surrounding the bounding rectangle of the check box. Possible values are 0 (no boundary), 1 (Thin), 2 (Medium), 3 (Thick).

Line Styles: How the border is rendered, the user interface offers the choices: Solid, Dashed, Beveled, Inset, and Underlined. The corresponding values for `Field.borderStyle` are `border.s`, `border.d`, `border.b`, `border.i`, and `border.u`.



Check Box Properties

Borders and Colors

Border Color: ■ Line Thickness: Thin

Fill Color: □ Line Style: Solid

Text

Font Size: Auto Text Color: ■

Font: Adobe Pi

☐ Locked

Figure 2: The Appearance Tab of Check Box Properties

Executing `Field.borderStyle=border.s` sets the Line Style to Solid.

Text Color: This property determines the color of the "check mark". To set the "check mark" to red, execute `Field.textColor=color.red`

We have pretty well illustrated these properties in the previous blogs, AcroTEX Blog #13 and AcroTEX Blog #14, so no examples will be presented here.

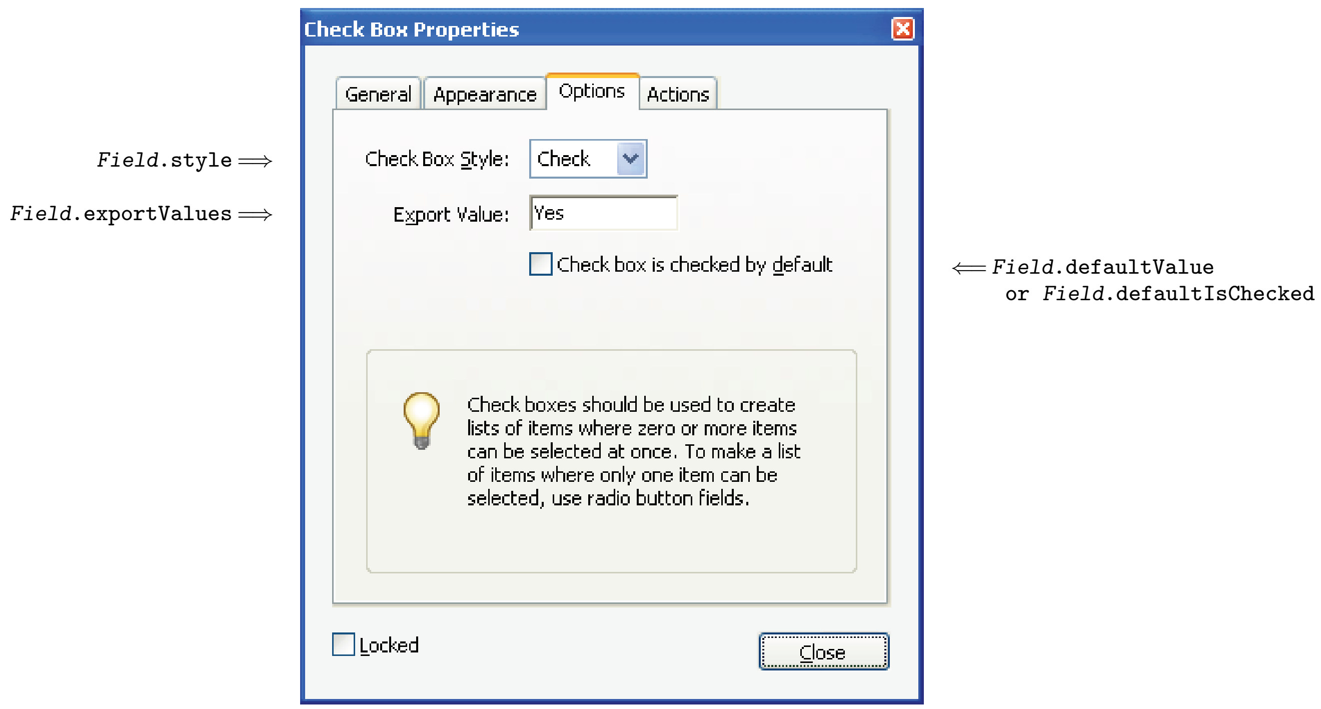
• The Options Tab

In Figure 3 on page 9, we set up the correspondence between each user interface element and its JavaScript counterpart.

Check Box Style: What style of check to use, the user interface offers the following choices: Check, Circle, Cross, Diamond, Square, and Star. The corresponding values for `Field.style` are `style.ch`, `style.ci`, `style.cr`, `style.di`, `style.sq`, and `style.st`; for example, `Field.style=border.ci` sets the Check Box Style to Circle. See Example 6.5, page 9.

Export Value: This field is the user interface to setting the export value. To set this value programmatically, use `Field.exportValues`. See Example 6.6, page 10.

Check box is checked by default: If this box is checked, the check box will be checked (be given the 'on' appearance) when the for field is reset. Use `Field.defaultValue` or



Check Box Properties

General | Appearance | Options | Actions

Check Box Style: Check

Export Value: Yes

☐ Check box is checked by default

Check boxes should be used to create lists of items where zero or more items can be selected at once. To make a list of items where only one item can be selected, use radio button fields.

☐ Locked

Close

Figure 3: The Options Tab of Check Box Properties

Field.defaultIsChecked to check or clear this box, and use *Field*.defaultIsChecked to test whether this box is checked. See Example 6.7, page 10.

The next example demonstrates the use of *Field*.style.

Example 6.5. Set the check box style using *Field*.style.

Check | Select It!   My Check Box: ✓

The push button action is given below.

```
1  var c = this.getField("myCombo6-5");
2  var f = this.getField("myCB6-5");
3  f.style=eval(c.value);
```

Comments. We get the Field objects in line (1) & (2) of the combo box and the check box; in line (3), we set the f.style property of the check box equal to eval(c.value). c.value is the export value of the combo box, and these have been set to the various values of the style property. ☐

Now we demonstrate the use of *Field*.exportValues. The *Field*.exportValues property is used for combo boxes and radio button fields. It's value is an array, we quote from

JavaScript for Acrobat API Reference,

> An array of strings representing the export values for the field. The array has as many elements as there are annotations in the field. The elements are mapped to the annotations in the order of creation (unaffected by tab-order).

When we take up the topic of widgets in a later section, Widgets–Multiple Fields with the same Name on page 14, we will not concern ourselves with what this means. Suffice it to say, the value is an array of export values. Our array, the one in Example 6.6 will only have length 1.

Example 6.6. Enter an export value (a string) in the text field, then press the push button. We will redefine the export value of the check box.

☐ Change Export Value! My Check Box ☐

The example will work for Acrobat, and for Adobe Reader with additional form usage rights.

The push button action is given below.

```
1 var t = this.getField("myText6-6").value;
2 var cb = this.getField("myCB6-6");
3 var stript = t.replace(/\s*/g,"");
4 if ( stript == "" ) t = "Yes"
5 cb.exportValues=[t];
6 cb.checkThisBox(0,true);
7 var ev = cb.value;
8 app.alert("The export value has been set to \""+ev+"\".");
```

Comments. In line (1) we get the value the user entered in the text field. In line (2) we get the Field object of the check box. Lines (4)-(5), we strip out all white spaces, if the text field has nothing but white spaces, `stript` is the empty string and we use `"Yes"` as the export value, line (4). In line (5), we set the export value, by putting the text string, `t`, into an array. To prove to you, my infrequent readers, I check the check box (6), and get the value of the field, line (7). The variable `ev` should be equal to the new export value if things went well for us. Line (8), it did! □

Example 6.7. Use *Field*.`defaultIsChecked` to determine if a check box is checked by default. If a check box is checked by default, the field will be checked if the form is reset. Uses *Field*.`defaultIsChecked` to randomly check the default box of these two check boxes.

☐ ☐ Report on Default Setting! Reset

The push button action is given below.

```
1  var cbL = this.getField("myCBL6-7");
2  var cbR = this.getField("myCBR6-7");
3  var bSetDefault = ( Math.random() <= .5 ) ? true : false;
4  cbL.defaultIsChecked(0, bSetDefault);
5  cbR.defaultIsChecked(0, !bSetDefault);
6  var msgStr = "The left check box is "
7      + ((cbL.isDefaultChecked(0)) ? "checked" : "not checked")
8      + " by default.\r\r"
9      + "The right check box is "
10     + ((cbR.isDefaultChecked(0)) ? "checked" : "not checked")
11     + " by default.\r\r"
12     + "Now click the Reset button."
13 app.alert(msgStr);
```

Comments: In line (3) we randomly select which text field to give a default value. In lines (4) & (5), we set the default values of the two fields, when we prepare and emit the message (6)–(12). □

7. Naming Conventions and Finer Points for using the Field Object

This section should have been presented in AcroTEX Blog #12, and may yet be moved there; here, we discuss the common naming conventions, and some of the finer points of using the Field object.

7.1. Naming Conventions

Much of the material in this section is taken from the introduction to the Field Object of the *JavaScript for Acrobat API Reference*.

The most natural way of naming fields is to use the *flat naming* system; for example, we may create a text fields with names of "FirstName", "MiddleName", and "LastName".[2] Though the names of the fields suggest an common purpose of these three fields—to obtain name information for an individual—there is no association between fields. If it were desired to change the border color of the text fields, for example, one would have to change the border color of each field, like so,

[2]Here we given the names of the fields as a JavaScript string. If these names are entered through the user interface, they would be entered as FirstName, MiddleName, and LastName.

```
var f=this.getField("FirstName");
f.strokeColor=color.red;
f=this.getField("MiddleName");
f.strokeColor=color.red;
f=this.getField("LastName");
f.strokeColor=color.red;
```

Slow and tedious. Yet, these fields have a common purpose, one should be able to build an association or structure between such fields. That is what the other naming convention does, the *hierarchical naming* system.

To use the *hierarchical naming* on the three fields above, we might name them `"Name.First"`, `"Name.Middle"`, and `"Name.Last"`; this forms a tree of fields. The period (.) separator denotes a *hierarchy shift*.

The root word `"Name"` is the parent of these three fields, while `"First"`, `"Middle"`, and `"Last"` are its children. The field `"Name"` is called the internal name because it has no visible appearance; `"First"`, `"Middle"`, and `"Last"` are called *terminal fields*, they have a visual appearance on the page.

A field using a flat naming convention is always a terminal field, it has not parent, nor does it have kids.

Important: When fields have a hierarchical structure, properties can be changed for the whole field structure. To change a property or attribute for all the children in a historically structured field, acquire the Field object of the parent field. With this object, change the properties of all the children.

Example 7.8. Toggle the boundary colors of these three fields using the Field object of the parent field.

First Name:	
Middle Name:	
Last Name:	

Now, let's change the color of the boundary. Toggle Boundary!

The push button action is given below.

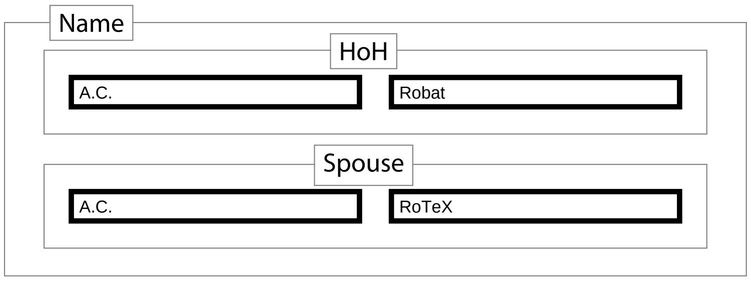
```
1 if ( typeof myColor == "undefined" ) myColor={ color: color.red };
2 var f = this.getField("Name");
3 f.strokeColor = myColor.color;
4 myColor.color = ( color.equal( myColor.color, color.red ) )
5     ? color.blue : color.red;
```

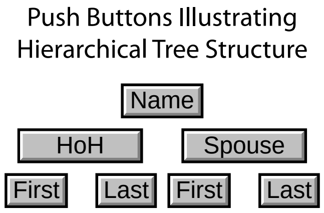
Comments: In line (1), we prepare a custom color object, we use this color to change the boundary color in line (3). Then, we change the color between red and blue in line (4)–(5). **Important:** Notice that we get the field object of the parent field in line (2), the property change is inherited by its kids.

The `color.equal` method is used in this example, □

Here is an example of a field structure with two hierarchy shifts.

Example 7.9.





The push buttons above-right illustrate the hierarchical tree structure of the text fields to the left. `Name` is the parent, and has four kids. `Name.HoH` is the parent of two kids, `First` and `Last`, and is the child of `Name`. `Name.Spouse` is the parent of two kids, `First` and `Last`, and is the child of Name. The tooltip of each button supply additional information. □

Using *Field*.getArray(). *`Field`*`.getArray` is useful method for working with a hierarchical structure. The method has no parameter, and returns an array of Field objects. If `f` is a (parent field with kids), and `g=f.getArray()`, then `g` is the array of field objects of the kids. You can count the number of kids the parent field has with *`Field`*`.getArray().length`.

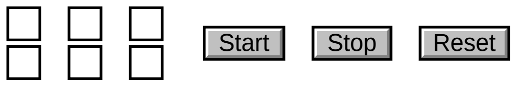
Here is an example of how to use *`Field`*`.getArray()`.

When you create a field, such as a check box, using the user interface and right-click one it and select Place Multiple Fields from the context menu, you can create an table of fields (rows and/or columns). The Acrobat interface uses hierarchical naming, if `cb` is the base name, the first rows are named:

```
cb.0.0    cb.0.1    cb.0.2    ...
cb.1.0    cb.1.1    cb.1.2    ...
...       ...       ...       ...
```

The *`Field`*`.getArray()` method is ideal for handing generically named fields such as these.

Example 7.10. Click the push button, and watch the action that occurs. Enjoy!



The code for the Start button follows:

```
var f = this.getField("cb7-10");     // get parent object
var aLiteIt = f.getArray();          // get its array of children
var shine = app.setInterval("liteItUp()",250);
var shineOff = app.setTimeOut("app.clearInterval(shine)", 10000);
```

We use the methods app.setInterval, app.setTimeOut, and app.clearInterval, see linked references for details.

The above script calls a function defined at the document level. That code is given below.

```
1  var aLiteIt = new Array();
2  var nLiteIt = -1;
3  function liteItUp() {
4      try { aLiteIt[nLiteIt].checkThisBox(0,false); } catch(e) {}
5      nLiteIt = ((++nLiteIt) % aLiteIt.length);
6      aLiteIt[nLiteIt].checkThisBox(0,true);
7  }
```

Comments: We create an array to keep the Field object of the kids of field "cb7-10", in line (1). nLiteIt is an index for the array aLiteIt, line (2). The function definition of liteItUp() is defined in lines (3)–(7). In line (4), we turn off the current check box. We enclose line (4) in a try/catch construct; the first time through nLiteIt has a value of -1, and an exception will be thrown. We increment nLiteIt mod aLiteIt.length and turn on the next check box.

If you are interested, look at the code for the Stop button as well. This is enclosed in try/catch in case you press the button when the animation is not running; the variables referenced there are undefined, in that case. □

7.2. Widgets–Multiple Fields with the same Name

In the user interface, it is not unusual to make copies of a field.[3] The result is that you have two or more fields with the same name. There are two interesting things about fields with the same name:

1. Fields that share a common name also share a *common value* (*Field*.value) same value. If the value of one field is changed, the other fields with the same name are updated with the new value.
2. Fields that share a common name can also have different presentations or appearances of their (common) value. Fields can be on different pages, have different border color,

[3]Form fields can also be created by the JavaScript *Doc*.addField, or through an authoring program like LaTeX using the **pdfmark** operator.

background color, text font, text size, and so on.

Placing fields with the same name on different pages, allows you to transfer date the user entered earlier to another page. Here is a short example of these two points. The following two text fields have the same name but different appearances. Enter some text in one field, and see it transferred to the other field:

If you open the Fields Navigation panel, you'll see the duplicate fields listed by the terminal field name, with a suffix of `#<index>`. The duplicated fields are indexed by a 0-based numbering system.

• What's a Widget.

Each presentation of a terminal field is called a *widget*.[4] A *widget does not have a name*, but is identified by its index within its terminal field. A phrase that is repeated numerous times throughout *JavaScript for Acrobat API Reference* is[5]

> The index is determined by the order in which the individual widgets were created (and is unaffected by the tab-order).

Below, we illustrate this statement, using check boxes, remember this is a blog on check boxes.

• Getting the Field object of a Widget.

When you create multiple fields with the same name, the visual appearances of these fields are called widgets, a term used by the super swave developers of the Acrobat engineering team. Beginning with Acrobat 6.0, we can acquire the Field object of an individual widget, and hence, can change its properties, changing its appearance.

Suppose `"myFieldName"` is the name of a field that has been duplicated (this is a terminal field name), and, thus, has several associated widgets. To acquire the Field object of the first widget (the one first created, by the above paragraph), we execute

```
var f = this.getField("myFieldName.0");
```

The returned Field object, encapsulates the properties of that first widget only. The period (.) here is *not* a *hierarchy shift*, but just a notation that delimits the end of the terminal field name and the beginning of the index value.

[4]Every form field that has a visual representation is a Widget Annotation, but this is not what is meant here—at least I don't think.

[5]We quote from the Field Object section of that mighty reference.

Once the Field object is acquired for a particular widget, we can change a property, for example, my favorite boundary color, like so,

```
var f = this.getField("myFieldName.0");
f.strokeColor=color.red;
```

Similarly, `this.getField("myFieldName.1")` gets the Field object of the second widget created, and so on.

Example 7.11. The check boxes that appear below all have the same name (`cb7-11`) and are, therefore, widgets. They have been created *column-wise*. Enter the widget index in the text field (an integer between 0 and 7) that corresponds to the second row and the third column. If you answer correctly, the border color of this widget will change color.

Reset ☐ ☐ ☐ ☐
☐ ☐ ☐ ☐

Click one of these check boxes? What happens? They all are checked aren't they. This is because all these check boxes have the same export value (`Yes`). When one is clicked, the value of that check box is `Yes`, all the other widgets share the same value of `Yes`, so yes, they all are turned on.

By the way, if you ever want to programmatically count the number of widgets in the field, here is how to do it.

Count Widgets!

```
var f=this.getField("ck7-11");
console.show(); console.clear();
if ( typeof f.page == "number" )
    console.println("No duplicate fields detected!");
else {
    console.println("Duplicate fields detected!!");
    console.println("The field \""+f.name+"\" has "
        +f.page.length+" widgets.");
}
```

Comments: In this example, we report results to the console window. *`Field`*`.page` returns an integer value, the page number, if the field only appears once in the document, and returns an array of page numbers. If the property returns an array, the field must be a terminal field with two or more widgets. We use this fact to make our determinations in this example. □

If the export values of the check box widgets are different, something unexpected happens.

☐ ☐ ☐ ☐
☐ ☐ ☐ ☐

Some of these widgets have an export value of `Yes`, others have an export value of `Dps`. Can you explain the behavior of these check boxes?

7.3. Getting and Setting Widget Attributes

In the sections titled Field Object and Field versus widget attributes in the now infamous *JavaScript for Acrobat API Reference*, there is a table and a two lists of properties and methods. The sections suggest there are two types of properties and methods: field-level and widget level. The two lists enumerate the properties and methods that fall into each category.

The table, Figure 1, reproduced on page 17, describes how the getting and setting of widget attributes are affected as a function of field-level and widget-level properties and methods.

Action	Field object that represents all widgets	Field object that represents one specific widget
Get a widget property	Gets the property of widget #0	Gets the property of the widget
Set a widget property	Sets the property of all widgets that are children of that field. (The `rect` property and `setFocus` method are exceptions that apply to widget #0.	Sets the property of that widget.
Get a field property	Gets the property of the field	Gets the property of the parent field.
Set a field property	Sets the property of the field	Sets the property of the parent field.

Table 1: Getting & Setting Widgets

The section Field versus widget attributes of *JavaScript for Acrobat API Reference*, lists the field-level and widget-level properties and methods. This listing follows.

• Field-level properties and methods

Field properties are like properties common to all the widgets, if you set *`Field`*`.readonly` to `true`, for example, all widgets with the same terminal name are set to readonly, regardless of whether the *`Field`* object is that of the parent field (where we used `getField` with the name of the terminal field as its argument), or is that of one of the individual widgets. Set one widget to readonly, you set them all to readonly.

These properties and methods are…

`calcOrderIndex`, `charLimit`, `comb`, `currentValueIndices`, `defaultValue`, `doNotScroll`, `doNotSpellCheck`, `delay`, `doc`, `editable`, `exportValues`, `fileSelect`, `multiline`, `multipleSelection`, `name`, `numItems`, `page`, `password`, `readonly`, `required`, `submitName`, `type`, `userName`, `value`, `valueAsString`, `clearItems`, `browseForFileToSubmit`, `deleteItemAt`, `getItemAt`, `insertItemAt`, `setAction`, `setItems`, `signatureInfo`, `signatureSign`, and `signatureValidate`.

• Widget-level properties and methods

Widget-level properties and methods are properties that can be set in an individual widget.

These properties and methods are…

`alignment`, `borderStyle`, `buttonAlignX`, `buttonAlignY`, `buttonPosition`, `buttonScaleHow`, `buttonScaleWhen`, `display`, `fillColor`, `hidden`, `highlight`, `lineWidth`, `print`, `rect`, `strokeColor`, `style`, `textColor`, `textFont`, `textSize`, `buttonGetCaption`, `buttonGetIcon`, `buttonImportIcon`, `buttonSetCaption`, `buttonSetIcon`, `checkThisBox`, `defaultIsChecked`, `isBoxChecked`, `isDefaultChecked`, `setAction`, and `setFocus`

To summarize, if we set a widget property using the Field object of the terminal field (representing all widgets), we set all widgets; if set a widget property with a Field object of an individual widget, we set that individual widget, that seems natural. For getting, if we are using the Field object of the terminal field (representing all widgets) we get the property of widget #0, this is a choice made by the Acrobat developers, the property value of this property may vary from widget to widget, so we just return the value for the first widget. If we are using a Field object for an individual widget, we get that property value for that particular widget.

One other comment that is a appropriate to the study of check boxes (you do remember check boxes, don't you), is the note found in Field versus widget attributes, that `checkThisBox`, `defaultIsChecked`, `isBoxedChecked`, and `isDefaultChecked` methods take a widget index, `nWidget`, as a parameter. If you invoke these methods on a Field object that represents one specific widget, the nWidget parameter is optional (and is ignored).

Let's finish this long blog with one last example, we'll use `checkThisBox` in a field of check box widgets.

Example 7.12. These check boxes all have the same name (they are widgets), and were created by row. To test your understanding, enter the index of the widget in the second row, third column. No special testing is done, we simply turn on that widget, see if you're right. We use `checkThisBox` and pass to it the index you entered into the text field.

Reset

☐ ☐ ☐ ☐
☐ ☐ ☐ ☐

How did you do? Did you turn on the check in the second row, third column on the first attempt, if you did, give yourself a gold ★.

The custom validation script for the text fields follows.

```
1  /*Custom Validate*/
2  gbValidate(0,7);
3  if (event.rc && (event.value != "")  ) {
4      var f=this.getField("ck7-12");
5      f.checkThisBox({ nWidget: event.value, bCheckIt: true });
6  }
```

Comments: We check to see if the entry is between 0 and 7, in line (1) (the function `gbValidate` is defined at the document level). If everything is okay, we get the terminal field object (the name of the terminal field is `"ck7-12"`, we then pass `event.value` (that's the value entered and committed to the text field) as the first parameter of `f.checkThisBox`, the second parameter indicates we want to check the box.

Check boxes with the same name but with different export values act much like a radio button field.

We'll eventually get around in later blogs to discussion validation scripts, and other stuff like format, keystroke, and calculation scripts. □

That's pretty much it for now, I simply must get back to my retirement. Adios! DPS