Contents

JavaScript Component SDK

Creating your own JavaScript Components
for Omnis Studio 10/11 and later

Omnis Software Ltd

May 2023

About This Manual

This manual describes how you can create your own JavaScript components to interface with Omnis Studio.

The manual is split into three main sections:

- A tutorial designed to familiarize the developer with building and configuration of a generic JavaScript component.
- Information on configuring your build environment to allow debugging of the C++ component.
- API reference material for the C++ component interface and the JavaScript control

There are two parts to developing a component for the JavaScript Client. The C++ Design Component that is loaded in Omnis Studio and the JavaScript file which is loaded in the client browser.

The C++ part is referred to as the **Component**, while the JavaScript part is referred to as the **Control**.

Creating a JavaScript Client control & component requires an understanding of Omnis Studio, C++ and JavaScript. Experience of creating standard Omnis External Components is also beneficial, though not essential.

C++ Design Component

The Omnis design component determines the Omnis properties, methods and events that are used to manage the state and behavior of a browser based control instance from within the Omnis server code. It also provides a design view of the control for layout within a remote form which is used to generate the HTML template for rendering the control in the client browser.

The design component is constructed using the Omnis external component library interface with additional functionality specific to designing for JavaScript. Therefore, to find details of the structures, messages and functions provided by the component library please refer to the Omnis Studio External Components document.

The jsGeneric example shows the basic framework for a JavaScript external component. The three main generic source files provided with this example can be used as templates and edited for your own component.

JavaScript Control

As well as having a C++ design component, your control must also have a JavaScript implementation. This is the substantive part of the runtime component, and where the majority of your development effort will be focused.

The C++ part of your code should create a bare skeleton for your JavaScript control, in its *innerHTML()* method. The JavaScript control is initially created with this structure, and you should build this up into your complex control.

More information can be found in the Fundamentals section.

Tutorial

This section will take you through the process of building the Generic component, and show steps on how to build on this towards a more complex component.

Creating a JavaScript Client control requires an understanding of C++ and JavaScript. Experience of creating standard Omnis External Components is also beneficial, though not essential.

This tutorial takes the jsGeneric example provided as a basis, and builds up from there.

Completed versions of the Generic 2 & 3 examples are also provided. If you wish to use these, rather than follow through the tutorial to create them from jsGeneric, follow through the Building Generic section, instead using jsgeneric2/3 etc.

Building Generic

C++ Component

The C++ part of your control is referred to as a **Component**, whereas the JavaScript part is referred to as a **Control**.

This section describes how to build the basic Generic component.

Windows

Refer to the Readme.txt for the latest requirements for the Windows build environment including the version of **Microsoft Visual Studio** to use.

Use Visual Studio to open the Windows jsgeneric project file (**jsgeneric.vcxproj**). Everything should be configured to build already.

The project is configured to include:

- **complib** - The **OMNISu.lib**, which is the library that provides the Omnis External Component API.
- **jscomplib** - The **JSCOMPu.lib**, which is the library that provides the Omnis JavaScript Component API.

All you need to do is select the **UNICODE ReleaseWebDesign** (for release builds) or **UNICODE DebugWebDesign** (for debug builds) target from the Configuration droplist, then Build.

By default the component (jsgeneric.dll) will be built into **UDebWebDesign** (for debug builds) or **URelWebDesign** (for release builds). If you wish to change the location that the component is built into, open your project's Properties, and set the **Output Directory** (under General).

You then need to copy the built component (**jsgeneric.dll**) into the **jscomp** folder in the Omnis tree. When Omnis starts it should load the design component, and be available in the Component Store. If you wish to change the location that the component is built into, open your project's Properties, and set the **Output Directory** (under General). For example, you can set Output Directory to **C:\Program Files\Omnis Software 25120\jscomp\** and Visual Studio will place the jsgeneric.dll in your jscomp folder. In addition, you need to set the Output File property for the correct configuration; this can be set via the two dropdown lists at the top of same window. By default, Visual Studio will use the current active configuration.

Note: Your component will require the Microsoft Visual C++ Runtime (2015 or later) to be installed on the machine running Omnis.

Mac

Refer to the Readme.txt for the latest requirements for the macOS build environment (64-bit) including the version of **xCode** to use.

All required project stationary is inside the **jscomp** folder and contains the following.

complib	The **u_complib.framework** which is the library framework providing the Omnis External Component API.
jscomplib	The **u_jscomplib.framework** which is the library framework providing the Omnis JavaScript Component API.
tools	The **omnisrc64** binary which is the Omnis resource compiler used to interpret Omnis *.rc files and build them into bundle resources.

jsgeneric	The project files for the **jsgeneric** example and a template for new components. These build targets which are Core Foundation Bundles to be loaded dynamically by the Omnis core.

The **jsgeneric** folder contains the following standard file hierarchy for building any JavaScript component on the Mac.

English.lproj	A localised resource folder which identifies the bundle's name in the **InfoPlist.strings** file (may not be required) , the component .PNG icons and the component resource file in **jsgeneric.rc.**

The first step, before you build your first component, is to add the Omnis resource compiler (**omnisrc64**) to your environment.

- Copy **omnisrc64** from the **tools** folder in the SDK.
- Paste this into **/Applications/Xcode.app/Contents/Developer/Tools/.**

To build the project, open the **Product** menu, and select:

- Build For ->**Testing** (for a debug build), or
- Build for -> **Profiling** (for a release build)

By default the component (**jsgeneric.u_webdesign**) will be built into **_OSXUnicodeDbg** (for debug builds) or **_OSXUnicode** (for release builds).

You then need to copy the built component (**jsgeneric.u_webdesign**) into the **jscomp** folder in the Omnis app package. When Omnis starts it should load the design component, and be available in the Component Store. You could also set up the project in such a way to have the component (**jsgeneric.u_webdesign)** built directly into the Omnis tree. To achieve this, you will need to select the jsgeneric project from the left panel and then choose the UnicodeCore target.

Afterwards, select the Build Settings section and change the option "**Installation Directory**" to the folder where you wish to place jsgeneric.u_websdesign component. For example, if you want to place the component directly into your jscomp folder, you need to set the Installation Directory to "**/Applications/Omnis.app/Contents/MacOS/jscomp**". Please note that you can have a directory for Debug and one for Release configurations.

js generic.rc

This provides the minimum set of resource ids and strings which are required to identify and pass messages to a component. The format is based on a Windows resource file and if using basic resource keywords can be used across platforms with the Omnis resource compiler tools.

The initial 4 resource IDs should be the names of .PNG image files, which can be used to provide the image for the component in the component store group in Omnis. The first two are for the small (16x16) icon - the second being a high-res (2x) version. The next two are for the large (48x48) icon.

The string table provides resource strings for your component library name, the name of the component object/control within the library and the group that this component library will belong to. The component group must be "Javascript Components".

Typically there would only be a single component object/control defined in the component library.

Resource 31020 is used to identify the version of the JavaScript web component with the first two numbers representing the major and minor version of the component.

Resource 31000 is the name of the procedure in your .CPP file that will be called by Omnis to handle messages sent to your component. If this resource is not present your component will not be called. If this string is not present your procedure should be called 'OmnisEXTCOMPONENT'.

```
1 PNG "icon16x16.png"
2 PNG "icon16x16_2x.png"
3 PNG "icon48x48.png"
4 PNG "icon48x48_2x.png"
STRINGTABLE DISCARDABLE
BEGIN
  1000 "jsGeneric"                  // component library name
  2000 "Generic Control"            // control name within component library
  3000 "JavaScript Components"      // component store group
  31020 "VER 0 0 %%ORFC_VER%%"       // version string
  31000 "jsGenericComponentWndProc"  // window message procedure
END
```

jsgeneric.h

This file includes the super class file that is required to provide the base declarations, e.g. the javaScriptComponent super class which is inherited by the subclassed jsgeneric object.

```
#include "jssuper.h"
```

There are also definitions for the four resource ids which are used to access the description of the component from the jsgeneric.rc resources file. These are passed as part of the WCCcontrol structure to be used by the super class.

```
#define LIB_RES_NAME 1000      // Resource id of library name
#define OBJECT_ID1 2000        // Resource id of con within library
#define COMP_STORE_GROUP 3000  // Resource id of component store group
#define OBJECT_ICON 1         // Resource bitmap id
```

The declaration of the jsgeneric component object class then follows. A component class must always be a subclass of the javaScript-Component class. This class and its webClientComponent base class provide the default message handling, functionality and property support.

```
class jsGenericComponent : public javaScriptComponent
{
  jsGenericComponent(HWND pFieldHWnd, WCCcontrol *pControl);
  ~jsGenericComponent();
  virtual void paintDesign(HDC pHdc);
  static qbool jsGetInnerHTML(HWND pHwnd, WCCcontrol *pControl, webClientComponent *pObject, EXTfldval &pInner
}
```

The jsgeneric component declares a standard constructor and destructor. The constructor is used to initially setup the defaults for the members of the class. It takes an HWND which defines the control's drawing area on the design window and a pointer to a WCCcontrol structure which defines information about a JavaScript component. The destructor can tidy up any data structures in the class and release memory used.

The virtual method paintDesign is declared in the generic object and overrides the default super class implementation in webClient-Component. This method will be called by the super class when your component is to be painted in a design window and your implementation will include the drawing instructions you require to render your component.

A static class member called jsGetInnerHTML is declared based on the JSC_GETINNERHTML function pointer type. This member handles the creation of the inner HTML that is initially sent to render an instance of a browser side JavaScript object.

To set options for the generation of the jsgeneric component's HTML a JSChtmlOptionsGeneric structure is declared which inherits from the base structure JSChtmlOptions. An instance of this structure is passed in the mHtmlOptions member of WCCcontrol structure to be used by the base class. The default constructor should invoke the JSChtmlOptions constructor which will set the standard defaults for the HTML options and then set any changes to those options in its implementation. In this case the option to pass Omnis component instance data via the $dataname property has been disabled.

```
struct JSChtmlOptionsGeneric : public JSChtmlOptions
{
  public:
    JSChtmlOptionsGeneric() : JSChtmlOptions() { mControlHasDataName = qfalse; }
};
```

jsgeneric.cpp

This provides the definitions for the generic component object class and also gives a basic template for the implementation of any component.

At the top of this file are the arrays which describe the events and properties that are defined for a component. These arrays and a count of their members are passed via an instance of the WCCcontrol structure to a component's superclass to provide information about a component.

There is an array of ECOmethodEvent structures to describe the events of the component. This is passed in the mEvent member of the WCCcontrol structure. The basic jsgeneric component does not define any events, but this array exists for use as a template for further expansion.

```
ECOmethodEvent jsGenericEvents[] =
{
  0
};

#define jsGenericEvent_Count (sizeof(jsGenericEvents) / sizeof(ECOmethodEvent))
```

There is a jsGenericEvent_Count definition to provide a count of the number of events defined in the jsGenericEvents array. This is passed in the mEventCount member of the WCCcontrol structure.

There is an array of ECOproperty structures to describe the properties of the component. This is passed in the mProperties member of the WCCcontrol structure. This example shows an empty template for the property array and there are no custom properties used by the jsgeneric component.

```
ECOproperty jsGenericProperties[] =
{
  0
};

#define jsGenericProperty_Count (sizeof(jsGenericProperties) / sizeof(ECOproperty))
```

There is a jsGenericProperty_Count definition to provide a count of the number of properties defined in the jsGenericProperties array. This is passed in the mPropertyCount member of the WCCcontrol structure.

Following the array definitions for a component is the actual implementation of the component object. In this case it is the jsGenericComponent, starting with the constructor.

The constructor passes its HWND and WCCcontrol values through to its javaScriptComponent superclass constructor. The superclass can utilize these parameters to respond to informational messages, handle component property changes and component method calls, and when it needs to call an overridden method in your subclass. In the generic constructor the value of the mNoDesignName member of the webClientComponent base class is set to qtrue to prevent the component's object name from being drawn on the control when it is displayed in a design window.

```
jsGenericComponent::jsGenericComponent(HWND pFieldHWnd, WCCcontrol *pControl) : javaScriptComponent(pFieldHWnd
{
  mNoDesignName = qtrue;
}
```

The destructor follows and since the jsGenericComponent subclass has nothing to tidy up the body of ~jsGenericComponent is empty.

The paintDesign method is a virtual method of the webClientComponent base. A sub-class overrides this to implement its custom drawing code. Typically, you draw within the client area of the component's window (HWND), i.e. the content area of the interface. The GDI and HWND APIs in the external component library can be used to specify 2D drawing primitives, e.g. framing/filling regions with foreground and background colors either as solid colors or patterns, clipping the drawing area, using transformations, drawing bitmaps, drawing text, etc. There is also a set of API calls to draw standard system themed controls, e.g. push buttons, check boxes, radio buttons, etc. The appearance of the component in design mode should mimic as closely as possible the appearance of the associated JavaScript control when rendered in a browser. The component object properties, whether standard or custom, can be used to set the appearance of the component in its design window. The drawing of the component's non-client area such as its border is automatically handled by Omnis and the component's superclass along with the drawing of some built in appearance properties. In the case of jsgeneric, its instance of the WCCcontrol structure will set the EXTIPJS_FLAG_BACKCOLOR_AND_BACKAPLPHA flag in its mCompFlags member to indicate that it requires the built-in $backcolor and $backalpha appearance properties. Changes to these properties will be handled automatically and the component will be redrawn.

The declaration of paintDesign for jsgeneric is an empty template for this example. It has no custom content and therefore it is sufficient to let the base class render this automatically based on its appearance properties.

```
void jsGenericComponent::paintDesign(HDC pHdc)
{
}
```

The next method is the implementation of jsGetInnerHTML which is used to generate the inner HTML template which is sent to the JavaScript object in the browser. This method has its signature defined by the JSC_GETINNERHTML function pointer type.

The next method is the implementation of jsGetInnerHTML which is used to generate the inner HTML template which is sent to the JavaScript object in the browser. This method has its signature defined by the JSC_GETINNERHTML function pointer type.

```
qbool jsGenericComponent::jsGetInnerHTML(HWND pHwnd, WCCcontrol *pControl, webClientComponent *pObject, EXTfld
{
qbool deleteObject = qfalse;
jsGenericComponent *object = (jsGenericComponent *) pObject;
  if (!object)
  {
    // Temporary object to hold property values - we need to use the property values in the supplied object wh
    // generating HTML for an open design window - the temporary object caters for the case when there is no o
    deleteObject = qtrue;
    object = new jsGenericComponent(0, 0);
  }
  str255 innerTemplate = str255(QTEXT("<div $$></div>"));
  pInner.setChar(innerTemplate);
  //insert the id of the client
  jsInsertId(pHwnd, pInner, qfalse);
  // insert the control's style, tab order, etc.
  jsInsertStyle(pHwnd, pControl, pInner, pWidth, pHeight, object);
  if (deleteObject)
    delete object;
  return qtrue;
}
```

The pObject parameter is used to pass a pointer to an instance of a webClientComponent, i.e. in this case it is the pointer to an instance of a jsGenericComponent. Since this is passed via the base class type it is cast to the correct type for the jsGenericComponent sub-class. This method can be called to generate the HTML either when a design window is open or at runtime when class format notation is used to set a component's property in the form. In the latter case there is no object instance and pObject will be NULL. This requires that a temporary object is created to hold property values set from the current format. In this case NULL values are passed in the two parameters of the constructor. An Omnis str255 is used to form the template for the inner HTML (a string of up to 255 characters) and this string is stored in the pInner parameter which is an Omnis EXTfldval data storage object. A template string is typically an HTML element, in this case a DIV, where the '$' placeholder is used to mark where an attribute will be substituted.

The method then makes use of utility methods in the javaScriptComponent base class to perform the placeholder substitutions. Calling jsInsertId causes the first marker to be replaced with the HTML id attribute which is used to give the element within the

HTML a unique identifier. The identifier value will be the component group's class name, i.e. the remote form name, followed by the unique numeric identifier of the component within the group and then the suffix for the level of the element. Since this is the inner HTML for the web control its suffix will be 'client'. Therefore the full literal substituted could be something like 'testForm_1001_client'. Typically, your own inner HTML method will always call this method to place the control's id in the HTML so that the JavaScript code can automatically identify the element. To add the correct HTML styles, dimensions, etc., for the control, the next substitution is made via the jsInsertStyle method. This method adds the HTML inline style attribute to the inner HTML with a value containing the CSS property values based on the component's properties, e.g. background-color from $backcolor.

Any temporary object created is then deleted and the method returns qtrue to signal success. The HTML generated is passed back in the pInner parameter to be used by the super class.

A jsHTMLoptions variable is used to store an instance of the JSChtmlOptionsGeneric structure declared in jsgeneric.h. This is used to create a set of options for the component when it generates its HTML styles, e.g. when using jsInsertStyle.

This structure is initialized with defaults from its JSChtmlOptions base structure and will be passed in the mHtmlOptions member of the instance of the WCCcontrol structure which is next defined.

```
static JSChtmlOptionsGeneric sHTMLoptions;
static WCCcontrol sJSControl =
{
  0,                                                   // Resource base - unused for JavaScript components
  LIB_RES_NAME,                                        // Resource id of library name
  OBJECT_ID1,                                          // Resource id of control within library
  OBJECT_ICON,                                         // Resource bitmap id
  jsGenericEvent_Count,                                // Count of events
  jsGenericEvents,                                     // Events
  jsGenericProperty_Count,                             // Count of properties
  jsGenericProperties,                                 // Properties
  0, 0,                                                // First and last constant
  EXTIPJS_FLAG_BACKCOLOR_AND_BACKALPHA,                // iPhone/JavaScript mCompFlags
  WCC_FLAG_CANFOCUS,                                   // mWccFlags
  COMP_STORE_GROUP,                                   // Resource id of component store group
  0,                                                   // Count of methods
  0,                                                   // Methods
  0,                                                   // Function to make class notation object
  0,                                                   // Resource id of of custom tab name
  0,                                                   // Fixed landscape width for control (zero if any width
  0,                                                   // Fixed landscape height for control (zero if any heig
  0,                                                   // Fixed portrait width for control (zero if any width
  0,                                                   // Fixed portrait height for control (zero if any heigh
  &sHTMLoptions,                                       // options for HTML generation
  jsGenericComponent::jsGetInnerHTML,                  // static to generate inner HTML
  0,                                                   // No control-specific styles function
  "ctrl_generic"                                       // Control name
};
```

The sJSControl variable uses the members of its structure to provide information about the Generic component.

- The name of the library, component and the icon for the component are defined based on the resource ids in jsgeneric.h which refer to the values in jsgeneric.rc.
- The component events and properties are defined based on the event and property arrays previously specified. If not used then 0 can be passed for the count and NULL for the array value.
- EXTIPJS_FLAG_xxx/EXTJS_FLAG_xxx definitions are used to flag the setting of specific JavaScript properties and behaviors that are required. In this case the Generic component will have the $backcolor and $backalpha properties.
- Likewise, to flag web control specific behaviors the WCC_FLAG_xxx definitions are provided. This indicates that the HTML control will accept the focus.
- To place the component in the JavaScript Components group in the Omnis Component Store the resource id is set from the definition in jsgeneric.h.

- The sHTMLoptions variable is passed to set HTML generation options for the inner HTML.
- The jsGetInnerHTML method is passed as the method that will be called to set the inner HTML.
- The name of the JavaScript object for the web control. The JavaScript name for the Generic control is 'ctrl_generic'.

This variable is passed when constructing a new instance of the jsGenericComponent object and is used by the object's superclass to provide information about the component in response to messages sent from Omnis.

The final section of the implementation of the Generic component declares an Omnis WndProc message procedure. This must be defined in order for Omnis to send messages to the component to instruct it to perform certain operations, e.g. to paint the component in a design window, to get and set property values, and so on. The name of this function must match the string resource with id 31000 in the component's resource file; jsgeneric.rc.

```
extern "C" LRESULT OMNISWNDPROC jsGenericComponentWndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lPara
{
  ECOsetupCallbacks(hwnd, eci);
  switch (message)
  {
    case ECM_OBJCONSTRUCT:
    {
      jsGenericComponent *object = new jsGenericComponent(hwnd, &sJSControl);
      lParam = (LPARAM) object;
      break;
    }
  }
  return JSCdefWindowProc(hwnd, message, wParam, lParam, eci, &sJSControl);
}
```

This procedure must contain the ECOsetupCallbacks API call to always setup the call from Omnis. A **switch** is used to process the messages that are to be handled by this procedure. With each **case** statement matching a **message** id that the component is handling. Whether the message is handled in the switch or not it must be sent through to the default JSCdefWindowProc message handler.

The ECM_OBJCONSTRUCT message is sent to create an instance of the component object. This occurs when a remote form design window is being opened and contains this component. To respond to this message an instance of the object is created using the constructor to pass in the hwnd for the child window of the control and an instance of a WCCcontrol structure to describe the control. In this case a jsGenericComponent object is created with the sJSControl variable passed through the constructor to the superclass. The new instance object's pointer is then assigned to the lParam parameter in order for the superclass and Omnis to use the component.

The JavaScript Control

The JavaScript part of your control is referred to as a **Control**, whereas the C++ side of your control is referred to as a **Component**.

This section describes how to build the basic Generic control - i.e. the JavaScript part.

You can use anything from a simple text editor, such as Notepad, up to a full IDE, such as **Webstorm** to write your JavaScript control. We recommend Webstorm as it will make your life a lot easier since it includes syntax highlighting, error checking, intellisense, and so on, which all make working with the code a much nicer experience, and prevent a lot of simple mistakes.

Prepare the JavaScript Part

- Copy the ctl_generic.js file from the SDK, and paste it into the **html/scripts** folder in your Omnis tree.
- Locate **jsctempl.htm** in the **html** folder in the Omnis tree, and edit the file.
- Add a <script> tag to the <head> to load the **ctl_generic.js** file **after** all other .js files.

```
<!-- Omnis Studio JavaScript client scripts -->
<script type="text/javascript" src="scripts/ssz.js"></script>
<script type="text/javascript" src="scripts/omjsclnt.js"></script>
<script type="text/javascript" src="scripts/omjqclnt.js"></script>
<script type="text/javascript" src="scripts/ctl_generic.js"></script> <!-- LOAD GENERIC -->
```

- *If you are using the initial 10.2 release, you will also need to make these changes to the design view template in **html/design/jsctempl.htm**.*
- Clear your browser's cache, in case it has cached an old copy of your form's .htm file

Running Generic

Once you have built and placed both the JavaScript and C++ parts into your Omnis tree, you're ready to run the Generic control.

- Create a Remote form in Omnis.
- In the Component Store you should see the Generic component. Drag this onto your form.
- By default, the component has inherited a few properties (due to properties of its WCCcontrol structure), such as $backcolor, $alpha etc - set these as you wish.
- Test the form.

If everything was set up properly, you should see a running Generic control on your form in your browser. It is a very simple control, but proves the concept.

Extending Generic

Now that you understand the basics of building and deploying a control, you may want to extend the generic control with additional functionality.

Generic 2

This section describes how you can extend the Generic example to include the standard properties **$text**, **$textcolor** and **$effect**. These new properties will provide values to display text in the JavaScript control and allow it's border to be altered.

As well as adding these properties to the component, and acting on them at runtime, they will be used to affect the drawing of the component in design mode. This example shows how to add the required custom drawing code in the component.

C++ Part

jsgeneric.rc

To identify this component and its control the value of string resources 1000 and 2000 are updated.

```
BEGIN
  1000 "jsGeneric2" // component library name
  2000 "Generic2 Control" // control name within component library
  ..
END
```

jsgeneric.h

The jsgeneric2 example will allow text to be displayed on the JavaScript control. The text value can be specified by adding support for the built-in **$text** property. This property value has to be managed by the **jsGenericComponent** subclass. Therefore, the **mText** private member is added to the subclass definition in **jsgeneric.h** and is typed to allow storage of up to 255 characters. To manage access to the property data the subclass overrides the attributeSupport method which is a virtual method of the webClientComponent base. As part of the data management process it is necessary to pass values to and from Omnis using data storage objects. In this case two accessor methods on the text data are defined.

```
class jsGenericComponent : public javaScriptComponent
{
  private:
    str255 mText;
  public:
    ..
    virtual qlong attributeSupport(LPARAM pMessage, WPARAM wParam, LPARAM lParam, EXTCompInfo* eci);
    ..
    qbool getText(EXTfldval& fval);
    qbool setText(EXTfldval& fval);
    ..
};
```

jsgeneric.cpp

To support the built-in **$text**, **$textcolor** and **$effect** properties, and to have them appear on the Omnis Property Manager, the properties array in **jsgeneric.cpp** needs to add the following entries.

```
ECOproperty jsGenericProperties[] =
{
  anumText, 0, fftCharacter, EXTD_FLAG_PROPTEXT|EXTD_FLAG_FAR_SRCH, 0, 0, 0,
  anumTextColor, 0, fftInteger, EXTD_FLAG_PROPTEXT|EXTD_FLAG_PWINDCOL, 0, 0, 0,
  anumEffect, 0, fftInteger, 0, 0, 0, 0,
};
```

Each entry is defined by setting the required values in it's ECOproperty structure. Unused entries are passed as 0. The jsgeneric2 example uses three of the entries.

- **mPropID** - The property identifier which in this case is the attribute number of a built-in property, e.g. **anumText** for **$text**.
- **mDataType** - The type of data that the property is storing, e.g **fftCharacter** for character data.
- **mFlags** - Flags to specify the behavior of the property, e.g. **EXTD_FLAG_PROPTEXT|EXTD_FLAG_FAR_SRCH** to indicate that it should appear on the text tab of the Omnis Property Manager and will be searched on during find and replace operations.

The **mText** data member of the jsGenericComponent is initialized with a default value in the constructor.

```
jsGenericComponent::jsGenericComponent(HWND pFieldHWnd, WCCcontrol *pControl) : javaScriptComponent(pFieldHWnd
{
  mNoDesignName = qtrue;
  mText = QTEXT("Generic Two");
}
```

The implementation of attributeSupport manages access to the properties that have been defined. A message id is passed in the first parameter to indicate the request being made and a switch is used to take action for each message. Generally, there are three messages that must be handled within this method.

- **ECM_PROPERTYCANASSIGN** - Determines if a property can be written to. If YES then return 1, otherwise return 0.
- **ECM_SETPROPERTY** - Sets a property value by retrieving the value passed from Omnis in the first external parameter of the eci structure. Copy that value into the relevant component object member. Return true/false for success/failure.
- **ECM_GETPROPERTY** - Gets a property value by copying the relevant component member into a data object. Pass this value back to Omnis by adding a parameter to the provided eci structure which contains a copy of that data object. Return true/false for success/failure.

```
qlong jsGenericComponent::attributeSupport( LPARAM pMessage, WPARAM wParam, LPARAM lParam, EXTCompInfo* eci )
{
  GDIignore(&wParam); GDIignore(&lParam);
  switch( pMessage )
  {
    case ECM_PROPERTYCANASSIGN:
    {
      return 1L;
    }
    case ECM_SETPROPERTY:
    {
      EXTParamInfo* param = ECOfindParamNum( eci, 1 );
      if (param)
      {
        EXTfldval fval( (qfldval)param->mData );
        qbool ret = qfalse;
        int propid = ECOgetId(eci);
        switch( propid )
        {
          case anumText: ret = setText(fval); break;
        }
        return ret;
      }
      break;
    }
    case ECM_GETPROPERTY:
    {
      EXTfldval fval; qbool ret = qfalse; int propid = ECOgetId(eci);
      switch( propid)
      {
        case anumText: ret = getText(fval); break;
        default: return 0L;
      }
      ECOaddParam(eci,&fval);
      return ret;
    }
  }
  return 0;
}
```

The wParam and lParam parameters of this method are generally unused and are not required to process these messages.

The eci parameter of this method is used to identify the id of the property via the **ECOgetId** external component method. In this case both ECM_SETPROPERTY and ECM_GETPROPERTY need to retrieve the id in order to access the correct member in the component. In these cases a switch is used on the property id to process each member value. If multiple properties are being managed by a component with some writable and others read-only, a similar mechanism could be used with the ECM_PROPERTYCANASSIGN message. If all properties are writeable, it is sufficient to just return true as is done here.

The ECM_SETPROPERTY message requires that the data held in the first parameter in the eci structure is retrieved. The **ECOfindParamNum** function locates the first **EXTParamInfo** parameter in the **EXTCompInfo** structure of eci. Within the **EXTParamInfo** structure is the **mData** member storing a pointer to the new value which is an Omnis qfldval data item. This is used to create the generic **EXTfldval** data object as a container for the underlying data. The request to set the **$text** property can then invoke the component's **setText** method. This will copy the new value into the component's **mText** member.

The ECM_GETPROPERY message locates the correct property from the id stored in eci and in the case of **$text** will use the component's **getText** method. The value is copied into an **EXTfldval** data object and sent back to Omnis by adding it to eci as a parameter using the **ECOaddParam** function.

The jsgeneric2 example needs to draw the value of **$text** in the design field's window in Omnis. The subclass of the component needs to do this drawing in its overridden paintDesign method. Whenever a portion of the component window needs repainting, this method will be called, e.g. if an appearance property changes such as **$backcolor**.

```
void jsGenericComponent::paintDesign(HDC pHdc)
{
  EXTfldval fval;
  qrect clientRect;
  WNDgetClientRect(mHWnd, &clientRect);
  HFONT font = 0, oldFont =0;
  qshort textLen = mText.length();
  // Draw text centred
  if(textLen > 0)
  {
    font = GDIcreateFont(&textSpec().mFnt, textSpec().mSty);
    oldFont = GDIselectObject(pHdc, font);
    GDIsetTextColor(pHdc, textSpec().mTextColor);
    clientRect.top += (clientRect.height() - GDIfontHeight(pHdc))/2;
    GDItextBox(pHdc, &clientRect, &mText[1], &textLen, mText.maxLength(), jstCenter);
  }

  // Clean-up
  if (font != 0)
  {
    GDIselectObject(pHdc, oldFont);
    GDIdeleteObject(font);
  }
}
```

The paintDesign method takes one parameter, **pHdc**, which is the **HDC** window device into which the drawing will occur, i.e. the design window. The characteristics of this device can be read and changed to affect the drawing operations, e.g. current font, current text color, etc. Typically the custom content is to be drawn in the client area (the content area) of the component's child window and so initially it is necessary to get the coordinates of the client rect using the **WNDgetClientRect** function. The **mHWnd** member is a public member of the component's webClientComponent base class which identifies the **HWND** child window from which its client area coordinates can be returned. The clientRect local variable will hold the returned client area coordinates, i.e. top, left, bottom, right. These coordinates are always local to the **HWND** client area so top and left are always zero.

The function requires the character string stored in the **mText** member (the value of **$text**) to be drawn centrally with the current value of **$textcolor**. This is only done if there is actually text to draw. That text needs a font to be rendered in and the **GDIcreateFont** function is used to create that font. The webClientComponent base class stores the component's current Omnis text specification in its mTextSpec member. This is a structure of type **GDItextSpecStruct** containing the **qfnt**, **qsty**, **qjst** and **qcol** for text display. The **textSpec** method returns the **mTextSpec** structure and from this the font and style can be passed to **GDIcreateFont** to create an HFONT pointer to use for drawing. The HFONT is used to set the new font characteristics for the **HDC** by calling the **GDIselectObject** function. The old font characteristics are returned and will be restored after this component has finished drawing.

To set the text color in the **HDC** the **GDIsetTextColor** function is called passing in the **mTextColor** member of the text specification. This is a **qcol** value set from **$textcolor**.

The actual drawing of the text string is performed by the **GDItextBox** function. This needs a rectangle in which to draw the text. Since the text is to be drawn centred vertically and horizontally, the vertical position of the **clientRect** is recalculated based on the line height of the current font using the **GDIfontHeight** function. The **clientRect** is passed to **GDItextBox** with the resized client rectangle, a C string pointer to the text, its length, total length and justification. Using **jstCenter** ensures that the text is centered in the rectangle.

All that remains to be done is to cleanup by resetting the font in the **HDC** using **GDIselectObject** and then to delete the **HFONT** which is no longer required.

To get and set values of the **mText** member of the generic component there are two accessor functions. These take an **EXTfldval** reference as their **fval** parameter in order to pass data to/from Omnis.

```
qbool jsGenericComponent::getText(EXTfldval& fval)
{
  fval.setChar(mText);
  return qtrue;
}
qbool jsGenericComponent::setText(EXTfldval& fval)
{
  fval.getChar(mText);
  inval();
  return qtrue;
}
```

The **getText** method will set the value of its **fval** parameter to store a copy of **mText**. This is then used in response to an ECM_GETPROPERTY message. The **setText** method uses the value of **fval** to set the value of **mText** in response to an ECM_SETPROPERTY message. Since the text has been altered it is necessary to redraw the component window. Therefore, the inval method is called to invalidate the client area and to force the paintDesign method to be called.

The implementation of **jsGetInnerHTML** needs to be expanded in this example to cater for the text that needs to be shown on the JavaScript control.

```
qbool jsGenericComponent::jsGetInnerHTML(HWND pHwnd, WCCcontrol *pControl, webClientComponent *pObject, EXTfld
{
  ..
  EXTfldval fval;
  if (!object)
  {
    ..
    ECOgetProperty(0, anumText, fval);
    fval.getChar(object->mText);
  }
  str255 innerTemplate = str255(QTEXT("<div $$$>$</div>"));
  ..
  fval.setLong(jstCenter);
  ECOsetProperty(pHwnd, anumAlign, fval);
  jsInsertStyle(pHwnd, pControl, pInner, pWidth, pHeight, object);
   jsInsertBorder(pHwnd, pInner);
  jsInsert(object->mText, pInner); //text
  ..
}
```

To manipulate property values an **fval** data storage variable is used which is of type **EXTfldval**. When the HTML needs to be generated and there is no instance of a component object, e.g. a property has been changed notationally, this requires that a new instance of the **jsGenericComponent** is created. This new component instance is assigned values for the properties that it handles by copying them from the current format instance using the **ECOgetProperty** function. This function uses an **EXTfldval** parameter to pass back the data. When setting up the **mText** member of the generic component a request is made for the $text property value using the **anumText** property identifier. The value is returned into **fval** and a string version copied into **mText** using the **getChar** method. This process would need to be repeated for all member values of a component that are used to generate the HTML.

This example needs to expand the inner HTML by inserting the border information and the text string. Therefore, there are two additional placeholder markers added to the HTML template.

To ensure that the text is central, the $**align** value is set to **jstCenter** using the **ECOsetProperty** function to handle the built-in property. The **fval** variable is used to pass the alignment constant value. When the style information is calculated and substituted into the HTML template using jsInsertStyle this will add the CSS **text-align** property with a value of **center**.

The border style is added based on the **$effect** property value. This substitutes the placeholder marker for an attribute name/value pair, where the name is **data-effect** and the value is the effect constant. The JavaScript control will automatically interpret this attribute and map the value to render the correct border style.

The text is then added to the inside of the DIV by using jsInsert to add the value of the **mText** string.

For generic2 there are only two changes to the WCCcontrol structure to reflect the changes to the component's control.

- The **mCompFlags** member needs the additional EXTIPJS_FLAG_EFFECT flag to add the **$effect** property. This will allow the border style to be changed.
- The **mJscCtrlName** member needs to be altered since this component's JavaScript control name is called "ctrl_generic2".

JavaScript Part

The $textcolor and $effect properties are handled well in this case by the base class. As such, you do not need to do anything for these to display and handle get/set values.

However, the $text is a little more involved, as the text could be shown in any number of ways, so it's up to you how to implement this property. In this case, the text string has been injected as the content of the client element, by the component's JSC_GETINNERHTML.

Center the Text

If you have completed the C++ part of generic2, and run the form, you'll see that the text is shown, but is not vertically centered.

Locate your control's updateCtrl() method, and add the following code:

```
var elem = this.getClientElem();
// Center the text vertically:
elem.style.lineHeight = elem.style.height;
```

In this instance, the text is just set as the content of our client element. The line height of the client element is set to its height, so that the text is vertically centered.

This is added to the updateCtrl() method so that it is also called if the size of the control changes, etc.

However, you then need to tell the control to do an update() as soon as it has finished constructing itself.

Add the following to your control's init_ctrl_inst() method, just before the return:

```
this.update();
```

Handle Read Requests for $text

Next create a "class" variable to hold the current content of $text.

Add the following code to init_ctrl_inst()

```
//Control-specific initialization:
var client_elem = this.getClientElem();
// Read initial value of $text. In this instance it is just the content of the client element.
this.mText = client_elem.innerHTML;
```

Here you can read the text from the client element of our control, and store it in a "class" variable named "mText".

Now you need to add handling for this property to our getProperty() implementation:

```
ctrl.getProperty = function(propNumber) {
  switch (propNumber) {
    case eBaseProperties.text:
    return this.mText;
  }
  return this.superclass.getProperty.call(this, propNumber); //Let the superclass handle it,if not handled her
};
```

Here you can add your own handling if the property which is being requested is $text (eBaseProperties.text). All other property requests fall back to the base class.

Handle Write Requests to $text

By default, getCanAssign() returns false for $text, so you need to add handling for $text and return true in our implementation of getCanAssign():

```
ctrl.getCanAssign = function(propNumber) {
  switch (propNumber) {
    case eBaseProperties.text:
      return true; // Allow $text to be assigned to.
  }
  return this.superclass.getCanAssign.call(this, propNumber); //Let the superclass handle it,if not handled he
};
```

Now that you can assign to $text, you need to add some handling to our implementation of setProperty() for when this happens.

```
ctrl.setProperty = function( propNumber, propValue )
{
  if (!this.getCanAssign(propNumber)) // Check whether the value can be assigned to
    return false;
  switch (propNumber) {
    case eBaseProperties.text: // Set the text as appropriate for this control.
    this.mText = propValue;
    var client_elem = this.getClientElem();
    client_elem.innerHTML = propValue;
    return true;
  }
  return this.superclass.setProperty.call( this, propNumber, propValue ); //Let the superclass handle it, if n
};
```

Here you can call getCanAssign() to check whether or not you can assign to this property. Then, if the property is $text, you set mText to the new value, and update the html content of your client element to reflect this.

Generic 3

In this section the Generic component will be further extended to include:

- **A custom propert**
- **Event Handling**
- **Using a $dataname variable**

Add a Custom Property - C++ Part

The following section describes the process by which a custom property, named **$togglecolor**, is added to a control.

jsgeneric.rc

In order to add a custom property, the first thing to do is to make a reference in the resource file jsgeneric.rc:

The following line should be added:

```
BEGIN
..
4000 "$togglecolor:If true, the control's color will be toggled on each click."
..
END
```

This defines a **resource ID** (4000) for the property, as well as the name of the property (**$togglecolor**) and description (the content after the colon).

jsgeneric.h

You need to add a constant (**cToggleColor**) to hold the resource ID of the custom property, as well need to provide accessor methods to handle getting and setting the value of this property. You will also use a class variable (**mToggleColor**) to hold the current value of the property.

These should be declared in the **jsgeneric.h** header file:

```
const attnumber cToggleColor = 4000;
class jsGenericComponent : public javaScriptComponent
{
  public:
    str255 mText;
    qshort mToggleColor;
    ..
    qbool getToggleColor(EXTfldval& fval);
    qbool setToggleColor(EXTfldval& fval);
```

jsgeneric.cpp

The first step is to add the new property to the ECOproperty structure, so that the property is added to the component, and shown in the property manager.

```
ECOproperty jsGenericProperties[] =
{
  anumText, 0, fftCharacter, EXTD_FLAG_PROPTEXT|EXTD_FLAG_FAR_SRCH, 0, 0, 0,
  ..
  cToggleColor, cToggleColor, fftBoolean, EXTD_FLAG_PROPAPP, 0, 0, 0, // Add $togglecolor property
};
```

In this case, the **$togglecolor** property is added, defined from the resource ID specified by **cToggleColor** (4000), as a **Boolean** type, under the **Appearance** tab of the property manager.

The member variable which is used to hold the current value of the property (**mToggleColor**), should be initialized in the component's **constructor**.

```
mToggleColor = qfalse;
```

The getter/setter accessor methods which were declared in the header file now need to be implemented.

```
qbool jsGenericComponent::getToggleColor(EXTfldval& fval)
{
  fval.setLong(mToggleColor);
  return qtrue;
}
qbool jsGenericComponent::setToggleColor(EXTfldval& fval)
{
  mToggleColor = fval.getLong();
  return qtrue;
}
```

Handling for the new property must be added to the attributeSupport() method, to call the accessor methods when a request is made to get or set the property's value.

```
case ECM_SETPROPERTY:
  ..
  switch( propid)
  {
    case anumText: ret = setText(fval); break;
    case cToggleColor: ret = setToggleColor(fval); break;
  }
  ..
case ECM_GETPROPERTY:
  ..
  switch( propid)
  {
    case anumText: ret = getText(fval); break;
    case cToggleColor: ret = getToggleColor(fval); break;
    default: return 0L;
  }
  ..
```

Finally, the JSC_GETINNERHTML method needs to be extended to insert the value of **$togglecolor** into the HTML which is sent to the client.

Firstly, another “**$**” placeholder character needs to be added to **innerTemplate**, inside the div.

```
str255 innerTemplate = str255(QTEXT("<div $$$$>$</div>"));
```

This will then be replaced by an attribute name/value pair.

Add the following code **before** the last insert call (i.e. **before jsInsert(object->mText, pInner);**), so that it replaces one of the first four **$**’s.

```
jsInsertNum("data-togglecolor", object->mToggleColor, pInner);
```

This will add an HTML attribute to the client element, which can be queried by the JavaScript code to ascertain the (initial) value of the **$togglecolor** property.

It is also necessary to add handling to load a value for the property in the case that no instance of the object exists:

```
..
if (!object)
{
  // Temporary object to hold property values - we need to use the property values in the supplied object when
  // generating HTML for an open design window - the temporary object caters for the case when there is no ope
  deleteObject = qtrue;
  object = new jsGenericComponent(0, 0);
  ECOgetProperty(0, anumText, fval);
  fval.getChar(object->mText);
  //Add handling for custom $togglecolor property:
  ECOgetProperty(0, cToggleColor, fval);
  object->mToggleColor = fval.getLong();
}
..
```

Add a Custom Property - JavaScript Part

The first step in handling properties is to create a constant for the variable’s ID. The IDs of built-in properties are accessed via the eBaseProperties enumerated type. As such, it is recommended that you create your own ‘enumerated type’ (an object) to hold all the IDs for the custom properties of your control.

Convention is to name this **‘PROPERTIES’**.

As these are constant values, you should create this outside any of your control’s methods (but inside the function which returns your control prototype), so that it will only be created once, rather than each time an instance of your control is created.

```
var PROPERTIES = {
  "toggleColor": 4000
};
```

Here, **PROPERTIES.toggleColor** has a value of 4000 - this matches the property's resource ID, defined in jsgeneric.rc.

The next step is to define a member variable of your control, to hold the current value of the property. This should be defined at the end of init_class_inst().

```
this.mToggleColor = false;
```

You must add handling for getting and setting the value of the property. This involves editing getProperty(), getCanAssign() and setProperty().

getProperty()

```
..
switch (propNumber) {
 ..
  case PROPERTIES.toggleColor:
    return this.mToggleColor;
  ..
```

If the propNumber requested matches that of $togglecolor, you only need to return the value of mTogglecolor.

getCanAssign()

```
..
switch (propNumber) {
  case eBaseProperties.text:
  case PROPERTIES.toggleColor:
    return true;
..
```

This is almost always a very simple method - in this case, just return true.

setProperty()

```
..
switch (propNumber) {
..
  case PROPERTIES.toggleColor:
    this.mToggleColor = propValue;
..
```

Again, this is a very simple implementation - **mToggleColor** is set to the value passed in as **propValue**.

All that remains, as part of the basic handling of the property, is to read the initial value, which is inserted into the HTML of the client element as part of the component's JSC_GETINNERHTML method.

In the init_ctrl_inst() method of your control, you should read the value of the **data-togglecolor** attribute from the client element, and call your control's setProperty() method to set this initial value.

```
..
this.mText = client_elem.innerHTML;
var attValue = client_elem.getAttribute("data-togglecolor") ? true : false;
this.setProperty(PROPERTIES.toggleColor, attValue);
..
```

This particular attribute will not be present if the value is 0. The use of the ternary operator allows a quick, inline, handling of this, and a casting to boolean.

The **behavior** of this property will be implemented as part of the event handling.

Event Handling – C++ Part

This section describes the process of adding functionality to handle click events, passing custom parameters.

The first step is to define the custom parameters which will be sent with the click event.

Add string resources for the custom parameters

Edit the **jsgeneric.rc** file, adding the following entries:

```
BEGIN
..
6000 "xPos:The x position of the click within the control"
6001 "yPos:The y position of the click within the control"
..
END
```

Create constants to point to those string resources

Add a couple of new constants (**cEvParamXPos** & **cEvParamYPos**) to the **jsgeneric.h** file, with values equal to the resource IDs specified (6000 & 6001).

```
const qshort cEvParamXPos = 6000;
const qshort cEvParamYPos = 6001;
```

Create an ECOparam array

An ECOparam structure is needed to pass parameters to your events.

Create an array of ECOparam structures above your ECOmethodEvent structure, including your two custom parameters:

```
ECOparam jsGenericEventParams[] =
{
  cEvParamXPos, fftInteger, 0, 0,
  cEvParamYPos, fftInteger, 0, 0
};
```

Now that the custom parameters are defined, the method itself can be declared.

Create an ECOmethodEvent array

The ECOmethodEvent array determines which events your component supports. In this case, you need to add a click event to your control.

```
ECOmethodEvent jsGenericEvents[] =
{
  ECE_CLICK, 0, 0, 2, &jsGenericEventParams[0], 0, 0
}
```

Here the event has been defined as a standard click event (**ECE_CLICK**), which receives **two** parameters starting from index **0** of the **jsGenericEventParams** array.

This will add an **evClick** event to the **$events** property of your component (by default, events are disabled for a component, and need to be enabled through the component's $events property.).

Event Handling - JavaScript Part

The first step on the JavaScript part is to add an event listener to the element you wish to trigger the event. In this case, the generic control is very simple and only has a single element (the client element), so it is to this that you should attach the event listener.

Create the following helper method in your control:

```
/**
* Adds a click handler if the device doesn't support touch, or equivalent touch handlers if it does.
* @param elem Element to add a click/touch handler to
*/
ctrl.addClickHandlers = function(elem) {
  if (!this.usesTouch)
    elem.onclick = this.mEventFunction;
  else {
    elem.ontouchstart = this.mEventFunction;
    elem.ontouchend = this.mEventFunction;
  }
};
```

This is a simple helper method which will add event listeners as appropriate for the client (substituting for touch events of the client is touch-based).

The main thing to note here is the use of the mEventFunction value - assigning this to a standard event means that your control's handleEvent() method will be called whenever that event is triggered.

This method should then be called from your init_ctrl_inst() implementation.

```
..
this.addClickHandlers(client_elem);
..
```

The next step is to add handling to the handleEvent() implementation of your control:

```
ctrl.handleEvent = function( event )
{
  if (!this.isEnabled()) return true; // If the control is disabled, don't process the event.
  switch (event.type) {
    case "click":
      return this.handleClick(event.offsetX, event.offsetY);
    case "touchstart":
      this.lastTouch = new Date().getTime(); // Note the time of the touch start.
      this.touchStartPos = {
        x: event.changedTouches[0].clientX,
        y: event.changedTouches[0].clientY
      }; // Note the starting position of the touch.
      // event.preventDefault(); // Prevent the click event, as we have handled it in the touch event.
      return;
    case "touchend":
      var time = new Date().getTime();
      if (time - this.lastTouch < 500) { //Treat as a click if less than 500ms have elapsed since touchstart
        if (touchWithinRange(this.touchStartPos, event.changedTouches[0], 20)) { //Only treat as a click if le
        return this.handleClick(event.changedTouches[0].offsetX, event.changedTouches[0].offsetY);
      }
    }
    break;
  }
  return this.superclass.handleEvent.call(this, event); //Let the superclass handle the event, if not handled l
};
```

This method is checking the **type** of the event, and behaving differently based on this.

In the simplest case, the click case just calls a control method named **handleClick()** (implemented below), passing the x & y position of the click.

The touch events are slightly more involved. They include logic to check whether a tap has occurred, rather than a long press, as well as making sure that it is not a swipe. As part of this, the logic calls a global method named **touchWithinRange()** to check that the touch position has not moved more than 20px in any direction. Add this method to your JavaScript file:

```
touchWithinRange = function (touchStartPos, touchObj, range) {
  if (Math.abs(touchStartPos.x - touchObj.clientX) > range)
    return false;
  if (Math.abs(touchStartPos.y - touchObj.clientY) > range)
    return false;
  return true;
};
```

The final stage is to implement the **handleClick()** method.

```
ctrl.handleClick = function(pX, pY) {
  if (this.canSendEvent(eBaseEvent.evClick)) {
    this.eventParamsAdd("pXPos", pX);
    this.eventParamsAdd("pYPos",pY);
    this.sendEvent(eBaseEvent.evClick);
  }
};
```

This first checks whether the click event is enabled. If so, it adds the x and y c-ordinates passed in to the event parameters, then sends the event to Omnis.

This particular control should also do some extra work here, to implement functionality determined by the **$togglecolor** property.

Extend the **handleClick()** method as shown:

```
ctrl.handleClick = function(pX, pY) {
  if (this.mToggleColor) {
    var backAlpha = this.getProperty(eBaseProperties.backalpha);
    var newAlpha = backAlpha > 0 ? 0 : 255; // If backAlpha > 0, newAlpha=0, else newAlpha=255
    this.setProperty(eBaseProperties.backalpha, newAlpha);
  }
  if (this.canSendEvent(eBaseEvent.evClick)) {
    ..
  }
};
```

Here the code to toggle the color is added **before** the check for whether the event is enabled, so that this occurs regardless of whether or not the event is enabled.

If the **$togglecolor** property is enabled, the control's **$backalpha** is toggled between 0 and 255.

Using a $dataname – C++ Part

This section describes the process of adding a **$dataname** property to your control.

This will be taken as a boolean, which will determine whether the text of the control should display strikethrough.

In order to give your component a **$dataname** property, you need to add an entry to your component's ECOproperty array.

The only necessary member of the structure is the **mFlags**, which must be set to **EXTD_FLAG_PRIMEDATA**. Only one property may have this flag set - this indicates that it is the special **$dataname** property.

```
ECOproperty jsGenericProperties[] =
{
  ..
  cToggleColor, cToggleColor, fftBoolean, EXTD_FLAG_PROPAPP, 0, 0, 0,
  0, 0, 0, EXTD_FLAG_PRIMEDATA, 0, 0, 0 // $dataname
}
```

In order for the **$dataname** to be sent to the client when constructing the control, you must set **mControlHasDataName**, in your control's JSChtmlOptions structure, to **qtrue**.

This is defined in **jsgeneric.h**:

```
struct JSChtmlOptionsGeneric : public JSChtmlOptions
{
  public:
    JSChtmlOptionsGeneric() : JSChtmlOptions() { mControlHasDataName = qtrue; }
};
```

Using a $dataname – JavaScript Part

The value of $dataname of a control can be obtained at any point by calling your control's getValue() method.

At the end of your control's updateCtrl() implementation, add the following code:

```
ctrl.updateCtrl = function() {
..
var dataname = this.getData();
if (this.mData != dataname) { // Only execute the following code if the value of the $dataname variable has ch
  this.mData = dataname;
  if (dataname)
    elem.style.textDecoration = "line-through";
  else
    elem.style.textDecoration = "none";
}
..
```

This sets the standard CSS "**text-decoration**" style on the client element, to enable/disable a strikethrough effect, based on the value of the $dataname.

It also uses a member variable (**mData**) to store the value of the $dataname variable. This allows the code to check whether or not the $dataname value has changed, as this code only ever needs to run if the $dataname changes. **updateCtrl()** may be called for many reasons, not just the $dataname changing.

While this would yield negligible performance improvements in this case, when your control gets more complex this can make a large difference to the performance.

Debugging

Debugging a Component

This section describes the process of debugging the C++ part of your JavaScript Component.

Windows

Make sure you set your build configuration to the **UNICODE DebugWebDesign** target.

Open your project's **Properties**.

- Under **General**, set the **Output Directory** to the **jscomp** folder in your Omnis installation.
- Under **Debugging**, set the **Command** to point to your omnis.exe (including the full path).

Start debugging (**F5**). Omnis will be launched with your new component, and any breakpoints in your code will be hit.

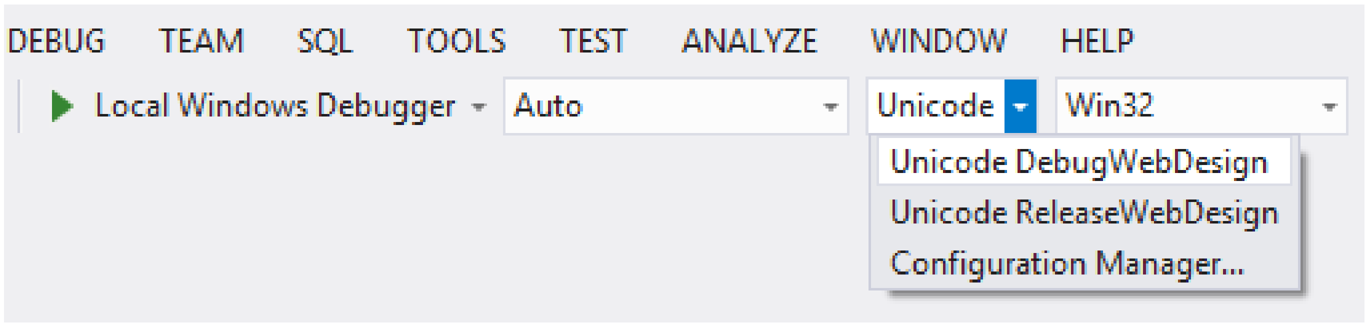
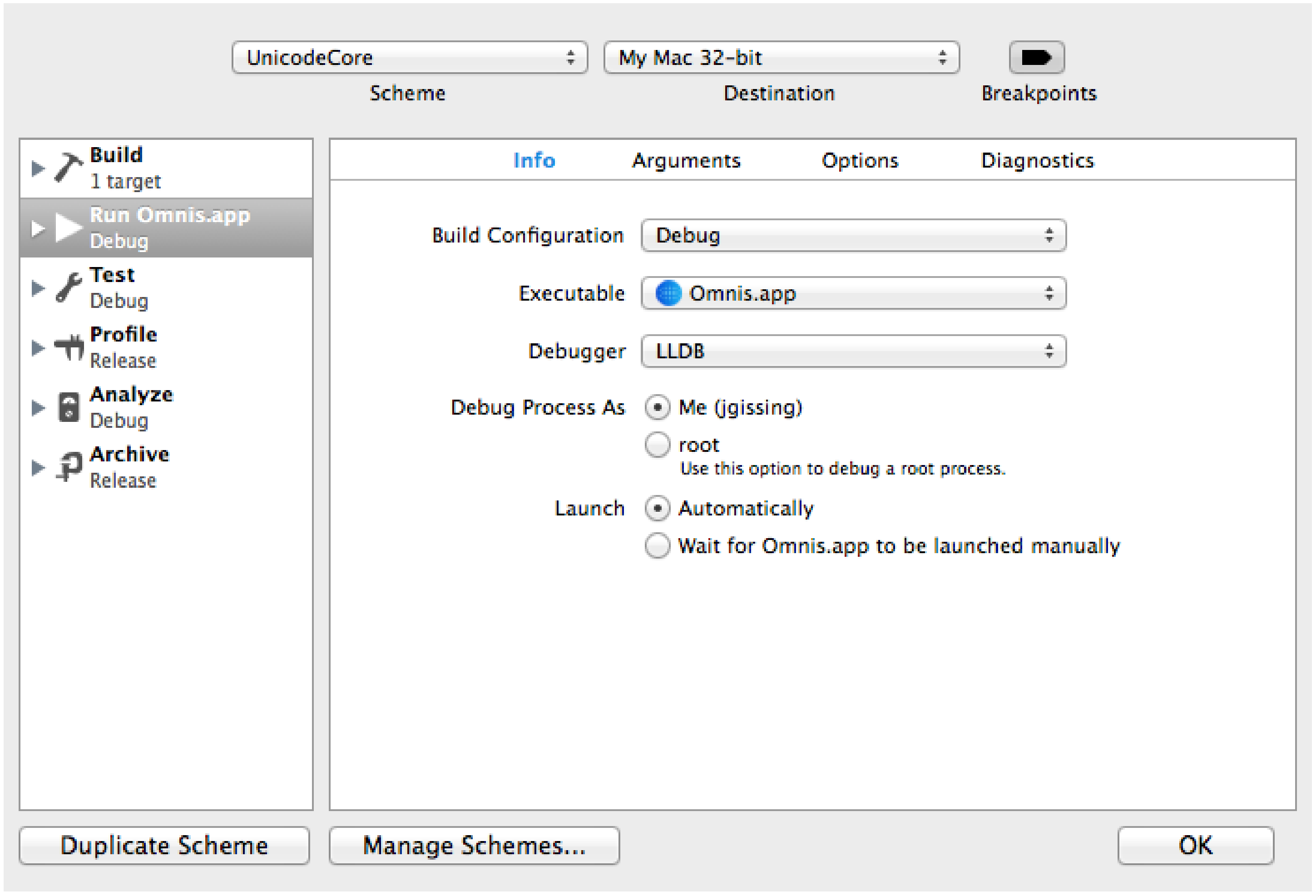


Figure 1:

Mac

The first step in debugging an OS X component, is to edit the **scheme**.

- Open the **Product** menu, and select **Scheme** > Edit Scheme.
- Select the **Run** configuration, and open its Info tab.
- Set its **Executable** to your Omnis app.



Build Configuration: Debug

Executable: Omnis.app

Debugger: LLDB

Debug Process As: [✓ Me (jgissing)] root
Use this option to debug a root process.

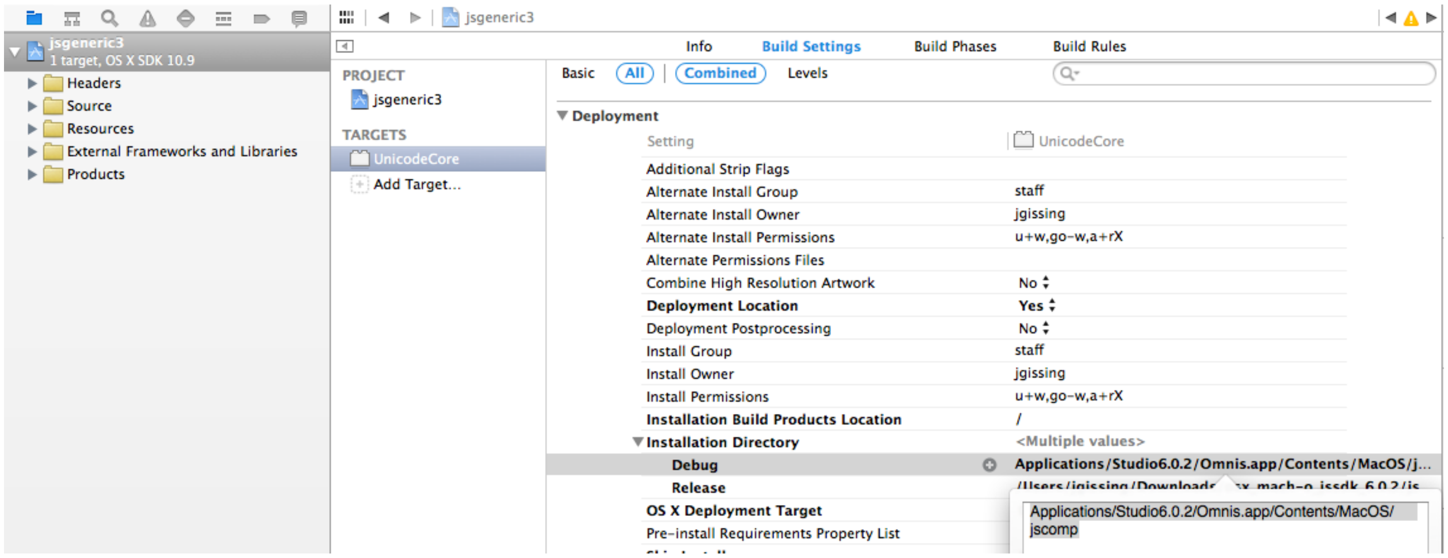
Launch: [✓ Automatically] Wait for Omnis.app to be launched manually

Figure 2:

The next step is to make sure that your component is built into the Omnis tree.

- Open the **Build Settings** in your project.

- Locate the Installation **Directory** setting.
- Set the **Debug** value to the full path to the **jscomp** folder in your Omnis app.



Setting	UnicodeCore
Additional Strip Flags	
Alternate Install Group	staff
Alternate Install Owner	jgissing
Alternate Install Permissions	u+w,go-w,a+rX
Alternate Permissions Files	
Combine High Resolution Artwork	No
Deployment Location	**Yes**
Deployment Postprocessing	No
Install Group	staff
Install Owner	jgissing
Install Permissions	u+w,go-w,a+rX
Installation Build Products Location	/
Installation Directory	<Multiple values>
Debug	**Applications/Studio6.0.2/Omnis.app/Contents/MacOS/j...**
Release	**/Users/jgissing/Downloads ... x_mach-o_jssdk_6.0.2/js**
OS X Deployment Target	
Pre-install Requirements Property List	

Applications/Studio6.0.2/Omnis.app/Contents/MacOS/jscomp

Figure 3:

You can now **Run** your project. Omnis will be launched, and any breakpoints in your code will be hit.

Debugging a Control

This section gives a simple overview of debugging a JavaScript **Control**, the JavaScript part of your component.

In order to debug JavaScript code, you need to make use of the debugging tools in your browser. Most of the major browsers have their own implementation for this. This document is based on using **Chrome** (since its debugging tools are very good), but the principles hold true for all browsers.

Viewing the DOM Structure

The first, and simplest, debugging procedure is to take a look at the structure of your control. This allows you to make sure that the elements are being constructed as you wish.

- Load your Omnis form in Chrome, right-click an element on your form, and select **Inspect Element**. This will open the Developer Tools, and take you to the clicked element within the DOM.
- Here you can inspect the HTML, to make sure that it is being constructed properly.
- You can also make changes here on-the-fly, and see their effects immediately. This is very helpful in allowing you to work out exactly what attributes/styles, etc, you need to set in order to get your element to appear just as you want.

Stepping Through Code

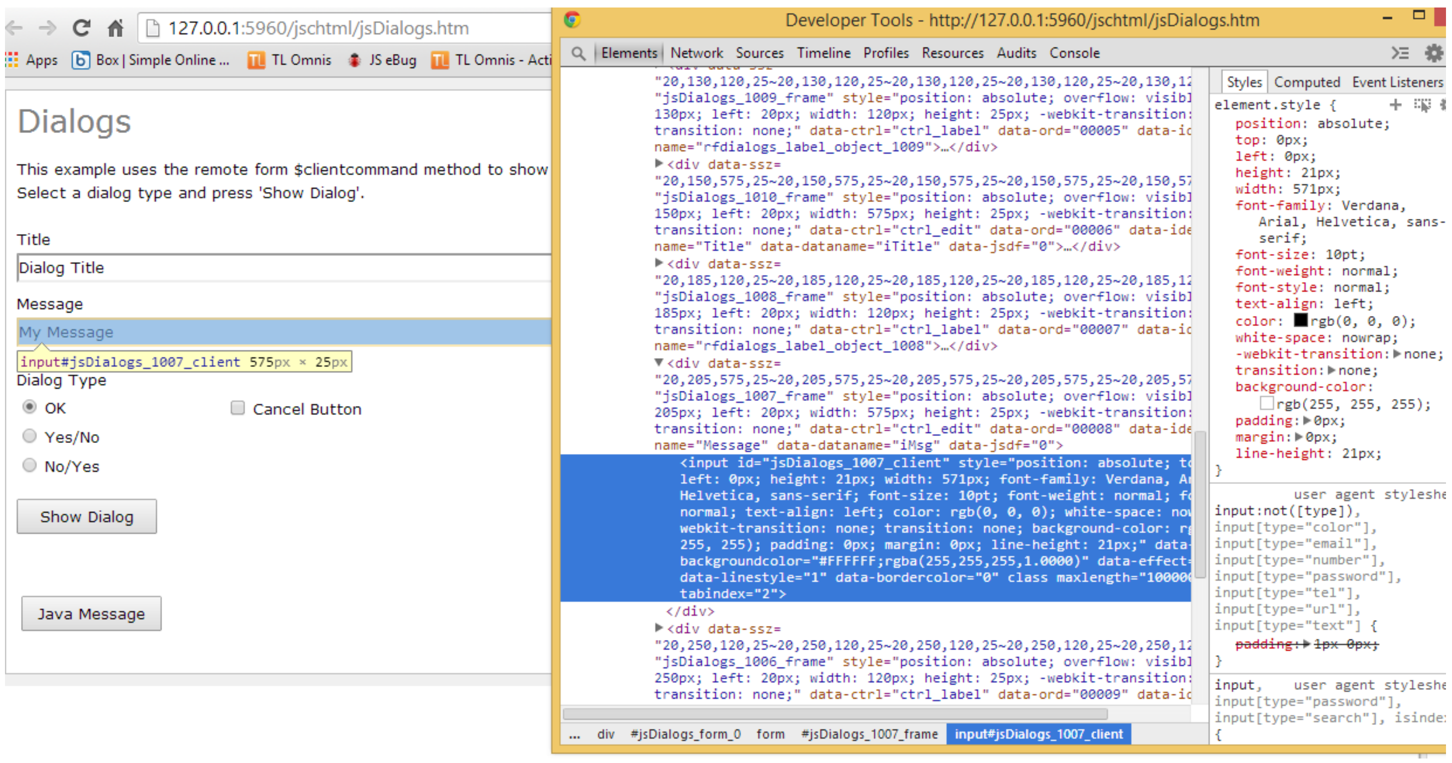
Another very useful debugging feature is the ability to step through your code.

Load your Omnis form in Chrome, then open the developer tools (**F12 / Cmd-Alt-I**). Alternatively you can right-click some element on the form and **Inspect element**.

Switch to the **Sources** tab, and select your .js file from the scripts folder shown in the source tree.

This will bring up your source code, and allow you to set breakpoints (by clicking in the margin).

When you hit a breakpoint, you can view the values of variables by simply hovering over them with the mouse, or in the developer tools sidebar.



127.0.0.1:5960/jschtml/jsDialogs.htm

Developer Tools - http://127.0.0.1:5960/jschtml/jsDialogs.htm

Dialogs

This example uses the remote form $clientcommand method to show

Select a dialog type and press 'Show Dialog'.

Title: Dialog Title

Message: My Message

input#jsDialogs_1007_client 575px × 25px

Dialog Type: [✓ OK] Yes/No No/Yes; Cancel Button

```
"20,130,120,25~20,130,120,25~20,130,120,25~20,130,120,25~20,130,12
"jsDialogs_1009_frame" style="position: absolute; overflow: visibl
130px; left: 20px; width: 120px; height: 25px; -webkit-transition:
transition: none;" data-ctrl="ctrl_label" data-ord="00005" data-ic
name="rfdialogs_label_object_1009">…</div>
▶<div data-ssz=
"20,150,575,25~20,150,575,25~20,150,575,25~20,150,575,25~20,150,57
"jsDialogs_1010_frame" style="position: absolute; overflow: visibl
150px; left: 20px; width: 575px; height: 25px; -webkit-transition:
transition: none;" data-ctrl="ctrl_edit" data-ord="00006" data-ide
name="Title" data-dataname="iTitle" data-jsdf="0">…</div>
▶<div data-ssz=
"20,185,120,25~20,185,120,25~20,185,120,25~20,185,120,25~20,185,12
"jsDialogs_1008_frame" style="position: absolute; overflow: visibl
185px; left: 20px; width: 120px; height: 25px; -webkit-transition:
transition: none;" data-ctrl="ctrl_label" data-ord="00007" data-ic
name="rfdialogs_label_object_1008">…</div>
▼<div data-ssz=
"20,205,575,25~20,205,575,25~20,205,575,25~20,205,575,25~20,205,57
"jsDialogs_1007_frame" style="position: absolute; overflow: visibl
205px; left: 20px; width: 575px; height: 25px; -webkit-transition:
transition: none;" data-ctrl="ctrl_edit" data-ord="00008" data-ide
name="Message" data-dataname="iMsg" data-jsdf="0">
   <input id="jsDialogs_1007_client" style="position: absolute; to
   left: 0px; height: 21px; width: 571px; font-family: Verdana, Ar
   Helvetica, sans-serif; font-size: 10pt; font-weight: normal; fo
   normal; text-align: left; color: rgb(0, 0, 0); white-space: now
   webkit-transition: none; transition: none; background-color: rg
   255, 255); padding: 0px; margin: 0px; line-height: 21px;" data-
   backgroundcolor="#FFFFFF;rgba(255,255,255,1.0000)" data-effect=
   data-linestyle="1" data-bordercolor="0" class maxlength="100000
   tabindex="2">
 </div>
▶<div data-ssz=
"20,250,120,25~20,250,120,25~20,250,120,25~20,250,120,25~20,250,12
"jsDialogs_1006_frame" style="position: absolute; overflow: visibl
250px; left: 20px; width: 120px; height: 25px; -webkit-transition:
transition: none;" data-ctrl="ctrl_label" data-ord="00009" data-ic
```

```
element.style {
  position: absolute;
  top: 0px;
  left: 0px;
  height: 21px;
  width: 571px;
  font-family: Verdana,
     Arial, Helvetica, sans-
     serif;
  font-size: 10pt;
  font-weight: normal;
  font-style: normal;
  text-align: left;
  color: ■rgb(0, 0, 0);
  white-space: nowrap;
  -webkit-transition:▶none;
  transition:▶none;
  background-color:
     □rgb(255, 255, 255);
  padding:▶0px;
  margin:▶0px;
  line-height: 21px;
}
           user agent styleshe
input:not([type]),
input[type="color"],
input[type="email"],
input[type="number"],
input[type="password"],
input[type="tel"],
input[type="url"],
input[type="text"] {
  ~~padding:▶ 1px 0px;~~
}
input,     user agent styleshe
input[type="password"],
input[type="search"], isinde
{
```

Figure 4:

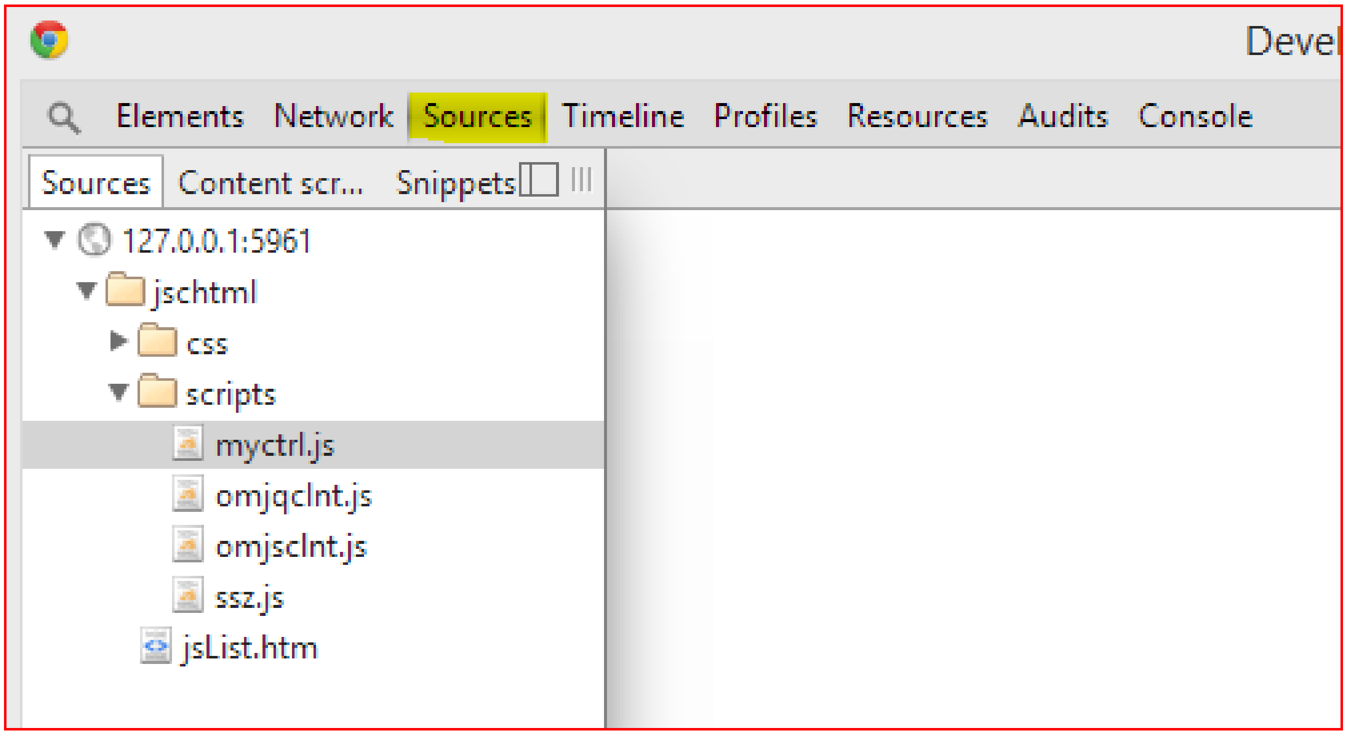
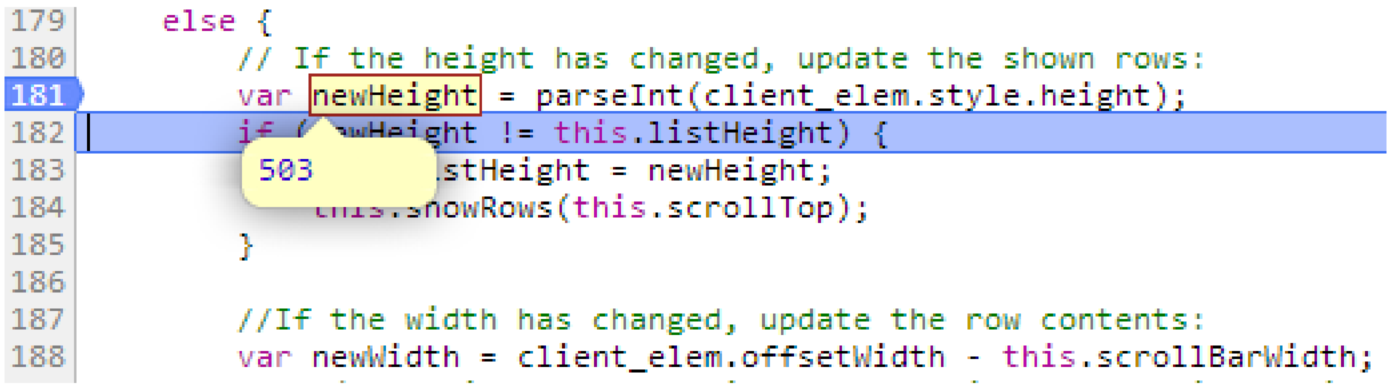


Figure 5:



```
    else {
        // If the height has changed, update the shown rows:
        var newHeight = parseInt(client_elem.style.height);
        if (  wHeight != this.listHeight) {
                 stHeight = newHeight;
            this.showRows(this.scrollTop);
        }

        //If the width has changed, update the row contents:
        var newWidth = client_elem.offsetWidth - this.scrollBarWidth;
```

Figure 6:

C++ Design Component Reference

The following are declared as part of the external components interface and are detailed in the Omnis Studio External Components documentation.

External Component Structures

ECOmethodEvent

The ECOmethodEvent structure defines information about a single component event or method. The WCCcontrol structure takes an array of this structure in its mEvent member to define a control's events and an array of this structure in its mMethods member to define a control's methods.

```
struct ECOmethodEvent
{
  qlong mId;
  qlong mNameResID;
  qlong mReturnDataType;
  qlong mParameterCount;
  ECOparam* mParameters;
  qlong mFlags; qlong mExFlags;
};
```

- **mId** - The unique identifier, within the event table, for the event. All external events must have a positive number and must not be zero. All negative numbers are assumed to be Omnis internal events. For a list of supported internal events, look for ECE_ in EXTDEFS.HE.
- **mNameResID** - Resource id which contains the event name. Event names must be unique and must not clash with Omnis internal events. The string 'ev' is used automatically as a prefix for any event. Use zero for internal events.
- **mReturnDataType** - Returned data type of type fftxxx. Specify 0 for no returned data (e.g. void) and fftNone for an unspecified data type.

- **mParameterCount** - Number of parameters for the event. Specify zero for no parameters.
- **mParameters** - Pointer to an array of ECOparam parameters. Specify NULL if there are no parameters.
- **mFlags** - Event flags of type EXTD_FLAG_xxxx. Use zero if you don't need any of the flags.
- **mExFlags** - Use zero. Extended flags for future enhancement.

ECOparam

The ECOparam structure defines information about a single event parameter.

```
struct ECOparam
{
  qlong mNameResID;
  qlong mDataType;
  qlong mFlags;
  qlong mExFlags;
};
```

- **mNameResID** - Resource id for the parameter name. Defined in your .rc file, with the name and description of the parameter (separated by a colon - e.g. '7000 "MyParam:A description of MyParam" ') . Standard parameters should use one of the ECE_xxx constants.
- **mDataType** - Data Type of the property. An ffttype value, e.g. fftInteger.
- **mFlags** - Parameter flags of type EXTD_FLAG_xxxx. Examples are EXTD_FLAG_PARAMOPT and EXTD_FLAG_PARAMALTER. Events from the JS Client will generally just use 0, as these flags usually don't apply.
- **mExFlags** - Must be zero. Extended flags for future enhancement.

ECOproperty

The ECOproperty structure defines information about a single component property. The WCCcontrol structure takes an array of this structure in its mProperties member to define the properties of a control.

```
struct ECOproperty
{
  qlong mPropID;
  qlong mNameResID;
  qlong mDataType;
  qlong mFlags;
  qlong mExFlags;
  qlong mEnumStart;
  qlong mEnumEnd;
};
```

- **mPropID** - Property Identifier. To use standard properties, use the appropriate anumxxx value, e.g. anumText. Custom properties conventionally use values from 4000.
- **mNameResID** - Resource id for the property name. Defined in your .rc file, with the name and description of the property (separated with a colon - e.g: '4000 "$myproperty:Description of my property" '. Necessary for custom properties but standard properties should use 0. Custom properties conventionally use the same value as mPropID.
- **mDataType** - Data Type of the property. An ffttype value, e.g. fftInteger.
- **mFlags** - An EXTD_FLAG_xxx value. Multiple flags can be set using the bitwise OR operator, e.g. EXTD_FLAG_PROPTEXT | EXTD_FLAG_PWINDCOL.
- **mExFlags** - Extended flags for future enhancement.
- **mEnumStart** - The first constant id in this property's constant enumeration.

- **mEnumEnd** - The last constant id in this property's constant enumeration.

The size of this array must be a multiple of seven.

Adding a $dataname property

If you wish to include a $dataname property, this is handled as a special case.

It requires an ECOproperty with all members set to 0, except mFlags, which should be set to **EXTD_FLAG_PRIMEDATA**.

You will also need to set **mControlHasDataName** to qtrue in the constructor of your control's **JSChtmlOptions** struct (in your control's header file).

JavaScript Component Structures

JSChtmlOptions

The JSChtmlOptions structure specifies the inner HTML generation options for a JavaScript client control. A component object passes an instance of this structure to its superclass via the mHtmlOptions member of the WCCcontrol structure.

```
struct JSChtmlOptions
{
  public:
    JSChtmlOptions()
    {
      mInnerStylePositionIsRelative = qfalse;
      mInnerStyleHasFont = qtrue;
      mInnerStyleHasTextColor = qtrue;
      mInnerStyleHasBackgroundColor = qtrue;
      mInnerStylePadding = 0;
      mInnerStyleHasScrollbars = qfalse;
      mInnerStyleHasAlign = qtrue;
      mInnerStyleHasNoWrap = qtrue;
      mControlHasEvents = qtrue;
      mControlHasDataName = qtrue;
      mAdjustWidthHeightForBorder = qfalse;
    }
  qbool mInnerStylePositionIsRelative;
  qbool mInnerStyleHasFont;
  qbool mInnerStyleHasTextColor;
  qbool mInnerStyleHasBackgroundColor;
  qbyte mInnerStylePadding;
  qbool mInnerStyleHasScrollbars;
  qbool mInnerStyleHasAlign;
  qbool mInnerStyleHasNoWrap;
  qbool mControlHasEvents;
  qbool mControlHasDataName;
  qbool mAdjustWidthHeightForBorder;
  str255 mInnerStyleExtra;
};
```

- **mInnerStylePositionIsRelative** - A boolean which if set to qtrue will set the CSS position property to a value of 'relative' within the HTML inline style attribute. This positions the inner HTML relative to its normal position. The default for this member is false which will set the position property to a value of 'absolute' to position the HTML relative to its parent.
- **mInnerStyleHasFont** - A boolean which if set to qtrue will include the CSS font properties within the HTML inline style attribute, e.g. font-family, font-size, etc. This allows text within the HTML to be styled based on the Omnis instance's text properties, e.g. $font, $fontsize, etc. The default is qtrue.

- **mInnerStyleHasTextColor** - A boolean which if set to qtrue will include the CSS color property within the HTML inline style attribute. This allows the color of text within the HTML to be based on the Omnis instance's $textcolor property. The default is qtrue.
- **mInnerStyleHasBackgroundColor** - A boolean which if set to qtrue will include the CSS background-color property within the HTML inline style attribute. This allows the background within the HTML to be based on the Omnis instance's $backcolor property. The default is qtrue.
- **mInnerStylePadding** - A byte value of between 0 and 255 which if greater than zero will set the CSS padding property within the HTML inline style attribute. This defines the space between the HTML element's border and its content. The default value is zero.
- **mInnerStyleHasScrollbars** - A boolean which if set to qtrue will include the CSS overflow-x and overflow-y properties within the HTML inline style attribute. This controls what happens to the HTML element's content if it overflows the content area. If the Omnis instance has a $autoscroll property set to kTrue the CSS properties will be set to 'auto', i.e. the content will have scrollbars in both directions. If $autoscroll is kFalse, the scroll behavior is based on the values of the $horzscroll and $vertscroll properties. If set to kTrue the CSS property will be set to 'scroll' and a scrollbar will appear in the required direction, if kFalse the property will be set to 'hidden' and there will be no scrollbar. By default the value of mInnerStyleHasScrollbars is false.
- **mInnerStyleHasAlign** - A boolean which if set to qtrue will include the CSS text-align property within the HTML inline style attribute. This sets the horizontal alignment of text in the HTML element. The value is set according to the value of the Omnis instance's $align property. When this is set to kLeftJst, kRightJst or kCenterJst the CSS value will be one of left, right, center. The default for mInnerStyleHasAlign is qtrue.
- **mInnerStyleHasNoWrap** - A boolean which if set to qtrue will include the CSS white-space property within the HTML inline style attribute. This will have a value of 'no-wrap'. This means that sequences of white space will be reduced to a single space and that text will not wrap to the next line. The default for mInnerStyleHasNoWrap is qtrue.
- **mControlHasEvents** - A boolean which if set to qtrue will include the data-event attribute in the outer frame DIV element of the HTML if the component generates events. This is set to provide information to the JavaScript control to identify which events it should support. The data-event attribute uses a comma delimited list of event codes as its value. These codes are based on the list of evXXX constants which are set in the Omnis instance's $events property. If this list is empty then the attribute is not added. The default for mControlHasEvents is qtrue.
- **mControlHasDataName** - A boolean which if set to qtrue will include the data-dataname attribute in the outer frame DIV element of the HTML if the component has a $dataname value. This is set to provide information to the JavaScript control to identify the instance variable providing data to that control. The data-dataname attribute uses the name of the instance variable as its value. The default for mControlHasDataName is qtrue.
- **mAdjustWidthHeightForBorder** - A boolean which if set to qtrue will adjust the width and height of the outer fame DIV element of the HTML by the border width. The default value is false.
- **mInnerStyleExtra** - An Omnis str255 string containing any extra CSS styling information to be appended to the HTML inline style attribute.

WCCcontrol

The WCCcontrol structure defines information about a JavaScript component. An instance of this structure is stored in the component's base class when a component is created. It returns information requested by messages sent from Omnis and it is also passed to a component's JSC_GETINNERHTML function for use by utility methods that generate HTML based on the options which this structure defines.

```
struct WCCcontrol
{
  qlong mResourceBase; // NOT USED
  qlong mLibResName; // Resource number of library
  qlong mObjectID; // External component object id
  qlong mIconID; // ICON id for control
  qlong mEventCount; // Count of events
  ECOmethodEvent *mEvents; // Control events
  qlong mPropertyCount; // Count of properties
  ECOproperty *mProperties; // Control properties
  qlong mFirstConstant; // First constant resource; zero if no constants
  qlong mLastConstant; // Last constant resource; zero if no constants
  qlong mCompFlags; // EXTIPJS_FLAG_xxx or EXTJS_FLAG_xxx flags sent in response to ECM_IPHONE_OR_JAVASCRIPT_C
  qlong mWccFlags; // WCC_FLAG_... flags
  qlong mCompStoreGroupRes; // Resource number of component store group
  qlong mMethodCount; // Count of methods
  ECOmethodEvent *mMethods; // Control methods
  FMTclassMakeFunc mMakeFmtClass; // Function to make class notation object
  qlong mCustomTabName; // Resource number of custom tab
  qdim mFixedLandscapeWidth; // Fixed landscape width for control (zero if any width allowed)
  qdim mFixedLandscapeHeight;// Fixed landscape height for control (zero if any height allowed)
  qdim mFixedPortraitWidth; // Fixed portrait width for control (zero if any width allowed)
  qdim mFixedPortraitHeight; // Fixed portrait height for control (zero if any height allowed)
  JSChtmlOptions *mHtmlOptions; // JavaScript client HTML options
  JSC_GETINNERHTML mGetInnerHtml; // JavaScript client function to get inner HTML
  JSC_APPENDCONTROLSTYLES mAppendControlStyles; // JavaScript client function to append control-specific style
  const char * mJscCtrlName; // JavaScript client control name
  attnumber mControlNameAnum; // Attribute for control name when mJscCtrlName is to be overridden
};
```

- **mResourceBase** - Not used for JavaScript components.
- **mLibResName** - Resource id which contains the component library name.
- **mObjectID** - Resource id which contains the name of the component object within the library.
- **mIconID** - Resource id for the component object's icon.
- **mEventCount** - The number of component events defined in the component.**mEvents** - A pointer to an array of component events. Specify NULL if there are no events.
- **mPropertyCount** - The number of component properties that the component defines.
- **mProperties** - A pointer to an array of component properties. Specify NULL if there are no properties.
- **mFirstConstant** - Resource id of the first property constant. Specifiy zero if there are no constants.
- **mLastConstant** - Resource id of the last property constant. Specifiy zero if there are no constants.
- **mCompFlags** - Flags of type EXTIPJS_FLAG_xxx or EXTJS_FLAG_xxx which indicate supported properties and behaviors of the component.
- **mWccFlags** - Flags of type WCC_FLAG_xxx which indicate supported behaviors of the component.
- **mCompStoreGroupRes** - Resource id which contains the name of the Component Store group to which this component belongs.
- **mMethodCount** - The number of component methods that the component defines.
- **mMethods** - A pointer to an array of component methods. Specify NULL if there are no methods.
- **mMakeFmtClass** - A FMTclassMakeFunc function that creates a sub-class of webClientComponent_formatNotation used to handle format notation when it needs to be processed by the control. This is zero if not required.(example). Not used for JS?
- **mCustomTabName** - Resource id which contains the name of the custom tab that this component is using. (example)
- **mFixedLandscapeWidth** - When a device is oriented in landscape mode then if greater than zero the value will fix the width of the control, otherwise any width is allowed. (example)
- **mFixedLandscapeHeight** - When a device is oriented in landscape mode then if greater than zero the value will fix the height of the control, otherwise any width is allowed. (example)
- **mFixedPortraitWidth** - When a device is oriented in portrait mode then if greater than zero the value will fix the width of the control, otherwise any width is allowed. (example)

- **mFixedPortraitHeight** - When a device is oriented in portrait mode then if greater than zero the value will fix the height of the control, otherwise any width is allowed. (example)
- **mHtmlOptions** - A pointer to the JSChtmlOptions structure which defines the options which affect the creation of the inner HTML template generated by the component.
- **mGetInnerHtml** - A JSC_GETINNERHTML function that generates the inner HTML template for the component.
- **mAppendControlStyles** - A JSC_APPENDCONTROLSTYLES function which appends control-specific styles to the inner HTML template. This is zero if there are no control-specific styles.
- **mJscCtrlName** - A C string which is the name of the JavaScript object which implement the control.
- **mControlNameAnum** - If overriding an existing control this is the attribute which specifies the name of the overriding JavaScript class. (example)

Flags

EXTIPJS_FLAG_xxx

The EXTIPJS_FLAG_xxx defines are used in the mCompFlags member of the WCCcontrol structure.

EXTIPJS_FLAG_BACKCOLOR_AND_BACKALPHA

Indicates that the control has the $backcolor and $backalpha properties to change the background color.

EXTIPJS_FLAG_BEFORE_AND_AFTER

Indicates that the control generates before and after events.

EXTIPJS_FLAG_EFFECT

Indicates that the control has the $effect property to change its border style.

EXTIPJS_FLAG_NO_ERASE_BACKGROUND

Indicates that the control does not use the automated erase background processing.

EXTIPJS_FLAG_NOENABLED

Indicates that the control does not have the $enabled property.

EXTIPJS_FLAG_TRANSPARENT_BACKGROUND

Indicates that the control does not have the $backcolor and $backalpha properties and needs a transparent background. EXTIP_FLAG_BACKCOLOR_AND_BACKALPHA must not be set.

EXTJS_FLAG_xxx

The EXTJS_FLAG_xxx defines are used in the mCompFlags member of the WCCcontrol structure.

EXTJS_FLAG_HAS_DEFAULT_BORDER

Indicates that the control can have its $effect property set to kJSborderDefault.

EXTJS_FLAG_HAS_DISPLAY_FORMAT

Indicates that the control has the $jsdateformat, $jsdateformatcustom and $jsnumberformat properties.

EXTJS_FLAG_HASMENUS

Indicates that the control has remote menus which need to be added to the context menu list sent to the client.

EXTJS_FLAG_NOTIFY_FLD_ENABLE

Indicates the control will be sent messages to change its enabled state.

WCC_FLAG_xxx

The WCC_FLAG_xxx defines are used in the mWccFlags member of the WCCcontrol structure.

WCC_FLAG_CANFOCUS

Indicates that the control accepts the focus.

WCC_FLAG_USESCROLLVIEW

Indicates that the scroll view takes the focus when an OMBorderedView container is supplied.

Messages

The following are the most commonly used Omnis external component messages which are passed to a JavaScript component. For further details and a full list of messages please refer to the Omnis Studio External Components documentation.

ECM_GETPROPERTY

The ECM_GETPROPERTY message is sent to the component when Omnis requires the data for a property.

The component should add a return parameter to the associated EXTCompInfo structure. This parameter will contain the property data.

Returns : True if successful, false otherwise.

ECM_PROPERTYCANASSIGN

The ECM_PROPERTYCANASSIGN message is sent to a component when Omnis needs to know if a property can be written to or is read only.

Returns: True if the property can be written to, false otherwise.

ECM_SETPROPERTY

The ECM_SETPROPERTY message is sent to a component when Omnis requires a property to change.

The parameter passed in the associated EXTCompInfo structure contains the new data for the property.

Returns : True if successful, false otherwise.

Function Pointer Types

JSC_GETINNERHTML

```
typedef qbool (*JSC_GETINNERHTML)(HWND pHwnd,
WCCcontrol *pControl, webClientComponent *pObject,
EXTfldval &pInnerHtml, qdim pWidth, qdim pHeight)
```

The JSC_GETINNERHTML function pointer type defines the function used to generate the inner HTML for a JavaScript control. An implementation of a function of this type will return an HTML string representation of the control in the pInnerHtml parameter. This string is constructed based on the values of all other parameters passed into the function. A function of this type is passed as the mGetInnerHtml member of a component's WCCcontrol structure. The function will be called when Omnis sends a message requesting the HTML for a JavaScript control.

Parameters:

- pHwnd - The child window associated with a component. This is used to obtain values of the Omnis standard properties for a component.
- pControl - This is a WCCcontrol structure used to provide information about the required properties and behaviors of the JavaScript control.
- pObject - A pointer to the instance of a component object. This will be NULL if there is no open design window.
- pInnerHtml - A reference to a destination data field that will contain the HTML string for a control.
- pWidth - The required CSS width of the inner HTML. This may be calculated based on border and padding options.
- pHeight - The required CSS height of the inner HTML. This may be calculated based on border and padding options.

Returns: true if successful, false otherwise.

Example:

A typical use of this method is shown in the implementation of jsGenericComponent::jsGetInnerHTML in the jsgeneric.cpp source example.

webClientComponent Reference

The webClientComponent class is the top level base component support class and provides the default message handling, functionality and property support. It is inherited through the derived javaScriptComponent class which is itself directly inherited by the bottom level derived class implementation of a JavaScript component object, e.g. jsGenericComponent. This component specific sub-class is required to override and implement a set of virtual class methods declared in the webClientComponent base which are then called to provide the standard JavaScript component behavior.

Public Members

These are class members that can be accessed directly to alter a component's behavior.

mNoDesignName

```
qbool mNoDesignName;
```

- **mNoDesignName** - A boolean which if set to qtrue will prevent the object from drawing its name on the control in the design window. The default is qfalse.

Virtual Methods

An instance of a JavaScript component is required to implement some of the following virtual methods to render a particular type of component and to provide the correct functionality.

attributeSupport()

```
virtual qlong attributeSupport(LPARAM pMessage,
WPARAM wParam, LPARAM lParam, EXTCompInfo* eci)
```

Implemented by a subclass to provide responses to property management requests from the base class.

Parameters:

- **pMessage** (LPARAM): The message identifier for the request, e.g. ECM_SETPROPERTY.
- **wParam** (WPARAM): Additional message information for the request. Typically this is not used for a JavaScript component.
- **lParam** (LPARAM): Additional message information for the request. Typically this is not used for a JavaScript component.
- **eci** (EXTCompInfo): This is a pointer to a structure used for providing information about the component.

Returns:

- (qlong): The success or failure of the request indicated by a 1 or 0.

inval()

```
virtual void inval()
```

Invalidate the whole of a component's window client area. This will result in a request to repaint the window and a call to paintDesign. Typically a subclass does not need to override this method. It is mainly used when a property changes the content of the window and it is sufficient to call the super class implementation.

paintDesign()

```
virtual void paintDesign(HDC pHdc);
```

Provides drawing commands to render a component in a design window.

Parameters:

- **pHdc** (HDC): Identifies the device in which the drawing will occur.

javaScriptComponent Reference

The javaScriptComponent class is the support class which must be the base class for all JavaScript component object classes. It provides utility methods for generating the inner HTML which are used to initially setup a JavaScript control in the client browser. It also inherits the webClientComponent base class which provides the default message handling, functionality and property support.

Class Methods

The following are static class methods that are generally used to build the HTML for a JavaScript control.

javaScriptComponent::escapeHTMLsyntaxChars()

javaScriptComponent::escapeHTMLsyntaxChars(EXTfldval &pFval, EXTfldval &pEscapedFval)

Converts the string in the pFval data field to a version with escaped HTML markup characters (<, >, &, etc.) and places the result in pEscapedFval. This will allow the escaped string to be interpreted as a character literal. This can be called as part of the HTML generation of the inner HTML for a JavaScript control in a component's JSC_GETINNERHTML function.

Parameters:

- **pFval** (EXTfldval&): The source data field containing the string to escape.
- **pEscapedFval** (EXTfldval&): The destination data field containing the resulting escaped string.

Example

```
//
// webClientComponent *object;
// EXTfldval &pInner;
//
str255 innerTemplate = str255(QTEXT("<div$></div>"));
pInner.setChar(innerTemplate);
EXTfldval textAttr;
textAttr(str15(QTEXT(" data-text=\"")));
EXTfldval textValue;
escapeHTMLsyntaxChars(object->mText, textValue);
textAttr.concat(textValue);
textAttr.concat('"');
jsInsert(textAttr, pInner, qfalse);
```

javaScriptComponent::jsAddIconUrl()

void javaScriptComponent::jsAddIconUrl(HWND pHwnd, qlong pIconID, EXTfldval &pFval, qbool pInsert = qtrue)

Inserts the URL of an icon image into the first '$' placeholder marker in pFval. The icon is identified by the pIconID parameter. The value of the path will be the location of the image relative to the directory containing the HTML page. If there are multiple image sizes for a single image id, each additional image file is appended to the path and separated by a semicolon character. The path is typically added to the inner HTML generated by a component's JSC_GETINNERHTML function to set the value of the HTML data-icon attribute for a JavaScript control. This attribute is used to display the icon image on the JavaScript control.

Parameters:

- **pHwnd** (HWND): The child window associated with this component.
- **pIconID** (qlong): The id of the icon image.
- **pFVal** (EXTfldval&): The destination data field which is usually an HTML string where the URL string of the icon is inserted into the first placeholder.
- **pInsert** (qbool): Optional. The default is qtrue to insert the URL as a replacement for the first placeholder marker. If qfalse then the URL is concatenated onto the end of the existing pFval string.

Example

```
//
// HWND pHwnd;
// webClientComponent *object;
// EXTfldval &pInner;
//
str255 innerTemplate = str255(QTEXT("<div data-icon=\"$\"></div>"));
pInner.setChar(innerTemplate);
jsAddIconUrl(pHwnd, object->mIconId, pInner); // e.g. <div data-icon="icons/datafile/omnispic/001655n16.png"><
```

javaScriptComponent::jsGetFontName()

void javaScriptComponent::jsGetFontName(HWND pHwnd, EXTfldval &pFontNameFval, qlong pFont = -1, qlong pFontsize = -1)

Gets the Omnis font name for a component associated with the pHwnd child window and uses it to set the value of the pFontNameFval data field. This value by default is based on the component's $font and $fontsize properties. When not -1 the pFont and pFontsize parameters are used to specify a particular Omnis font index and/or size that should be used. This can be called as part of the HTML generation of the inner HTML for a JavaScript control in a component's JSC_GETINNERHTML function.

Parameters:

- **pHwnd** (HWND): The child window associated with a component.
- **pFontNameFval** (EXTfldval&): The destination data field to store the font name.
- **pFont** (qlong): A font index to identify the required font. By default, this is -1 to indicate it is not used.
- **pFontsize** (qlong): A font size to identify the required font. By default, this is -1 to indicate it is not used.

Example

```
// HWND pHwnd;
// EXTfldval &pInner;
// %f - font family marker to be replaced with CSS property values
str255 innerTemplate = str255(QTEXT("<div style=\"%f\"></div>"));
pInner.setChar(innerTemplate);
EXTfldval fval;
EXTfldval fvalResult;
jsGetFontName(pHwnd, fval); // font name based on $font and $fontsize
jsStyleAppendFontName(fval, fvalResult);
fval.setChar(str15(QTEXT("%f")));
pInner.replaceStr(fval, fvalResult, qtrue); //<div style="font-family:'Courier New', Monospace;"></div>
```

javaScriptComponent::jsGetIconPath()

void jsGetIconPath(HWND pHwnd, qlong pIconID, str255& pStr)

Sets the path to an icon image in the pStr string parameter. The icon is identified by the pIconID parameter. The value of the path will be the location of the image relative to the directory containing the HTML page and will be contained in double quotes. If there are multiple image sizes for a single image id, each additional image file is appended to the path and separated by a semicolon character. The path is typically added to the inner HTML generated by a component's JSC_GETINNERHTML function to set the value of the HTML data-icon attribute for a JavaScript control. This attribute is used to display the icon image on the control.

Parameters:

- **pHwnd** (HWND): The child window associated with a component.
- **pIconID** (qlong): The id of the icon image.
- **pStr**(str255&): The string which will store the quoted icon path. This will be an empty set of quotes if the icon is invalid.

Example

```
//
// HWND pHwnd;
// webClientComponent *object;
// EXTfldval &pInner;
//
str255 strIcon;
str255 innerTemplate = str255(QTEXT("<div data-icon=$></div>"));
pInner.setChar(innerTemplate);
jsGetIconPath(pHwnd, object->mIconId, strIcon);
jsInsert(strIcon, pInner, qfalse); // e.g. <div data-icon="icons/datafile/omnispic/001655n16.png"></div>
```

javaScriptComponent::jsInsert()

void javaScriptComponent::jsInsert(EXTfldval& pInsertStr, EXTfldval& pDst, qbool pEscapeHTMLsyntaxChars = qtrue)

Inserts the string in pInsertStr into the first '$' placeholder marker in pDst. The string in pInsertStr is typically an HTML attribute value which substitutes for a value marker in an HTML template stored in pDst. This can be called as part of the HTML generation of the inner HTML for a JavaScript control in a component's JSC_GETINNERHTML function.

Parameters:

- **pInsertStr** (EXTfldval&): The data field which contains the attribute value string to insert.
- **pDst** (EXTfldval&): The destination data field which contains the HTML string in which to place pInsertStr.
- **pEscapeHTMLsyntaxChars** (qbool): Optional. The default is qtrue to substitute pInsertStr with escaped HTML markup characters (<, >, &, etc.). This will allow them to be interpreted as character data.

Example

```
//
// webClientComponent *object;
// EXTfldval &pInner;
//
str255 innerTemplate = str255(QTEXT("<div data-icon=\"$\"></div>"));
pInner.setChar(innerTemplate);
jsInsert(object->mFldVal, pInner); // e.g. <div data-icon="icons/datafile/omnispic/001655n16.png"></div>
```

javaScriptComponent::jsInsertBool()

javaScriptComponent::jsInsertBool(char *pAttName, qbool pValue, EXTfldval& pDst)

If pValue is true this creates a string, in the format of an attribute/value pair, from the name in pAttName and the value '1'. This string is inserted into the first '$' placeholder marker in pDst. A leading space is prepended to the string so it takes the form <space><attname>=1. If pValue is qfalse an empty string is inserted. This can be called as part of the HTML generation of the inner HTML for a JavaScript control in a component's JSC_GETINNERHTML function.

Parameters:

- **pAttName** (char*): The attribute name to insert.
- **pValue** (qbool): A boolean qtrue value indicates that the attribute is included in the HTML with a value of 1.
- **pDst** (EXTfldval&): The destination data field which contains the HTML string in which to place the attribute name/value pair.

Example

```
//
// webClientComponent *object;
// EXTfldval &pInner;
//
str255 innerTemplate = str255(QTEXT("<div$></div>"));
pInner.setChar(innerTemplate);
jsInsertNum("data-vert", object->mVertical, pInner); // e.g. <div data-vert=1></div>
```

javaScriptComponent::jsInsertBorder()

void javaScriptComponent::jsInsertBorder(HWND pHwnd, EXTfldval &pDest)

Uses the Omnis border properties for a component associated with the pHwnd child window to create the HTML data attributes which will be inserted into the first '$' placeholder marker in pDest. This is called as part of the HTML generation of the inner HTML for a JavaScript control in a component's JSC_GETINNERHTML function. The HTML data attributes are then used by the JavaScript control, e.g. data-effect, data-linestyle, data-bordercolor, data-borderradius, to calculate the correct border styling and positioning for the control.

Parameters:

- **pHwnd** (HWND): The child window associated with a component. This is used to obtain the value of the Omnis border properties for a component, e.g. $effect, $linestyle, $bordercolor, $borderradius.
- **pDest** (EXTfldval&): The destination data field that contains the HTML string for the control.

Example

```
// HWND pHwnd;
// EXTfldval &pInner;
// %e - border style marker to be replaced with data attributes
str255 innerTemplate = str255(QTEXT("<div %e></div>"));
pInner.setChar(innerTemplate);
EXTfldval fval;
EXTfldval fvalResult;
fvalResult.setChar(str15(QTEXT("$")));
jsInsertBorder(pHwnd, fvalResult);
fval.setChar(str15(QTEXT("%e")));
pInner.replaceStr(fval, fvalResult, qtrue); // e.g. "<div data-effect="10" data-linestyle="1" data-bordercolor
```

javaScriptComponent::jsInsertColor()

javaScriptComponent::jsInsertColor(const char *pAttName, qcol pColor, EXTfldval &pDest)

Replaces a '$' placeholder in *pDest* with an attribute named as specified in *pAttName*, and with a color value (RGB integer or constant) as specified by *pColor*.

Parameters:

- **pAttName** (char*): The name for the attribute to insert.
- **pColor** (qcol): The color value to use.
- **pDest** (EXTfldval&): The destination data field which contains the HTML string in which to place the attribute.

Example

```
// EXTfldval &pInner;
// $ - marker to be replaced with custom color attribute
str255 innerTemplate = str255("<div $></div>");
pInner.setChar(innerTemplate);
ECOgetProperty(pHwnd, anumBackColor, fval); // get the value of $backcolor
qcol color = fval.getLong();
jsInsertColor("data-mycolor", color, pInner);
```

javaScriptComponent::jsInsertColorPair()

javaScriptComponent::jsInsertColorPair(HWND pHwnd, EXTfldval &pDest, qcol pColorValue, qlong pAlpha)

DEPRECATED in 10.2

Color Pairs are no longer needed (all browsers now support alpha), and they do not support Themed Colors. Use jsInsertColor() instead, and pass the alpha as a separate attribute.

Uses the Omnis qcol color in pColorValue and the alpha value in pAlpha to create the equivalent CSS color property values for the component associated with the child window in pHwnd. If pAlpha is not -1 a pair of color values will be created separated by a semicolon. The first will be the CSS hex color and the second will be the CSS RGBA color. These will be inserted into the first '$' placeholder marker in pDest. This is called as part of the HTML generation of the inner HTML for a JavaScript control in a component's JSC_GETINNERHTML function. The control can the use these color alternatives based on whether it supports an RGBA value.

Parameters:

- **pHwnd** (Object): The child window associated with a component.
- **pDst** (Object): The destination data field which contains the HTML string in which to place the color pair.
- **pColorValue** (qcol): The qcol used to create a CSS hex color value or the RGB part of a CSS RGBA color value if pAlpha is not -1.
- **pAlpha** (qlong): If -1 then only a CSS hex color is created. Otherwise, a value of between 0 and 255 will be used to create the alpha part of a CSS RGBA color value.

Example

```
// HWND pHwnd;
// EXTfldval &pInner;
// $ - marker to be replaced with CSS color property value
str255 innerTemplate = str255(QTEXT("<div data-control-color=\"$\"></div>"));
pInner.setChar(innerTemplate);
ECOgetProperty(pHwnd, anumBackColor, fval); // get the value of $backcolor
qcol color = fval.getLong();
ECOgetProperty(pHwnd, anumBackgroundAlpha, fval); // get the value of $backalpha
qlong alphaValue = fval.getLong();
jsInsertColorPair(pHwnd, pInner, color, alphaValue); // e.g. "<div data-control-color="#0000FF;rgba(0,0,255,0.
```

javaScriptComponent::jsInsertIcon()

javaScriptComponent::jsInsertIcon(HWND pHwnd, char *pAttName, qlong pIconId, EXTfldval& pDst)

Creates a string, in the format of an attribute/value pair, where the attribute name is the value in pAttName and the attribute value is a URL locating the icon with the id passed in pIconId. This string is inserted into the first '$' placeholder marker in pDst. The value of the URL will be the location of the image relative to the directory containing the HTML page. If there are multiple image sizes for a single image id then each additional image file is appended to the URL and separated by a semicolon character. A leading space is prepended to the string so it takes the form <space><attname>="<icon url>". If the pIconId value does not have a valid ID an empty string is inserted into pDst. This can be called as part of the HTML generation of the inner HTML for a JavaScript control in a component's JSC_GETINNERHTML function.

Parameters:

- **pHwnd** (HWND): The child window associated with a component whose icon URL is to be used.
- **pAttName** (char*): The attribute name to insert.
- **pIconID** (qlong): The id of the icon image.
- **pDst** (EXTfldval&): The destination data field which contains the HTML string in which to place the attribute name/value pair.

Example

```
// HWND pHwnd;
// webClientComponent *object;
// EXTfldval &pInner;
//
str255 innerTemplate = str255(QTEXT("<div$></div>"));
pInner.setChar(innerTemplate);
jsInsertIcon(pHwnd, "data-icon", object->mIconId, pInner); // e.g. <div data-icon="icons/datafile/omnispic/001
```

javaScriptComponent::jsInsertId()

void javaScriptComponent::jsInsertId(HWND pHwnd, EXTfldval &pDest, qbool pFrame, strxxx *pSuffix = 0)

Inserts the HTML id attribute's name/value pair into the first '$' placeholder marker in pDest. This is called as part of the HTML generation of the inner HTML for a JavaScript control in a component's JSC_GETINNERHTML function. Typically pFrame is qfalse and pSuffix is unused. The value of id placed in the HTML will be the component group's class name, i.e. the remote form name, followed by the unique numeric $ident value of the component within the group and the suffix for the level of the element, i.e. id="<classname>_<ident>_client/frame".

Parameters:

- **pHwnd** (HWND): The child window associated with a component. This is used to obtain the value of the Omnis standard properties for the component.
- **pDest** (EXTfldval&): The destination data field that contains the HTML string for the control.
- **pFrame** (qbool): This is qfalse if the id value is to have a suffix of '_client' when generating the inner HTML of the control. This is qtrue if the suffix is to be '_frame' for the outer HTML of the control.
- **pSuffix** (strxxx*): An alternative suffix for the id value. If set then pFrame is ignored.

```
// HWND pHwnd;
// EXTfldval &pInner;
//
str255 innerTemplate = str255(QTEXT("<div $></div>"));
pInner.setChar(innerTemplate);
jsInsertId(pHwnd, pInner, qfalse); // e.g. <div id="testGeneric_1001_client"></div>
```

A typical use of this method is shown in the implementation of jsGenericComponent::jsGetInnerHTML in the jsgeneric.cpp source example.

javaScriptComponent::jsInsertIfNotEmpty()

javaScriptComponent::jsInsertIfNotEmpty(const char *pAttName, EXTfldval &pInsertStr, EXTfldval &pDst)

When the value is not empty this method inserts the HTML attribute name/value pair, formed from the attribute name in pAttName and the attribute value in pInsertStr, into the first '$' placeholder marker in pDst. The substitution will place a space before the attribute name. When the value of pInsertStr is empty then the placeholder is replaced with nothing. This can be called as part of the HTML generation of the inner HTML for a JavaScript control in a component's JSC_GETINNERHTML function.

Parameters:

- **pAttName** (EXTfldval&): The attribute name to insert.
- **pInsertStr** (const char *): The data field which contains the attribute value string to insert.
- **pDst** (EXTfldval&): The destination data field which contains the HTML string in which to place the attribute name/value pair.

Example

```
// HWND pHwnd;
// webClientComponent *object;
// EXTfldval &pInner;
//
str255 innerTemplate = str255(QTEXT("<div$></div>"));
pInner.setChar(innerTemplate);
jsInsertIfNotEmpty("data-text", object->mText, pInner); // results in <div data-text="some text"></div>
```

javaScriptComponent::jsInsertNum()

javaScriptComponent::jsInsertNum(char *pAttName, qlong pValue, EXTfldval& pDst)

Creates a string, in the format of an attribute name/value pair, from the name in pAttName and the numeric value in pValue and inserts it into the first '$' placeholder marker in pDst. A leading space is prepended to the string so it takes the form <space><attname>="<value>". If pValue is zero an empty string is inserted. This can be called as part of the HTML generation of the inner HTML for a JavaScript control in a component's JSC_GETINNERHTML function.

Parameters:

- **pAttName** (char*): The attribute name to insert.
- **pValue** (qlong): The attribute value to insert. If zero an empty string is inserted.
- **pDst** (EXTfldval&): The destination data field which contains the HTML string in which to place the attribute name/value pair.

Example

```
// webClientComponent *object;
// EXTfldval &pInner;
//
str255 innerTemplate = str255(QTEXT("<div$></div>"));
pInner.setChar(innerTemplate);
jsInsertNum("data-max", object->mMax, pInner); // e.g. <div data-max="100"></div>
```

javaScriptComponent::jsInsertStyle()

void javaScriptComponent::jsInsertStyle(HWND pHwnd, WCCcontrol *pControl, EXTfldval &pDest, qdim pWidth, qdim pHeight, javaScriptComponent *pObject)

Inserts the inline style attribute name/value pair into the first '$' placeholder marker in pDest. This can be called as part of the HTML generation of the inner HTML for a JavaScript control in a component's JSC_GETINNERHTML function. The style attribute value will contain the CSS property values set from the component's standard Omnis property values to set characteristics in the JavaScript control such as padding, position, font, alignment, text color, scrollbars, focus and background color. These are included based on the options that have been used in the pControl parameter.

Parameters:

- **pHwnd** (HWND): The child window associated with this component. This is used to obtain values of the Omnis standard properties for this component.
- **pControl** (WCCcontrol): Describes the JavaScript control's required properties and behavior. This is a WCCcontrol structure where the mHtmlOptions member controls which standard CSS values are added and the mAppendControlStyles member appends control-specific styles to the JavaScript control.
- **pDest** (EXTfldval&): The destination data field to contain the HTML string for the control.
- pWidth (qdim): The required CSS width of the inner HTML. This is passed into the JSC_GETINNERHTML function and may be calculated based on border and padding options.
- **pHeight** (qdim): The required CSS height of the inner HTML. This is passed into the JSC_GETINNERHTML function and may be calculated based on border and padding options.
- **pObject** (qdim): An instance of a component object which can be used to retrieve component specific CSS styles.

A typical use of this method is shown in the implementation of jsGenericComponent::jsGetInnerHTML in the jsgeneric.cpp source example.

```
// HWND pHwnd;
// WCCcontrol* pControl;
// EXTfldval &pInner;
// qdim pWidth, pHeight;
// webClientComponent* object;
//
str255 innerTemplate = str255(QTEXT("<div $></div>"));
pInner.setChar(innerTemplate);
jsInsertStyle(pHwnd, pControl, pInner, pWidth, pHeight, object); // e.g. <div style="position:absolute; paddin
```

javaScriptComponent::jsInsertWithAttName()

javaScriptComponent::jsInsertWithAttName(EXTfldval& pInsertStr, EXTfldval& pDst, strxxx &pAttName)

Inserts the HTML attribute value string in pInsertStr, prepended with the HTML attribute name in pAttName, into the first '$' placeholder marker in pDst. The substitution places an HTML attribute name/value pair into the HTML template stored in pDst. This is called as part of the HTML generation of the inner HTML for a control in a component's JSC_GETINNERHTML function.

Parameters:

- **pInsertStr** (EXTfldval&): The data field which contains the attribute value string to insert.
- **pDst** (EXTfldval&): The destination data field which contains the HTML string in which to place the attribute name/value pair.

- **pAttName** (strxxx&): The attribute name to insert.

Example

```
omnis // // HWND pHwnd; // webClientComponent *object; // EXTfldval &pInner; // str255 innerTemplate
= str255(QTEXT("<div $></div>")); pInner.setChar(innerTemplate); str255 dataText(QTEXT("data-text"));
jsInsertWithAttName(object->mText, pInner, dataText); // results in <div data-text="some text"></div>
```

javaScriptComponent::jsStyleAppendColor()

javaScriptComponent::jsStyleAppendColor(HWND pHwnd, attnumber pColorAnum, EXTfldval& pDst, attnumber pAlphaComponent=0)

Uses the Omnis color and alpha property values of the component associated with the pHwnd child window to create the equivalent CSS color property value. The CSS property value will be appended onto the data field in pDest. The attribute numbers passed in pColorAnum and pAlphaComponent identify the color and alpha properties of the component which are to be used. The pColorAnum value will generate a CSS hex color or if non-zero the pAlphaComponent will generate a CSS RGBA color value formed from both parameters.

Parameters:

- **pHwnd** (HWND): The child window associated with a component. This is used to obtain the value of the component properties specified in pColorAnum and pAlphaComponent.
- **pColorAnum** (attnumber): The attribute number which identifies the Omnis property whose value will be used to create the CSS hex color value or the RGB part of the CSS RGBA color value if pAlphaComponent is not zero.
- **pDst** (EXTfldval&): The destination data field which contains the string onto which the CSS color property value will be appended.
- **pAlphaComponent** (attnumber): Optional. Default is zero. If non-zero then this attribute number identifies the Omnis property whose value will be used to create the alpha part of the CSS RGBA color value.

Example

```
// HWND pHwnd;
// EXTfldval &pInner;
// qbool alpha;
// $ - marker to be replaced with CSS color property value
str255 innerTemplate = str255(QTEXT("<div style=\"$\"></div>"));
pInner.setChar(innerTemplate);
EXTfldval fval;
EXTfldval fvalResult;
fvalResult.setEmpty(fftCharacter, dpDefault);
fvalResult.concat(str15(QTEXT("color:")));
if(!alpha)
{
  jsStyleAppendColor(pHwnd, anumBackColor, fvalResult); // append the hex color value using $backcolor
  fval.setChar(str15(QTEXT("$")));
  pInner.replaceStr(fval, fvalResult, qtrue); // e.g. "<div style="color:#0000FF"></div>"
}
else
{
  jsStyleAppendColor(pHwnd, anumBackColor, fvalResult, anumBackgroundAlpha); // append the RGBA color value us
  fval.setChar(str15(QTEXT("$")));
  pInner.replaceStr(fval, fvalResult, qtrue); // e.g. "<div style="color:rgba(0,0,255,0.5020)"></div>"
}
```

javaScriptComponent::jsStyleAppendColorFromValue()

javaScriptComponent::jsStyleAppendColorFromValue(HWND pHwnd, qcol pColorValue, EXTfldval &pDest, qlong pAlpha=-1)

Uses the Omnis qcol color in pColorValue and the alpha value in pAlpha to create the equivalent CSS color property value for the component associated with the child window in pHwnd. The CSS property value will be appended onto the data field in pDest. The qcol value of pColorValue will generate a CSS hex color or if not -1 the pAlpha value will be used to generate a CSS RGBA color value formed from both parameters.

Parameters:

- **pHwnd** (HWND): The child window associated with a component.
- **pColorValue** (qcol): The qcol used to create a CSS hex color value or the RGB part of a CSS RGBA color value if pAlpha is not -1.
- **pDst** (EXTfldval&): The destination data field which contains the string onto which the CSS color property value will be appended.
- **pAlpha** (qlong): Optional. Default is -1. A value of between 0 and 255 will be used to create the alpha part of a CSS RGBA color value.

Example

```
// HWND pHwnd;
// EXTfldval &pInner;
// qbool alpha;
// $ - marker to be replaced with CSS color property value
str255 innerTemplate = str255(QTEXT("<div style=\"$\"></div>"));
pInner.setChar(innerTemplate);
EXTfldval fval;
EXTfldval fvalResult;
fvalResult.setEmpty(fftCharacter, dpDefault);
fvalResult.concat(str15(QTEXT("color:")));
ECOgetProperty(pHwnd, anumBackColor, fval); // get the value of $backcolor
qcol color = fval.getLong();
if(!alpha)
{
  jsStyleAppendColor(pHwnd, color, fvalResult); // append the hex color value
  fval.setChar(str15(QTEXT("$")));
  pInner.replaceStr(fval, fvalResult, qtrue); // e.g. "<div style="color:#0000FF"></div>"
}
else
{
  ECOgetProperty(pHwnd, anumBackgroundAlpha, fval); // get the value of $backalpha
  qlong alphaValue = fval.getLong();
  jsStyleAppendColor(pHwnd, color, fvalResult, alphaValue); // append the RGBA color value
  fval.setChar(str15(QTEXT("$")));
  pInner.replaceStr(fval, fvalResult, qtrue); // e.g. "<div style="color:rgba(0,0,255,0.5020)"></div>"
}
```

javaScriptComponent::jsStyleAppendFontAlign()

void jsStyleAppendFontAlign(qlong pAlignValue, EXTfldval &pDst)

Uses the Omnis qjst text justification constant in pAlignValue to create an equivalent CSS text-align property and appends that CSS property string to the value in pDst. This is called as part of the HTML generation of the inner HTML for a control in a component's JSC_GETINNERHTML function.

Parameters:

- **pAlignValue** (qlong): The Omnis qjst constant, e.g. jstLeft, jstRight, etc.
- **pDst** (EXTfldval&): The destination data field to which to append the CSS property string.

Example

```
// HWND pHwnd;
// EXTfldval &pInner;
// %j - font align marker to be replaced with CSS property value
str255 innerTemplate = str255(QTEXT("<div style=\"%j\"></div>"));
pInner.setChar(innerTemplate);
EXTfldval fval;
EXTfldval fvalResult;
ECOgetProperty(pHwnd, anumAlign, fval); // get the value of $align
fvalResult.setEmpty(fftCharacter, dpDefault);
jsStyleAppendFontAlign(fval.getLong(), fvalResult);
fval.setChar(str15(QTEXT("%j")));
pInner.replaceStr(fval, fvalResult, qtrue);
```

javaScriptComponent::jsStyleAppendFontName()

void javaScriptComponent::jsStyleAppendFontName(EXTfldval& pInsertStr, EXTfldval& pDst)

Uses the Omnis font name in pInsertStr to create an equivalent CSS font-family property and appends that CSS property string to the value in pDst. This is called as part of the HTML generation of the inner HTML for a control in a component's JSC_GETINNERHTML function. It can be used to provide the correct CSS font styling for the text in an HTML control based on a components $font and $fontsize properties.

Parameters:

- **pInsertStr** (EXTfldval&): The Omnis font name, e.g. "Verdana,Arial,Helvetica,Sans-serif".
- **pDst** (EXTfldval&): The destination data field to which to append the CSS property string.

Example

```
// HWND pHwnd;
// EXTfldval &pInner;
// %f - font family marker to be replaced with CSS font property values
str255 innerTemplate = str255(QTEXT("<div style=\"%f\"></div>"));
pInner.setChar(innerTemplate);
EXTfldval fval;
EXTfldval fvalResult;
jsGetFontName(pHwnd, fval); // font name based on $font and $fontsize
jsStyleAppendFontName(fval, fvalResult);
fval.setChar(str15(QTEXT("%f")));
pInner.replaceStr(fval, fvalResult, qtrue);
```

javaScriptComponent::jsStyleAppendFontSize()

void javaScriptComponent::jsStyleAppendFontSize(qlong pSize, EXTfldval& pDst)

Maps the pSize value which will be an Omnis font size value to the equivalent CSS font-size property and appends that string to the value in pDst. This is called as part of the HTML generation of the inner HTML for a control in a component's JSC_GETINNERHTML function. It can be used to provide the correct CSS font size for the text in an HTML control based on a components $fontsize property.

Parameters:

- **pSize** (qlong): The Omnis font size value to be mapped to the equivalent CSS font-size property.
- **pDst** (qlong): The destination data field to which to append the CSS property string.

Example

```
// HWND pHwnd;
// EXTfldval &pInner;
// %z - font size marker to be replaced with CSS font-size property value
str255 innerTemplate = str255(QTEXT("<div style=\"%z\"></div>"));
pInner.setChar(innerTemplate);
EXTfldval fval;
EXTfldval fvalResult;
ECOgetProperty(pHwnd, anumFontsize, fval); // get the $fontsize value
fvalResult.setEmpty(fftCharacter, dpDefault);
jsStyleAppendFontSize(fval.getLong(), fvalResult);
fval.setChar(str15(QTEXT("%z")));
pInner.replaceStr(fval, fvalResult, qtrue);
```

javaScriptComponent::jsStyleAppendFontStyle()

void jsStyleAppendFontStyle(qlong pStyle, EXTfldval& pDst)

Maps the pStyle value which will be an Omnis qsty constant to the equivalent CSS font properties and appends that CSS string to the value in pDst. This is called as part of the HTML generation of the inner HTML for a control in a component's JSC_GETINNERHTML function. It can be used to provide the correct CSS font styling for the text in an HTML control based on a component's $fontstyle property.

Parameters:

- **pStyle** (qlong): The Omnis qsty value, e.g. styPlain, styBold, styItalic, styUnderline. This will be mapped to the equivalent CSS font property.
- **pDst** (qlong): The destination data field to which to append the CSS property string.

Example

```
// HWND pHwnd;
// EXTfldval &pInner;
// %s - font style marker to be replaced with CSS property values
str255 innerTemplate = str255(QTEXT("<div style=\"%s\"></div>"));
pInner.setChar(innerTemplate);
EXTfldval fval;
EXTfldval fvalResult;
ECOgetProperty(pHwnd, anumFontstyle, fval); // get the $fontstyle value
fvalResult.setEmpty(fftCharacter, dpDefault);
jsStyleAppendFontStyle(fval.getLong(), fvalResult);
fval.setChar(str15(QTEXT("%s")));
pInner.replaceStr(fval, fvalResult, qtrue);
```

javaScriptComponent::setRedrawOnSize()

```
void javaScriptComponent::setRedrawOnSize()
```

This is called to specify that all of the client area of a component child window is invalidated when the width or height of its window changes. This allows the correct redrawing of the component contents in a design window. By default only the uncovered areas are invalidated. If the contents of the component need to be adjusted to fit the width and height, this method should be called in the component subclass constructor.

Virtual Methods

The following are methods that can be overridden by a component subclass.

javaScriptComponent::getCtrlName

```
qbool getCtrlName(EXTfldval& pFval)
```

This method is overridden by a component subclass when it supports an Omnis property to provide the name of an existing JavaScript control. The name is used to identify an existing control which will have its inner HTML generated from this component to override the HTML that the control would normally have used. The control name should be set in the pFval parameter.

A component that uses this mechanism will also specify the Omnis property attribute number in the mControlNameAnum member of the WCCcontrol structure.

The default base class implementation does nothing and returns qfalse.

Parameters:

- **pFval** (EXTfldval): The destination data field that contains the name of the JavaScript control.

Returns:

- (qbool): qtrue/qfalse for success/failure.

Example:

The JSHTML component uses this method to get the JavaScript control class that was set in its $ctrlname property.

JavaScript Control Reference

Fundamentals

Your JavaScript control must include a few fundamentals, as follows.

Control Name

Your control name should match that defined in your C++ component's *WCCcontrol* structure.

The Structure of a JavaScript Control

All JavaScript controls share an overall html structure. They comprise an outer frame element, and an inner client element.

Frame Element:

- Has no content, other than the client element.
- Contains the Omnis properties which pertain to the container of the control, e.g. its border.

Client Element:

- Initially has its content set to that specified in the innerHTML() method of the C++ component.
- Should contain the entire control structure.

You should only ever add child elements to the Client element (or its children), never to the Frame element.

JavaScript Control code structure

Most methods of a JavaScript control are added to the object's prototype. This helps reduce memory usage when lots of instances of the control are present (as may be the case with a re-useable component such as you are creating).

The standard way that we create a JS control is by creating a constructor function for the control (generally at the bottom of the file), and then create an Immediately Invoked Function Expression (IIFE) which creates the prototype for the control.

The prototype must inherit from the ctrl_base class (the base class for all JavaScript controls). This can be achieved by setting the prototype of your new control to be initially equal to a new instance of ctrl_base, then you can extend it.

The following gives a brief example of the structure:

```
ctrl_mycontrol.prototype = (function() {

    var ctrl = Object.create(ctrl_base.prototype); // The object which will become the prototype - inherited f
    var myConst = 123; // A constant value


    ctrl.publicMethod1 = function(p1) {
        // public methods which are inherited from the base class, and called by the JS Client framework will 
        this.instVar = p1;
        privateMethod1.call(this, "abc"); // Make sure to use call() to pass the appropriate value for 'this'.
    };


    function privateMethod1(p1) {
      // Private method code.
    }
    return ctrl; // Return the prototype object.

})();

/**
* constructor for our control
* @constructor
*/
function ctrl_mycontrol()
{
    this.init_class_inst(); // initialize our class
}
```

Control Methods

Your control must also implement the following methods:

- Constructor
- init_class_inst
- init_ctrl_inst

You will probably want to implement more Control Methods than these, and there are several optional ones available.

The most-used of these are already provided as stubs in the generic.js file, included in the SDK, so this is always a good starting point from which you can extend the functionality of your own control.

Loading Controls

In order for your control to work, you must tell the .htm page to load your .js file as a resource.

- Locate your form’s .htm file, in Omnis’ html folder.

Alternatively, you may prefer to use your jsctempl.htm file from this same location, so whenever you generate a form’s .htm file using Test Form, it will be created from this template.

- Edit the file, adding a <script> tag to the <head> to load your .js file after all other .js files.

Do the same for any custom CSS files you are using for your control. Add these after all other CSS files except user.css, but before the <script> tags.

Themed Colors

With the introduction of theme colors to Studio 10.2 you can support these in your control.

C++ Component

When creating its *innerHTML*, your component should use the jsInsertColor() method to insert color properties for your control to use. This will insert the property value as an integer: either a Theme Color Constant or an explicit Omnis RGB Integer.

Example

```
jsInsertColor("data-mycolor", color, pInner); // inserts the value of color into the data-mycolor attribute in
```

JS Control

Your JavaScript control should then be able to read this in and store it as an integer to work with the Theme related methods.

Example

```
let myColor = +this.clientElem.getAttribute("data-mycolor"); // coerce the value of "data-mycolor" into an int
```

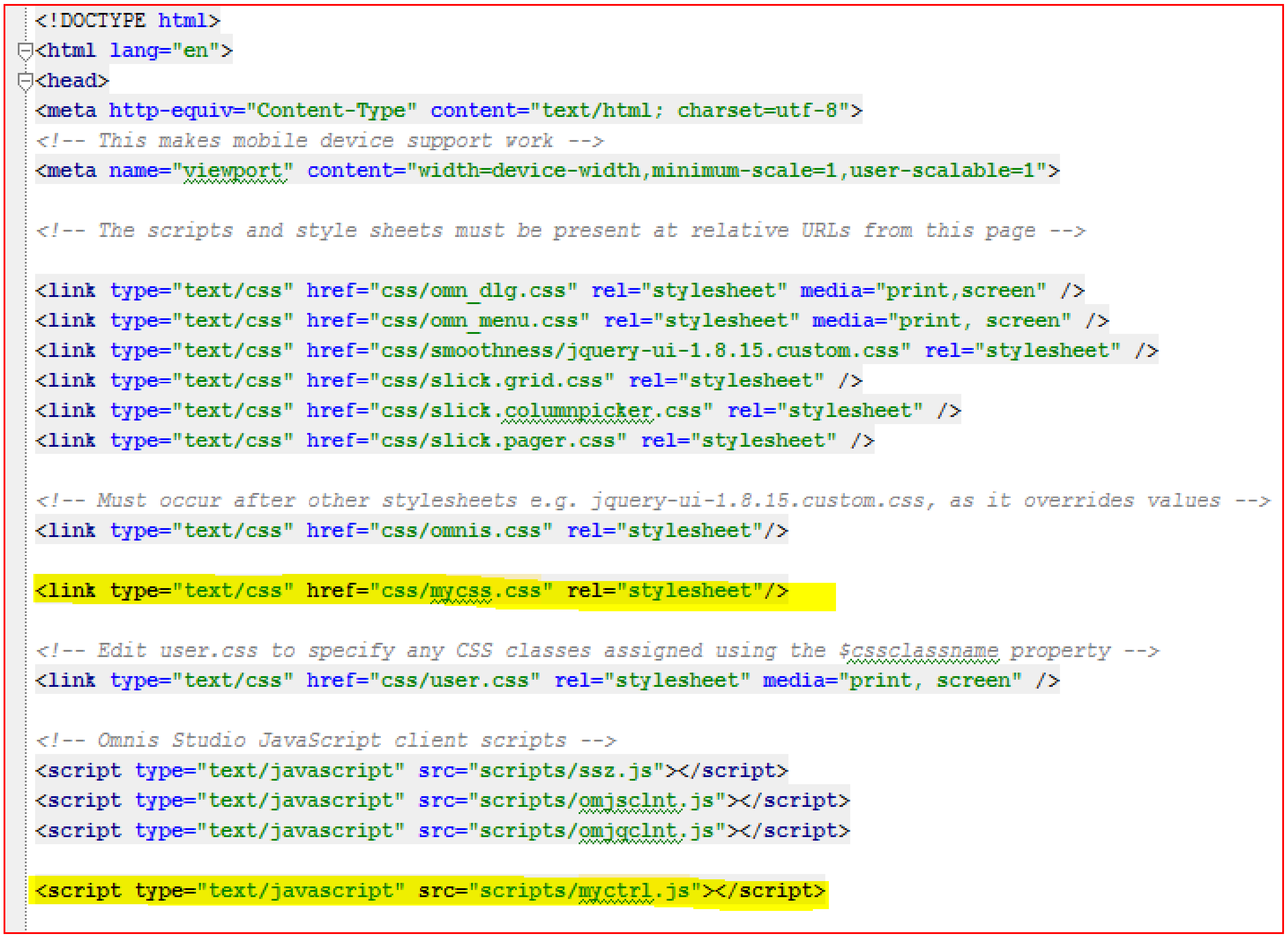
Color properties that are set at runtime will be received from the server as integers in your control’s setProperty() method.

Default Colors

Default property colors are all handled client-side. Your control inherits the DefaultColors property, which is an object that is used to determine the color to apply to certain elements (usually when its associated property is set to *kColorDefault*).

Use this in conjunction with *ctrl_base.DEFAULT_COLOR_NAMES* (see Constants) to access the correct member. You can override the properties here to change the default colors of your control.

Example



```
<!DOCTYPE html>
<html lang="en">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<!-- This makes mobile device support work -->
<meta name="viewport" content="width=device-width,minimum-scale=1,user-scalable=1">

<!-- The scripts and style sheets must be present at relative URLs from this page -->

<link type="text/css" href="css/omn_dlg.css" rel="stylesheet" media="print,screen" />
<link type="text/css" href="css/omn_menu.css" rel="stylesheet" media="print, screen" />
<link type="text/css" href="css/smoothness/jquery-ui-1.8.15.custom.css" rel="stylesheet" />
<link type="text/css" href="css/slick.grid.css" rel="stylesheet" />
<link type="text/css" href="css/slick.columnpicker.css" rel="stylesheet" />
<link type="text/css" href="css/slick.pager.css" rel="stylesheet" />

<!-- Must occur after other stylesheets e.g. jquery-ui-1.8.15.custom.css, as it overrides values -->
<link type="text/css" href="css/omnis.css" rel="stylesheet"/>

<link type="text/css" href="css/mycss.css" rel="stylesheet"/>

<!-- Edit user.css to specify any CSS classes assigned using the $cssclassname property -->
<link type="text/css" href="css/user.css" rel="stylesheet" media="print, screen" />

<!-- Omnis Studio JavaScript client scripts -->
<script type="text/javascript" src="scripts/ssz.js"></script>
<script type="text/javascript" src="scripts/omjsclnt.js"></script>
<script type="text/javascript" src="scripts/omjqclnt.js"></script>

<script type="text/javascript" src="scripts/myctrl.js"></script>
```

Figure 7:

```
this.DefaultColors[ctrl_base.DEFAULT_COLOR_NAMES.BACKGROUND] = Theme.COLORS.primary // sets the default backgr
```

You can extend this further to add custom default colors to your control. You should create a new object of color names and add the equivalent members the DefaultColors property.

You will then be able to use this further on in your code when setting elements' colors to ensure they have the correct color when the property relating to it is set to kColorDefault.

Example

```
// Set up default colors
const MY_COLOR_NAMES = {
MY_COLOR: "MY_COLOR" // custom color name
};
this.DefaultColors[MY_COLOR_NAMES.MY_COLOR] = Theme.COLORS.secondary; // assigns secondary color as the defaul

// Assigning a color
const theme = this.getTheme(); // gets the Theme object
const myElem = document.getElementById("myElement");
myElem.style.backgroundColor = theme.getColorString(value, this.DefaultColors[MY_COLOR_NAMES.MY_COLOR]); // se
```

Text Color Handling

There is special handling for text colors to attempt to use the right default text color on top of the background color set.

For example, if you have set the background color of an element to secondary color, then when setting the text color, if its property is set to *kColorDefault*, it will use secondary text color.

To do this you need to resolve the background color first (as it may be set to *kColorDefault*), and then pass this color into getTextColorString() as its 'onColor'. See also resolveDefaultColor().

Example

```
const theme = this.getTheme(); // gets the Theme object
const myElem = document.getElementById("myElement");

const resolvedBackColor = this.resolveDefaultColor(backColor, ctrl_base.DEFAULT_COLOR_NAMES.BACKGROUND); // re
myElem.style.color = theme.getTextColorString(value, resolvedBackColor, Theme.COLORS.primaryText) // sets the
```

Constants

Constants used throughout the JavaScript client are generally defined as part of an enumerated type.

Following are the base enumerated types you may need to make use of.

eAnums

Attribute numbers, used for ascertaining the context of method calls.

Values

- **priv** = -1 (Special value for private method calls)
- **none** = 0
- **$cfield** = 89 (The current field - same as $cfield in Omnis)
- **$cinst** = 77 (The current form instance - same as $cinst in Omnis)

- **$cwind** = 66 (The current top-level form instance - same as $cwind in Omnis).

eBaseBorder

Set of border style constants.

Values

- **None** = 0
- **Plain** = 1
- **Inset** = 2
- **Embossed** = 3
- **SingleInset** = 9
- **SingleEmbossed** = 10
- **Default** = 100

eBaseEvent

Standard events that are supported by the base class and that can be sent to the server.

Values

- **evNone** = 0
- **evBefore** = -3
- **evAfter** = -4
- **vDoubleClick** = -6
- **evTabSelected** = -43
- **evCellChanged** = -53
- **evOpenContextMenu** = -58
- **evHeaderClick** = -68
- **evExecuteContextMenu** = -90
- **evHeadedListDisplayOrderChanged** = -91
- **evUserChangedPane** = -94
- **evFormToTop** = -77
- **evScreenOrientationChanged** = -89
- **evSubFormToTop** = -92
- **evAnimationsComplete** = -93
- **evDragBorder** = -95

- **evLayoutChanged** = -106

eBaseProperties

The group of constants describing the base properties. They correspond to the Omnis properties of the same name.

Values

- **name** = -1
- **ident** = -25
- **iconid** = -250
- **tooltip** = -262
- **title** = -400
- **top** = -401
- **left** = -402
- **height** = -403
- **width** = -404
- **horzscroll** = -409
- **vertscroll** = -410
- **backcolor** = -416
- **backpattern** = -417
- **dragborder** = -438
- **order** = -440
- **dataname** = -433
- **text** = -444
- **visible** = -446
- **uppercase** = -453
- **negallowed** = -454
- **zeroempty** = -455
- **multipleselect** = -463
- **forecolor** = -464
- **fontsize** = -466
- **fontstyle** = -467
- **align** = -468
- **linestyle** = -469
- **edgefloat** = -470
- **container** = -473

- **effect** = -494
- **hscroll** = -498
- **vscroll** = -499
- **enabled** = -506
- **checked** = -571
- **fontname** = -690
- **bordercolor** = -720
- **textcolor** = -721
- **contextmenu** = -760
- **fieldstyle** = -887
- **events** = -956
- **okkeyobject** = -980
- **cancelkeyobject** = -985
- **userinfo** = -1091
- **disabledefaultcontextmenu** = -1117
- **disablesystemfocus** = -1130
- **alpha** = -1200
- **screensize** = -1201
- **commandid** = -1204
- **remotemenu** = -1205
- **backalpha** = -1227
- **beginanimations** = -1232
- **commitanimations** = -1233
- **vertcentertext** = -1243
- **jsdateformat** = -1250
- **jsdateformatcustom** = -1251
- **borderradius** = -1252
- **autoscroll** = -1253
- **resizemode** = -1255
- **jsnumberformat** = -1256
- **cssclassname** = -1257
- **sqlobject** = -1265
- **errorline** = -1303
- **errortextpos** = -1305
- **textishtml** = “$textishtml”
- **pagesize** = -1307
- **visibleinbreakpoint** = -1324

- **autocomplete** = “$autocomplete
- **autocapitalize** = “$autocapitalize”
- **autocorrect** = “$autocorrect”
- **preventlayoutanimation** = -1341

eClientPlatforms

enum of possible client platforms.

Values

- **Unknown** = -1
- **Windows** = 0
- **macOS** = 1
- **Linux** = 2
- **iOS** = 3

eDateParts

The group of date part constants used in omnis_date methods.

Values

- **kYear** = 1
- **kMonth** = 2
- **kWeek** = 3
- **kDayOfYear** = 4
- **kQuarter** = 5
- **kMonthofQuarter** = 6
- **kWeekofQuarter** = 7
- **kDayofQuarter** = 8
- **kWeekofMonth** = 9
- **kDay** = 10
- **kHour** = 19
- **kMinute** = 20
- **kSecond** = 21
- **kCentiSecond** = 22

eDoMethodFlags

Set of flags which can be used when calling methods.

You should **OR** values together to set multiple flags. E.g. eDoMethodFlags.completionEvent | eDoMethodFlags.noOverlay.

Values

- **completionEvent** = 1 (If set, the return value of a server method is to be passed to form (not field) client method <omnisMethodName_return> for the form)
- **clientOnly** = 2 (If set, only a client method is to be called)
- **noOverlay** = 4 (If not set, a user-interface protection overlay will be applied to the form, to prevent further interaction until the method returns)
- **noLog** = 8 (If not set, and clientOnly is set, when running against an Omnis Design Server, a message will be logged to the console if a client method could not be called).

eOmnisDataSubTypes

The group of values used to denote Omnis data subtypes.

Note that JavaScript is not strongly typed, so will not restrict data you enter to these types. If necessary, you should take care to ensure the data you enter matches these formats.

Values	
	default = 0
Integer Subtypes	intShort = 32 int32 = 0 int64 = 64
Number Subtypes	numberShort = 32 numberRealDps = 31 (Logical & this with the required number of dps. E.g. eOmnisDataSubTypes.numberRealDps & 4)
Character Subtypes	characterSimple = 0 characterNational = 1
Date Subtypes	date1900 = 0 date1980 = 1 date2000 = 2 dateTime1900 = 3 dateTime1980 = 4 dateTime2000 = 5 time = 6 dateTimeC = 7 (date time including a century) timestamp = 1000 (full date time stamp) Values 1001-1030 can also be passed as an index into the date formats defined in #DFORMS.
List Subtypes	listRow = 1 (If used with a type of fftList, the type will become row)

eOmnisDataTypes

The group of constants describing the various Omnis data types.

Values

- **fftNone** = 20
- **fftCharacter** = 21
- **fftBoolean** = 22
- **fftDate** = 23

- **fftSequence** = 24
- **fftNumber** = 25
- **fftInteger** = 26
- **fftPicture** = 27
- **fftBinary** = 28
- **fftList** = 29
- **fftRdef** = 30
- **fftCrb** = 31
- **fftFieldname** = 32
- **fftItemRef** = 33
- **fftCalc** = 34
- **fftConstant** = 35
- **fftRow** = 36
- **fftObject** = 37
- **fftObjref** = 38

ctrl_base.DEAFULT_COLOR_NAMES

Set of keys to be used with the controls DefaultColors member.

Values

- **BACKGROUND** = “background” (The key within this.DefaultColors referencing the default background color)
- **TEXT** = “text” (The key within this.DefaultColors the referencing default text color)
- **BORDER** = “border” (The key within this.DefaultColors referencing the default border color)
- **SYSTEM_FOCUS** = “SYSTEM_FOCUS” (The key within this.DefaultColors referencing the default system focus color)

Info

Date Formatting

Areas of the JavaScript client allow you to specify formatting strings to determine how Date/time data should be displayed.

A date format string can make use of any of the following control characters:

- **a** - am/pm
- **A** - AM/PM
- **s** - Hundredths of a second. E.g. “007”
- **S** - Seconds (0-59)
- **N** - Minutes (0-59)
- **V** - Short day of week name. e.g. “Fri”
- **w** - Full day of week name. e.g. “Friday”

- **D** - The day of the month, as 2 digits. e.g: “02”.
- **j** - Day of month with no leading zero. e.g. “2”.
- **d** - The ordinal day of the month. e.g “12th”.
- **E** - The day of the year (1 - 366).
- **H** - Hour with leading 0 (0 - 23).
- **h** - Hour with no leading 0 (1 - 12).
- **K** - hour with no leading zero (0..23)
- **k** - hour with leading zero (01..12)
- **M** - Month of year, with leading 0. e.g. “06”.
- **P** - Month of year, with no leading zero. e.g. “6”
- **m** - Short month name. e.g. “JUN”.
- **n** - Full month name. e.g. “June”
- **Y** - Year as 2 digits. e.g. “14”
- **y** - Full year. e.g. “2014”
- **O** - Timezone offset (+01:00)

Other characters will appear as-is.

E.g. “D/M/y H:N:S.s a” would display like:

```
"26/09/2014 17:30:05.012 pm"
```

Definitions

Color Pair

DEPRECATED in 10.2

Color Pairs are no longer needed (all browsers now support alpha), and they do not support Themed Colors. You should instead use RGB Integers or color constant values.

A Color Pair is a string defining a solid color and a color with alpha support.

Functionality within the JavaScript client makes use of these Color Pairs in order to support browsers which do not support RGBA colors.

The string must be formatted as the solid color, followed by the alpha color, separated by a semi colon:

“<solid color>;<alpha color>”

The solid color can have either the “#RRGGBB” format, or “rgb(r,g,b)” format.

The alpha color must have the “rgba(r,g,b,a)” format.

Omnis RGB Integer

Omnis RGB Integers are the standard way to represent colors in the JS Client.

They describe a color, with no alpha component. They are therefore 3 byte unsigned integers, and so have a maximum value of 16777215.

They are similar to a standard decimal color, but with the byte order reversed.

Therefore, reading the byte order from left to right.

- The first byte is the blue component.
- The second byte is the green component.
- The third byte is the red component.

RGB Integers with a *negative* value indicate that it represents a color constant, such as a themed color.

You can convert a CSS color string to an Omnis RGB Integer using the jOmnis.rgbFromCssColor() method.

Similarly, you can convert an Omnis RGB Integer to a CSS color string using <theme>.getColorString() (or <theme>.getTextColorString() if it's intended for use as a text color).

Number Formatting

Areas of the JavaScript client allow you to specify formatting strings to determine how numeric data should be displayed.

See the Number Formatting section of the Creating Web & Mobile Apps document (under "JavaScript Component Properties").

JavaScript Structures

omnis_cols

An omnis_cols is an object which represents a list's columns group (equivalent to $cols).

omnis_cols()

omnis_cols(pList)

Constructor for an omnis_cols object.

Parameters:

- **pList** (omnis_raw_list): Raw list data. See getListData() for info on how to get an instance of this object.

Returns:

- (omnis_cols): A new omnis_cols object, relating to the passed list.

Example

```
var myCols = new omnis_cols(myList.getListdata());
```

$add()

$add(pName, pType, pSubType, pLen)

Adds a new column onto the end of the list's columns.

Parameters:

- **pName** (String): The column name.
- **pType** (Integer): The column type. One of the eOmnisDataTypes values.
- **pSubType** (Integer): The column subtype. One of the eOmnisDataSubTypes values.
- **pLen** (Integer): The length of the column data type. E.g. character limit.

Alternatively, you can instead just pass an array, the first element of which is the name of an Omnis Instance variable, and the column will be defined from the instance variable's definition.

Returns:

- (Object): The new omnis_list_col column object.

Example

```
var newCol = myCols.$add("Age", eOmnisDataTypes.fftInteger, eOmnisDataSubTypes.intShort, 0);
```

$count()

$count()

Returns the number of columns.

Parameters:

- None

Returns:

- (Integer): Number of columns.

Example

```
var colCount = myCols.$count();
```

getCanAssign( )

getCanAssign(pColumn, pPropNumber)

Returns whether the specified property can be changed for the specified column.

Parameters:

- **pColumn** (String or Integer): The column name or number (optionally prefixed with “C”).
- **pPropNumber** (Integer): The property number to query.

Returns:

- (Boolean): Whether the property can be assigned to.

Example

```
var canAssign = myCols.getCanAssign("Quantity", eBaseProperties.name); // checks whether we can change the nam
```

getProperty( )

getProperty(pColumn, pPropNumber)

Gets a property from a column.

Parameters:

- **pColumn** (String or Integer): The column name or number (optionally prefixed with “C”).
- **pPropNumber** (Integer): The property to query. One of:
  eListProperties.coltype
  eListProperties.colsubtype
  eListProperties.colsublen
  eBaseProperties.ident
  eBaseProperties.name

Returns:

- (Var): The value of the column’s property.

Example

```
var colName = myCols.getProperty(3, eBaseProperties.name); // Get column 3's name
```

getValue()

getValue(pName)

Gets a column by name or column number.

Parameters:

- **pName** (String or Integer): The column name or column number (optionally prefixed by “C”).

Returns:

- (Object): The column omnis_list_col object instance.

Example

```
var col3 = myCols.getValue(3);
```

setPropCli( )

setPropCli(pColumn, pPropNumber, pPropValue)

Sets a property of a column to the specified value.

Parameters:

- **pColumn** (String or Integer): The column name or number (optionally prefixed with “C”).
- **pPropNumber** (Integer): The property to set. One of:
  eListProperties.coltype
  eListProperties.colsubtype
  eListProperties.colsublen
  eBaseProperties.name
- **pPropValue** (Var): The new value to set the property to.

Returns:

- (Boolean): Success

Example

```
var success = myCols.setPropCli("Quantity", eBaseProperties.name, "Amount"); // Rename the Quantities column t
```

$addafter()

$addafter(pAfterCol, pName, pType, pSubType, pLen)

Adds a new column after the specified column.

Parameters:

- **pAfterCol** (Integer or Object): The column number, or column object instance, to insert the new column after.

- **pName** (String): The column name.
- **pType** (Integer): The column type. One of the eOmnisDataTypes values.
- **pSubType** (Integer): The column subtype. One of the eOmnisDataSubTypes values.
- **pLen** (Integer): The length of the column data type. E.g. character limit.

Alternatively, you can instead just pass an array, followed by pAfterCol. The second element of the array should be the name of an Omnis Instance variable, and the column will be defined from the instance variable's definition.

Returns:

- (Object): A reference to the newly created omnis_list_col column object.

Example

```
var newCol = myCols.$addafter(2, "NewCol", eOmnisDataTypes.fftCharacter, eOmnisDataSubTypes.characterSimple, 1
// Or...
var newCol = myCols.$addafter(["","iVar1"],2); // Add column defined from instance variable iVar1 as column 3
```

$addbefore()

$addbefore(pBeforeCol, pName, pType, pSubType, pLen)

Adds a new column before the specified column.

Parameters:

- **pBeforeCol** (Integer or Object): The column number, or column object instance, to insert the new column before.
- **pName** (String): The column name.
- **pType** (Integer): The column type. One of the eOmnisDataTypes values.
- **pSubType** (Integer): The column subtype. One of the eOmnisDataSubTypes values.
- **pLen** (Integer): The length of the column data type. E.g. character limit.

Returns:

- (Object): A reference to the newly created omnis_list_col column object.

Example

```
var newCol = myCols.$addbefore(2, "NewCol", eOmnisDataTypes.fftCharacter, eOmnisDataSubTypes.characterSimple,
// Or...
var newCol = myCols.$addbefore(["","iVar1"],2); // Add column defined from instance variable iVar1 as column 3
```

$remove()

$remove(pCol)

Removes a column from the list.

Parameters:

- **pCol** (Object or Integer): The column number, or column object to remove.

Returns:

- None

Example

```
mCols.$remove(3); // Remove column 3 from the list.
```

omnis_date

This object represents an Omnis Date, and provides various methods for date operations.

An omnis_date instance has the following properties:

- dtY – Year
- dtM – Month
- dtD – Day
- dtH – Hour
- dtN – Minutes
- dtS – Seconds
- dtG - Milliseconds

omnis_date()

omnis_date(pValue)

Constructor. Creates a new omnis_date object.

Parameters:

- **pValue** (Object): The initial date.
  null - To create a new omnis_date using the current date/time.
  JavaScript Date Object - constructs an omnis_date from the value of a standard JavaScript Date object.
  omnis_date instance - constructs a new omnis_date from the value of another omnis_date object.

Returns:

- (Object): a new omnis_date object instance.

Example

```
var rightNow = new omnis_date(null);
```

equals()

equals(pOmnisDate)

Compares this omnis_date instance to another.

Parameters:

- **pOmnisDate** (Object): Another omnis_date instance to check for equality.

Returns:

- (Boolean): Whether the two omnis_date objects are equal.

Example

```
var identical = myDate.equals(myOtherDate);
```

toJavaScriptDate()

toJavaScriptDate()

Returns the date specified by the omnis_date instance as a standard JavaScript Date object

Parameters:

- None

Returns:

- (Object): JavaScript Date object representing this omnis_date.

Example

```
var jsDate = myDate.toJavaScriptDate();
```

getHasTime()

getHasTime(pOmnisDate) **Static function**

Returns whether the passed omnis_date instance has a time component.

This is a static function, and does not require an omnis_date instance to call it.

Parameters:

- **pOmnisDate** (Object): The omnis_date instance to check for a time component.

Returns:

- (Boolean): True if the date includes a time component, false if not.

Example

```
var myDate = new omnis_date(null);
omnis_date.getHasTime(myDate);
```

add()

add(pPart, pAmount)

Adds pAmount units to the date's component specified by pPart.

Does not modify the original omnis_date, but returns a new instance with the modifications.

Parameters:

- **pPart** (Integer): The date component to add to. One of the eDateParts constants.
- **pAmount** (Integer): The number of units to increase the date component by. (Use negative values to decrease).

Returns:

- (Object): A new omnis_date instance with pAmount added to pPart.

Example

```
var myDate = new omnis_date(null);
myDate = myDate.add(eDateParts.kDay, 10); // Add 10 days to myDate
```

getFormat()

getFormat()

Gets the format string for this date, based on its subtype (subtype only applies if it's an Omnis instance variable).

Parameters:

- None

Returns:

- (String): The date format string for this omnis_date instance. E.g. "j m y H:N:S"

Example

```
var fmtString = myDate.getFormat();
```

omnis_list

The omnis_list is the client-side representation of an Omnis list or row variable.

omnis_list()

omnis_list(pOmnisList)

Constructor for an omnis_list object.

Parameters:

- **pOmnisList** (Object): Pass null to create a new, empty, omnis_list. Or pass an omnis_raw_list instance to create a copy of it.

Returns:

- (omnis_list): The new omnis_list object.

Example:

```
var myList = new omnis_list(null);
var myListCopy = new omnis_list(myList.getListData());
```

getChar()

getChar(pCol, pRowNumber, pZeroEmpty)

Returns character data for the specified column and row.

Parameters:

- **pCol** (String or Integer): Can be either the column name, or the column number (1-indexed).
- **pRowNumber** (Integer): The row number (1-indexed). If zero, the current line will be used.
- **pZeroEmpty** (Boolean): If true, zero values are returned as an empty string.

Returns:

- (String): The value specified, converted to a string.

Example:

```
var qntString = myList.getChar("quantity", 2, true);
```

getData( )

getData(pCol, pRowNumber)

Returns the data in the cell specified by the column and row.

Parameters:

- **pCol** (String or Integer): Can be either the column name, or the column number (1-indexed).
- **pRowNumber** (Integer): The row number (1-indexed). If zero, the current line will be used.

Returns:

- (Var): The data from the specified position in the list.

Example:

```
var name = myList.getData("Name", 2);
```

setData()

setData(pCol, pRowNumber, pNewData)

Sets the data in the specified row and column.

Parameters:

- **pCol** (String or Integer): Can be either the column name, or the column number (1-indexed).
- **pRowNumber** (Integer): The row number (1-indexed). If empty or zero, the current line will be used.
- **pNewData**(Var): The new data to apply to the cell.

Returns:

- (Boolean): Whether the new value was successfully applied.

Example:

```
var success = myList.setData("Name", 2, "Bert");
```

search()

search(pCol, pSearchValue, pSetCurrentRow, pSelectMatches, pDeSelectNonMatches)

Searches a column in the list and returns the first matching row.

Parameters:

- **pCol** (String or Integer): Can be either the column name, or the column number (1-indexed).
- **pSearchValue** (Var): The value to search for.
- **pSetCurrentRow** (Boolean): Whether the list's current row should be set to the returned row.
- **pSelectMatches** (Boolean): If true, all matching rows are selected.
- **pDeselectNonMatches** (Boolean): If true, all non-matching rows are de-selected.

Returns:

- (Integer): The first matching row number (1-indexed), or zero if no match is found.

Example:

```
var bertsRowNum = myList.search("Name", "Bert", false, false, false);
```

getRowSelectionState()

getRowSelectionState(pRowNumber)

Gets the selection state for the specified row.

Parameters:

- **pRowNumber** (Integer): The row number (1-indexed). If zero, the current line will be used.

Returns:

- (Integer): 1 if the row is selected, 0 if not.

Example:

```
var isSelected = myList.getRowSelectionState(3);
```

setRowSelectionState()

setRowSelectionState(pRowNumber, pNewSelectionState)

Sets the selection state for the specified row.

Parameters:

- **pRowNumber** (Integer): The row number (1-indexed). If zero, the current line will be used.
- **pNewSelectionState** (Integer): The new selection state. (1=selected, 0=not selected)

Returns:

- (Boolean): Whether the selection state was successfully changed.

Example:

```
var success = myList.setRowSelectionState(3, 1); // Select row 3
```

setRowSelectionStateAllRows()

setRowSelectionStateAllRows(pNewSelectionState)

Sets the selection state for all rows in a list.

Parameters:

- **pNewSelectionState** (Integer): New selection state. 1 for selected, 0 for un-selected.

Returns:

- False

Example:

```
myList.setRowSelectionStateAllRows(0); // De-select all rows
```

getCurrentRow()

getCurrentRow()

Returns the current row number for the list.

Parameters:

- None

Returns:

- (Integer): The current line in the list.

Example:

```
var curLine = myList.getCurrentRow();
```

setCurrentRow()

setCurrentRow(pNewCurrentRow)

Sets the current line in the list.

Parameters:

- **pNewCurrentRow** (Integer): The new current row number.

Returns:

- (Boolean): Whether the update was successful.

Example:

```
var success = myList.setCurrentRow(3);
```

getRowCount()

getRowCount()

Gets the number of rows in the list.

Parameters:

- None

Returns:

- (Integer): Number of rows in the list.

Example:

```
var rowCount = myList.getRowCount();
```

getColumnCount()

getColumnCount()

Gets the number of columns in the list.

Parameters:

- None

Returns:

- (Integer): Number of columns in the list.

Example:

```
var colCount = myList.getColumnCount();
```

addRow()

addRow(pBeforeRowNumber, pColumnCount)

Adds an empty row to the list.

Note that you can only add a row to a list (not a row).

Parameters:

- **pBeforeRowNumber** (Integer): The row number the new row will be inserted before. If 0, it will be added to the end of the list.
- **pColumnCount** (Integer): The initial number of columns for the new row.

Returns:

- (Integer): The row number of the newly added row, or INVALID_LIST_OPERATION (-1), if the action failed.

Example:

```
var newRowNum = myList.addRow(3,2);
```

deleteRows()

deleteRows(pActionOrRow)

Deletes the specified rows from the list.

Parameters:

- **pActionOrRow** (Integer): One of:
  Row number to delete.
  An eListDeleteRow enumerated type. (Current, Selected or Deselected members)

Returns:

- (Type): Row number deleted, or number of rows deleted for eListDeleteRow.Selected or Deselected action.

Example:

```
var deletedCount = myList.deleteRows(eListDeleteRow.Selected); // Delete all selected rows.
```

findColumn()

findColumn(pCol, pBeginsWith)

Gets the specified column's column number in the list or row.

Parameters:

- **pCol** (String or Integer): One of:
  The column number.
  The column name.
  The column number, prefixed by "C". E.g. "C4".
- **pBeginsWith** (Boolean): If true, it will return the first column whose name begins with the string passed in pCol. Defaults to false.

Returns:

- (Integer): The column number (1-indexed)

Example:

```
var nameCol = myList.findColumn("Name", false);
```

getRowArray()

getRowArray(pRowNumber)

Returns a row's data as an array. Each element of the array being a column of the row.

Parameters:

- **pRowNumber** (Integer): The row number (1-based).

Returns:

- (Array): The row's data. Each element of the array corresponds to the value of a column in the row.

Example:

```
var myData = myList.getRowArray(3);
var colTwoValue = myData[1];
```

getListData()

getListData()

Returns the raw list data (an omnis_raw_list object), from the list. This is needed to pass into many of the row/col methods.

Parameters:

- None

Returns:

- (Object): The omnis_raw_list data structure.

Example

```
var rawList = myList.getListData();
```

omnis_list_col

This object represents a single column in a list.

Most column actions are performed using the omnis_cols object. However, the omnis_cols methods often involve passing instances of this object around.

```
var myCols = new omnis_cols(myList.getListData());
var myCol = myCols.getValue(3); // Get the third column in the list.
```

$clear()

$clear()

Clears all values in this column.

Parameters:

- None

Returns:

- (Boolean): Success.

Example

```
var success = myCol.$clear();
```

$count( )

$count(pSelOnly)

Returns the number of (optionally selected) rows in the column.

Parameters:

- **pSelOnly** (Optional) (Boolean): If true, only selected rows will be counted. Defaults to false.

Returns:

- (Integer): The row count.

Example

```
int selectedRows = myCol.$count(true);
```

$removeduplicates()

$removeduplicates(pSortNow, pIgnoreCase)

Removes any rows which have duplicate values in this column.

Duplicates are removed from the bottom up.

Parameters:

- **pSortNow** (Optional) (Boolean): If true, the list will be sorted by this column before duplicates are removed. Defaults to false.
- **pIgnoreCase** (Optional) (Boolean): If true, checks for duplicity will ignore case. Defaults to false.

Returns:

- (Integer): The number of deleted rows.

Example

```
var removedLines = myCol.$removeduplicates(true, true);
```

omnis_raw_list

The omnis_raw_list object is a component of the omnis_list object.

The omnis_raw_list functions are not exposed as part of this API, as you should use the higher-level methods available from the omnis_list object.

The **omnis_list** wrapper of an **omnis_raw_list** can be obtained from the omnis_raw_list's **__list** property.

JavaScript API Reference

Control

The following Instance Methods must be called on an instance of a ctrl object. This is a JavaScript control object, and is what you are creating.

As such, you can call these from your control's class methods using the *this* keyword.

Example

```
ctrl.myMethod = function() {
  var elem = this.getClientElem(); // "this" in this context is an instance of the ctrl_generic control.
} ;
```

There are also several Inherited Methods which you can implement your class to handle specific events.

Control – Inherited Methods

The following are inherited methods which you can implement in your control, in order to catch specific 'events'.

You can add such an inherited method as shown below:

```
<ctrl_name>.prototype.<method_name> = function() {
  //Custom code here
};
```

These are methods in the base class, which you are overriding.

If, in your handling of the 'event', you wish to fall back to default handling, you need to call the superclass version of the method, passing an instance of your control object, followed by the parameters you received.

Example

```
<ctrl_name>.prototype.<method_name> = function(p1, p2) {
  //Custom code here
  this.superclass.<method_name>.call(this, p1, p2 );
};
```

constructor

All controls must have a constructor with no parameters.

The constructor is a static function, with the same name as your control.

The constructor should call your control's init_class_inst() method.

Parameters:

None

Example

```
function ctrl_generic() {
  this.init_class_inst(); // initialize our control
};
```

init_class_inst()

init_class_inst()

This initializes the class, and must be called by the constructor.

The default code should be left as is - it sets up the superclass.

Use this method to

- Initialize class variables.

Parameters:

None

Returns:

None

Example

```
ctrl.init_class_inst = function() {
  this.superclass = ctrl_base.prototype;
  this.superclass.init_class_inst.call( this ); // call our superclass class initializer
  this.myVar1 = 0;
};
```

init_ctrl_inst()

init_ctrl_inst(pForm, pElem, pRowCtrl, pRowNumber)

This is called after the control has been added to the page, with the content as set by the C++ side of your component's jsGetInnerHTML() method.

This must begin by calling init_ctrl_inst() in the superclass.

Use this method to:

- Read attributes from the element.
- Read the $dataname data. <link to this.getData()>
- Fine-tune/extend your control’s layout & appearance.
- Setup event handlers. <link>

Parameters:

- **pForm** (Object): Reference to the parent form.
- **pElem** (Object): The html element the control belongs to.
- **pRowCtrl** (Object): Pointer to a complex grid control, if this control belongs to a complex grid.
- **pRowNumber** (Integer): The row number this control belongs to, if it belongs to a complex grid.

Returns:

- (Boolean): True if this control is a container of other controls.False if it is a standard control.

Example

```
ctrl.init_ctrl_inst = function( form, elem, rowCtrl, rowNumber ) {
  this.superclass.init_ctrl_inst.call( this, form, elem, rowCtrl, rowNumber ); // call our superclass init_ctr
  //TODO: Control-specific initialization goes here
  return false;
};
```

delete_class_inst()

delete_class_inst()

This method is called when the object is removed, typically when a form is closed.

You should override this function if there is some work to do prior to your control being deleted.

Make sure to end your call to this by calling the superclass version.

Implement this method to:

- Perform any custom clean-up when your control is destroyed.

Parameters:

None

Returns:

None

Example

```
ctrl.delete_class_inst = function() {
  // Custom cleanup code here.
  this.superclass.delete_class_inst.call(this); // Call superclass version to perform standard deletion proced
}
```

updateCtrl()

updateCtrl(pWhat, pRow, pCol, pMustUpdate)

This is called in when the control may need to be updated. Generally because the instance data variable has changed, ore something has occurred which necessitates an update.

You should use this method to update your control based on what has changed.

Note that there is no superclass version of this method, so you should not attempt to call the base class' implementation.

Implement this method to:

- Check the value of the $dataname data, or other variables.
- Update the inner html of the control.

The parameters passed to this method only need to be handled if your control uses a list or row as its $dataname, and you wish to optimize your update code.

Parameters:

- **pWhat** (String): Specifies which part of the data has changed. One of:"" - All of the data, if pRow & pCol are not specified. If they are, just the data specified by pRow and pCol."#COL": A single column in the $dataname list (specified by 'row' and 'col') or a row's column (specified by 'col')"#LSEL" - The line selection state of a particular row. pRow specifies the row number this pertains to."#LSEL_ALL" - The selection state of one or more rows have changed. Query the list's lstLSEL member to access the new selection state."#L" - The current line."#NCELL" - An individual cell in a (nested) list. In this case, 'row' is an array of row & column numbers. Of the form "row1,col1,row2,col2,..."
- **pRow** (Integer): If specified, the row number which has changed. (1-indexed). If 'what' is "#NCELL", this must be an array of row and col numbers. Optionally, a modifier may be added as the final array element, to change which part of the nested data is to be changed. (Currently only "#L" is supported).
- **pCol** (Integer or String): If specified, the column number (1-indexed), or name which has changed.
- **pMustUpdate** (Boolean): If true, the data has changed, and you should update your control accordingly. ***DEPRECATED***

pMustUpdate is ***deprecated*** and may be removed in a future update. pWhat will perform the same functionality (when it is "", or otherwise 'falsey').

Returns:

None

Example

```
ctrl.updateCtrl = function(pWhat, pRow, pCol, pMustUpdate) {
  // Check if the $dataname variable has changed:
  if (pWhat == "") {
    this.curData = this.getData();
    this.rebuildMyControl(); // Custom method to rebuild control.
  }
};
```

handleEvent()

handleEvent(pEvent)

This is called when an event handler which was assigned this.mEventFunction is triggered.

Implement this method to:

- Create a switch statement based on pEvent.type to handle multiple events.
- If multiple elements in your control can trigger the same kind of event (e.g. “click”), you may like to use jOmnis.getEventCurrentTarget() to determine which element was clicked.

Parameters:

- **pEvent** (Object): The standard JavaScript event object, pertaining to this event.

Returns:

None

Example

```
ctrl.handleEvent = function( pEvent ) {
  switch (pEvent.type) {
    case "click":
      this.doMyClick(); // Handle the click
      jOmnis.stopEventNow(pEvent); // Prevent further propagation of this event.
      break;
    default:
      return this.superclass.handleEvent.call( this, pEvent ); // Let the base class handle other events
  }
};
```

getCanAssign()

getCanAssign(pPropNumber)

Use this method to create a switch statement which returns true or false as to whether the passed property can be assigned to, if it’s a custom property (or a base property you wish to treat differently). Otherwise, let the superclass handle it.

Implement this method to:

- Determine whether a custom property can be assigned to.

Parameters:

- **pPropNumber** (Integer): The Omnis property number.

Returns:

- (Boolean): Whether the property can be assigned to.

Example

```
ctrl.getCanAssign = function(pPropNumber) {
  switch (pPropNumber) {
    case PROPERTIES.prop1:
      return true;
    default:
      return this.superclass.getCanAssign.call(this, pPropNumber); // Let base class handle other properties.
  }
};
```

getProperty()

getProperty(pPropNumber)

This is called when Omnis code attempts to get the value of a property of the control.

If it is a custom property (or a base property you need to use special handling for), you should catch and handle this yourself. Otherwise, let the superclass handle it.

The propNumbers of standard properties can be accessed via the eBaseProperties enumerated type.

It is good practice to call this method from your JavaScript code when you want to get the value of a property.

Implement this method to:

- Catch and handle requests for your custom properties, or properties you need to add special handling for.

Parameters:

- **pPropNumber** (Integer): The Omnis property number.

Returns:

- (Var): The value of the property.

Example

```
ctrl.getProperty = function(propNumber) {
  switch (propNumber) {
    case PROPERTIES.prop1:
      return this.mProp1;
    default:
      return this.superclass.getProperty.call(this, propNumber); // Let the base class handle other properties
  }
};
```

setProperty()

setProperty(pPropNumber, pPropValue)

This is called when Omnis code attempts to set the value of a property of the control.

If it is a custom property, you should catch and handle this yourself. Otherwise, let the superclass handle it for default properties.

The propNumbers of standard properties can be accessed via the eBaseProperties enumerated type.

It is good practice to call this method from your JavaScript code when you want to set a property.

Implement this method to:

- Catch and handle changes to your custom properties, or properties you need to add special handling for.
- Begin by calling this.getCanAssign(pPropNumber), to respect rules set therein.

Parameters:

- **pPropNumber** (Integer): The Omnis property number.
- **pPropValue** (Var): The new value for the property.

Returns:

- (Boolean): Whether the property was set successfully.

Example

```
ctrl.setProperty = function( propNumber, propValue ) {
  if (!this.getCanAssign(propNumber))
    return false;
  switch (propNumber) {
    case eBaseProperties.enabled:
      this.handleEnabled(propValue);
      return true;
  default:
      return this.superclass.setProperty.call( this, propNumber, propValue ); // call our superclass setPropert
  }
};
```

sizeChanged()

sizeChanged()

Called when the size of the control has changed (e.g. due to a screen size change).

Implement this method to:

- Add special handling for when the size of the control has changed.

Parameters:

None

Returns:

None

Example

```
ctrl.sizeChanged = function() {
  // Custom code here
};
```

fillChanged()

fillChanged()

Called when the background of the control has changed due to a property change.

Implement this method to:

- Add special handling for when the background of the control has changed.

Parameters:

None

Returns:

None

Example

```
ctrl.fillChanged = function() {
  // Custom code here
};
```

fontChanged()

fontChanged()

Called when the font changes - either family, style or size.

Implement this method to:

- Add special handling for when the control’s font has changed.

Parameters:

None

Returns:

None

Example

```
ctrl.fontChanged = function() {
  // Custom code here
};
```

enabledChanged()

enabledChanged()

Called when $enabled has changed (and also when constructing the control).

If you implement this, you will probably want to get the current enabled state. You can use this.isEnabled() to return this.

Implement this method to:

- Handle enabling/disabling of your control.

Parameters:

None

Returns:

None

Example

```
ctrl.enabledChanged = function() {
  var state = this.isEnabled();
  // Custom code here
};
```

activeChanged()

activeChanged()

Called when $active has changed (and when constructing the control).

If you implement this, you will probably want to get the current active state. You can use this.isActive() to return this.

Implement this method to:

- Handle making your control active/inactive.

Parameters:

- None

Returns:

- None

Example:

```
ctrl.activeChanged = function() {
  var state = this.isActive();
  // Custom code here
};
```

getTrueVisibility()

getTrueVisibility(pChildCtrl)

Returns the true visibility state of the control, based on parent object and own visible property.

More complex controls may need to override this. If you do, it's recommended to first call the base class' implementation as this already does the work of checking parent controls and containing subforms' visibility.

Parameters:

- **pChildControl** (Object): The child control, if this function is being called from a child control.

Returns:

- (Boolean): True only if the control is actually visible.

Example

```
ctrl.getTrueVisibility = function(pChildCtrl) {
  var visible = this.superclass.getTrueVisibility.call( this, childCtrl ); // call base class
  // If the base class thinks the control should be visible, check that no control-specific behaviour contradi
  if (visible) {
    visible = this.reallyVisible();
  }
  return visible;
};
```

visibilityChanged()

visibilityChanged()

Called when the visibility of your control may have changed (e.g. it may be on a paged pane which is no longer visible).

Typically you would call the base class' implementation of this first, then add your own logic.

If the base class returns false, but you decide to return true in your own logic, you should also set the visibility css style of your control to the new value.

Implement this method to:

- Catch when the visibility of your control changes.
- Provide extra handling to determine if the visibility of your control has actually changed.

Parameters:

None

Returns:

- (Boolean): Whether the visibility of the element differed from whether it was actually visible.

Example

```
ctrl.visibilityChanged = function()
{
  var ret = this.superclass.visibilityChanged.call(this);
  if (ret)
    this.updateLayout();
    return ret;
}
```

formBuilt()

formBuilt()

If you called registerForFormBuilt() in your control, this method will be called when the rest of the form has finished constructing.

There is no superclass implementation of this method, so do not attempt to call one.

Parameters:

None

Returns:

None

Example

```
ctrl.formBuilt = function () {
  alert("The form has finished construction!");
}
```

getErrorBorderDiv()

getErrorBorderDiv()

When a control's $errortext is assigned to, it will show the error text around the control, and add a red border. By default, this border is drawn around the control's client element.

If this does not make sense for a control, it can override this method and return the element around which the error border should be drawn.

Parameters:

None

Returns:

- (Element): The element to which the error border should be applied.

Example

```
ctrl.getErrorBorderDiv = function() {
  return this.mMainElem;
}
```

Control - Instance Methods

The following are method calls you can make on a ctrl instance (i.e. your JavaScript control).

alphaFromCssColor()

alphaFromCssColor(pCssColor)

Gets the alpha value (0-255) from the given CSS color string. 0 being fully transparent, and 255 being completely opaque.

Parameters:

- **pCssColor** (String): A CSS Color string (e.g. “#FFFF00”, “rgb(255,255,0)” or “rgba(255,255,0,0.5)”)

Returns:

- (Integer): The alpha value of the color (0 - 255).

Example

```
var alpha = this.alphaFromCssColor("rgba(155,155,208,0.5)"); // alpha becomes 128
```

callMethod()

callMethod(pMethodName, ...)

Calls a method (custom or built-in) of your control. Can be:

- A JavaScript method (e.g. ctrl_mycontrol.prototype.myMethod = function {...})
- An Omnis field method.

If there is a method in the JavaScript control with the same name as an Omnis field method, the method of the JavaScript control will be called.

Parameters:

- **pMethodName** (String): The name of the method to call.
- **...** : Any following parameters will be used as arguments when calling the specified method.

Returns:

- (Var): Return value of the method called.

Example

```
var ret = this.callMethod("myMethod","Dave",23); //Call the method "myMethod", passing "Dave" & 23 as argument
```

callMethodEx()

callMethodEx(pMethodName, pFlags, ...)

Calls a method (custom or built-in) of your control, with extra configuration flags. Can be:

- A JavaScript method (e.g. ctrl_mycontrol.prototype.myMethod = function {...})
- An Omnis field method.

If there is a method in the JavaScript control with the same name as an Omnis field method, the method of the JavaScript control will be called.

Parameters:

- **pMethodName** (String): The method name.
- **pFlags** (Integer): Flags defined from eDoMethodFlags.
- **...** : Any extra parameters are passed as arguments to the method call.

Returns:

- (Var): Return value of the method called.

Example

```
var ret = this.callMethodEx("myMethod", eDoMethodFlags.completionEvent | eDoMethodFlags.noOverlay, "Dave", 123
```

canSendEvent()

canSendEvent(pEventCode)

Returns true if the given event is enabled for this control and it can be sent to the server(/client event).

Parameters:

- **pEventCode** (Integer|String): The code for the event you wish to trigger. One of the eBaseEvent constants, or a custom event id number (declared in your component's .rc file, and referenced in its ECOmethodEvent structure - e.g. 5001).

Returns:

- (Boolean): Whether the event can be sent.

Example (Default event)

```
if (this.canSendEvent(eBaseEvent.evClick) {
    this.eventParamsAdd("pName","Ernie");
    this.eventParamsAdd("pAge", 52);
    this.eventParamsAdd("pOccupation", "Milkman");
    this.sendEvent(eBaseEvent.evClick); // Sends event with the 3 parameters defined above.
}
```

Example (Custom Event)

```
var eMyControlEvents = { "evThingOccurred": 5000, "evThingHappened": 5001}
if (this.canSendEvent(eMyControlEvents.evThingHappened) {
    this.eventParamsAdd("pName","Ernie");
    this.eventParamsAdd("pAge", 52);
    this.eventParamsAdd("pOccupation", "Milkman");
    this.sendEvent("evThingHappened"); // Sends event with the 3 parameters defined above.
}
```

doSetProperty()

doSetProperty(pPropNumber, pPropValue)

This is a wrapper for setProperty(), which instead adds the property to the current animation block, if possible (i.e. the property is supported for animations). If the property can't be animated, it will be assigned immediately.

See beginAnimations() and commitAnimations().

If you have overridden setProperty() in your control, this will also be called into at the appropriate time.

Parameters:

- **pPropNumber** (Integer|String): The Omnis property number. Either an eBaseProperties constant, or a custom value for any custom properties of your control.
- **pPropValue** (Var): The new value to assign to the property.

Returns:

- (Boolean): Success

Example

```
var formInst = this.form;
formInst.beginAnimations(1000, 4); // Begin animation block
this.doSetProperty(eBaseProperties.left, 300); // Add property change to animation block.
formInst.commitAnimations(); // Run the animations.
```

eventParamsAdd()

eventParamsAdd(pName, pValue)

Adds a parameter to the event parameter list.

Call this method for each event parameter prior to calling sendEvent, which will call the event, sending the parameters.

The parameter list is cleared once sendEvent() has been called.

Parameters:

- **pName** (String): Parameter name (must match that defined in your component's resource (.rc) file).

- **pValue** (Var): Parameter value.

Returns:

None

Example

```
this.eventParamsAdd("pName","Ernie");
this.eventParamsAdd("pAge", 52);
this.eventParamsAdd("pOccupation", "Milkman");
this.sendEvent(eBaseEvent.evClick); // Sends event with the 3 parameters defined above.
```

focus()

focus()

Sets the focus to the control's client element.

Parameters:

None

Returns:

None

Example

```
this.focus();
```

getCanAssign()

getCanAssign(pPropNumber)

Gets whether you are allowed to assign a new value to a property.

This method will typically be overridden in your JavaScript control, to handle any custom properties, but fall back to the base class' implementation for base properties.

Parameters:

- **pPropNumber** (Integer|String): The Omnis property number to check. Either an eBaseProperties constant, or a custom value for any custom properties of your control.

Returns:

- (Boolean): Whether the property can be assigned to.

Example

```
var canDisable= this.getCanAssign(eBaseProperties.enabled)
```

getClientElem()

getClientElem()

Returns the client HTML element for this control.

Parameters:

None

Returns:

- (Object): The client HTML element.

Example

```
var elem = this.getClientElem();
```

getData()

getData(pWhat, pRow, pCol, pAltDataIndex, pFmt, pFormatString)

Returns the data associated with the control (typically the control's $dataname instance variable).

All parameters are optional.

The parameters only apply if your control uses a list or row instance variable for its $dataname.

Parameters:

- **pWhat** (String): Specifies which part of the data. One of:"": Gets all of the data, if pRow & pCol are not specified. If they are, just the data specified by pRow and pCol."#LSEL": Gets the list's line selection flags."#L": Gets the list's current line.
- **pRow** (Integer): If specified, only returns data for that row in the list. (1-indexed)
- **pCol** (Integer or String): If specified, only returns data for that column in the list row. (1-indexed)
- **pAltDataIndex** (Integer): If not null, this is the data index of another instance variable to get, rather than the control's $dataname variable.
- **pFmt** (String): If specified,the format required for the return value.null means return the value unchanged."s" means return a string."sz" means return a string which is empty if the value is zero.
- **pFormatString** (String): If pFmt is s or sz, and the data is a date/time type, the date format used to format the date and time.If pFmt is s or sz, and the data is a numeric type, the numberformat used to format the number.

Returns:

- (Var): The requested data.

Example

```
// Get the whole $dataname instance variable data:
var myFullData = this.getData();
// Get the current line from the $dataname list:
var curLine = this.getData("#L");
// Get the data from column 2 in row 3 of the $dataname list:
var myListCell = this.getData("", 3, 2);
```

setData()

setData(pNewData, pWhat, pRow, pCol, pFmt, pFormatString, pUpdateThis, pIgnorePager)

Sets the value of the variable assigned to this control's $dataname.

Parameters:

- **pNewData** (Var): The new data to assign to the variable.
- **pWhat** (String): Specifies which part of the data to assign to. One of:"" - Change the data (row data if a list)."#COL" - Change the data for a list's column specified by pRow and pCol, or a row's column specified by pCol."#1COL" - Same as #COL for this method."#LSEL" - Change a list's line selection flags for the row specified by pRow."#LSEL_ALL" - Change the complete list selection array for a list."#L" - Change a list's current line.
- **pRow** (Optional) (Integer): If specified, changes the data only for that row in a list (range = 1..n)
- **pCol** (Optional) (Integer): If specified, changes the data only for that column in a list row (range = 1..n)
- **pFmt** (Optional) (String): If specified, the source format for the new data.null means keep the format of the new data"s" means the new data is a string which is to be converted to the correct data type for the instance variable if possible. "a" means the new data is always to be used, and not to be compared with the oldData (as it has been changed in place by a client method).
- **pFormatString** (Optional) (String) If pFmt is "s", the Number or Date formatting string (depending on data type) to format the new data.
- **pUpdateThis** (Optional) (Bool) If true, specifies that the control should update itself in response to this. Defaults to false.
- **pIgnorePager** (Optional) (Bool) If true, this will ignore any attached pager.

Returns:

- (Boolean): Success

Example

```
this.setData(newValue, ""); // Change the full contents of the $dataname variable to the contents of newValue.
```

getDisplayFormat()

getDisplayFormat(pForceNumber)

Gets the format string to use when formatting a number or date-time variable.

If the control's $dataname is a numeric type, this returns the number format.

If the control's $dataname is a date type, this returns the date-time format.

Parameters:

- **pForceNumber** (Boolean): If true, this will always return the number format, regardless of the control's data type.

Returns:

- (String): The required format string.

Example

```
var dateFormat = this.getDisplayFormat(false);
```

getIconUrlFromImgSrc()

getIconUrlFromImgSrc(pElem)

Gets the URL to an image from the given element's src attribute. Typically, this would be an img element.

Parameters:

- **pElem** (Object): The html element.

Returns:

- (String): The URL to the image, or an empty string if the src attribute doesn't exist.

Example

```
var imgURL = this.getIconUrlFromImgSrc(elem);
```

getIconUrlFromStyleUrl()

getIconUrlFromStyleUrl(pElem)

Returns the URL to the image specified in the element's background-image property, when specified in the format:

`"url(<url to image>)"`

Parameters:

- **pElem** (Object): The html element.

Returns:

- (String): URL to the background image.

Example

```
// elem.backgroundImage = "url(images/myImage.png)"
var imgURL = this.getIconUrlFromStyleUrl(elem);
// imgURL becomes "images/myImage.png"
```

getProperty()

getProperty(pPropNumber)

Gets the value of the given Omnis property number.

This method will typically be overridden in your JavaScript control, to handle any custom properties, but fall back to the base class' implementation for base properties.

Parameters:

- **pPropNumber** (Integer): The Omnis property number (either an eBaseProperties value, or a custom value for your control).

Returns:

- (Var): The property's value.

Example

```
var textColor = this.getProperty(eBaseProperties.textcolor);
```

getTextWidthDiv()

getTextWidthDiv()

Returns a div element, suitable for measuring text width (i.e. a div which tightly wraps its content).

Make sure to remove the div once you are finished with it, using removeTextWidthDiv().

Parameters:

- None

Returns:

- (Object): HTML div element which wraps any added text.

Example

```
var divElement = this.getTextWidthDiv();
divElement.innerHTML = "my string"; // Inject the text into the div
var stringWidth = divElement.clientWidth; //Measure the width of the div
this.removeTextWidthDiv(divElement);
```

removeTextWidthDiv()

removeTextWidthDiv(pElem)

Called to remove the div used for measuring text width (obtained using getTextWidthDiv()).

Parameters:

- **pElem** (Object): The div to remove.

Returns:

- None

Example

```
var divElement = this.getTextWidthDiv();
divElement.innerHTML = "my string"; // Inject the text into the div
var stringWidth = divElement.clientWidth; //Measure the width of the div
this.removeTextWidthDiv(divElement);
```

getTrueVisibility()

getTrueVisibility(pChildControl)

Returns true visibility state of the control, based on parent object and own visible property.

More complex controls may need to override this.

Parameters:

- **pChildControl** (Object): The child control, if this function is being called from a child control.

Returns:

- (Boolean): Whether the control is currently visible.

Example

```
var isVisible = this.getTrueVisibility(null);
```

isEnabled()

isEnabled()

Returns whether the control is enabled (based on its state, and the state of any container(s).

Parameters:

- None

Returns:

- (Boolean): Enabled state.

Example

```
var enabled = this.isEnabled();
```

removeFromTabOrder()

removeFromTabOrder()

Removes the control from the tab order.

Prevents the user from tabbing into it.

Parameters:

- None

Returns:

- None

Example

```
this.removeFromTabOrder(); // Remove this control from the tab order.
```

sendEvent()

sendEvent(pEventCode)

Sends an event to the server (or the client executed event if it's set to execute on client).

This function will send and clear the parameter list.

You would typically check canSendEvent() before calling this.

Parameters can be added to the event using eventParamsAdd().

Parameters:

- **pEventCode** (Integer): The code for the event you wish to trigger. One of the eBaseEvent constants, or a custom event name string (declared in your component's .rc file, and referenced in its ECOmethodEvent structure - e.g. "evMyEvent").

Returns:

- None

Example (Default event)

```
if (this.canSendEvent(eBaseEvent.evClick) {
  this.eventParamsAdd("pName","Ernie");
  this.eventParamsAdd("pAge", 52);
  this.eventParamsAdd("pOccupation", "Milkman");
  this.sendEvent(eBaseEvent.evClick); // Sends event with the 3 parameters defined above.
}
```

Example (Custom Event)

```
var eMyControlEvents = { "evThingOccurred": 5000, "evThingHappened": 5001}
if (this.canSendEvent(eMyControlEvents.evThingHappened) {
  this.eventParamsAdd("pName","Ernie");
  this.eventParamsAdd("pAge", 52);
  this.eventParamsAdd("pOccupation", "Milkman");
  this.sendEvent("evThingHappened"); // Sends event with the 3 parameters defined above.
}
```

setPropCli()

setPropCli(pPropNumber, pPropValue)

Sets the Omnis property of the control to the value passed.

This is a wrapper for the doSetProperty() method which will also attempt String Table translation on pPropValue, if it is a string.

If you have overridden setProperty() in your control, this will also be called into at the appropriate time.

Parameters:

- **pPropNumber** (Integer): The Omnis property number to check. Either an eBaseProperties constant, or a custom value for any custom properties of your control.
- **pPropValue** (Var): The value to assign to the property.

Returns:

- (Boolean): Success

Example

```
var success = this.setPropCli(eBaseProperties.left, 100);
success = this.setPropCli(eBaseProperties.text, "Hello"); // Will attempt to translate "Hello" using string ta
```

setProperty()

setProperty(pPropNumber, pPropValue)

Sets the given Omnis property to the given value, and performs any other necessary action required by changing this value.

This method will typically be overridden in your JavaScript control, to handle any custom properties, but fall back to the base class' implementation for base properties.

Parameters:

- **pPropNumber** (Integer|String): The Omnis property number to check. Either an eBaseProperties constant, or a custom value for any custom properties of your control.
- **pPropValue** (Var): The value to assign to the property.

Returns:

- (Boolean): Success

Example

```
var success = this.setProperty(eBaseProperties.fontSize, 12); // Set the font size (handled automatically)
// Set a custom property (must be handled by overridden setProperty() method in your control):
success = this.setProperty(PROPERTIES.iconSize, "100px");
```

setPropertyBorder()

setPropertyBorder(pElem, pValue, pDefault)

Applies the given border style to an element.

Parameters:

- **pElem** (Object): The html element.
- **pValue** (Integer): The border style - one of the eBaseBorder enumerated types.
- **pDefault** (Optional) (Object): The eBaseBorder style to use whenpValue is eBaseBorder.

Returns:

- True

Example

```
this.setPropertyBorder(elem, eBaseBorder.Plain);
```

setupIcon()

setupIcon(pUrl)

Gets an icon url from the format sent by Omnis, into a useable and appropriate URL.

Replaces the icon library placeholder ("%L") in the supplied icon URL.

If more than one icon is available, chooses the best icon to use based on the client resolution.

Parameters:

- **pUrl** (String): The icon URL, in the format sent by Omnis.

Returns:

- (String): URL to icon file.

Example

```
// data-icon is: "icons/lib/%l/033333n16.png;033333n16_15x.png;033333n16_2x.png"
var iconAttrib = client_elem.getAttribute("data-icon");
var iconUrl = this.setupIcon(iconAttrib);
// On a retina device, iconUrl will become "icons/lib/myLibrary/033333n16_2x.png"
```

update()

update(pWhat, pRow, pCol)

Call this method to cause your control to be updated.

The update may not be called immediately, but will be flagged to do so at the next available opportunity.

This will end by calling into your control's updateCtrl() method, if necessary.

The parameters only really apply if your control uses a list to define its structure, and you want to optimize your handling of updates. If it does not, you can safely pass no parameters.

Parameters:

- **pWhat** (Optional) (String): Specifies which part of the data has changed. One of:
  "": All of the data, if pRow & pCol are not specified. If they are, just the data specified by pRow and pCol.
  "#COL": A single column in the $dataname list (specified by 'row' and 'col') or a row's column (specified by 'col')
  "#LSEL": The line selection flags.
  "#LSEL_ALL": All lines in the $dataname list have been selected.
  "#L": The current line.
  "#NCELL": An individual cell in a (nested) list. In this case, 'row' is an array of row & column numbers. Of the form "row1,col1,row2,col2,..."
- **pRow** (Integer): If specified, the row number which has changed. (1-indexed). If 'what' is "#NCELL", this must be an array of row and col numbers. Optionally, a modifier may be added as the final array element, to change which part of the nested data is to be changed. (Currently only "#L" is supported).
- **pCol** (Integer or String): If specified, the column number (1-indexed), or name which has changed.

Returns:

- None

Example

```
this.update("COL", 3, 2); // Call update, specifying that just the data in col 2 of the 3rd row has changed.
```

visibilityChanged()

visibilityChanged()

Call this if you wish to check whether your control's visibility has changed (e.g. perhaps it's on a paged pane which is no longer visible).

This checks the visibility of your client element against the value of getTrueVisibility().

This will also set your client element's visibility style to the new value.

Parameters:

- None

Returns:

- (Boolean): Whether the visibility of the element differed from whether it was actually visible.

Example

```
if (this.visibilityChanged()) {
  this.rebuildMyControl();
}
```

isActive()

isActive()

Returns whether the control is active (based on its state, and the state of any container(s)).

Parameters:

- None

Returns:

- (Boolean): Active state.

Example

```
var active = this.isActive();
```

getTheme()

getTheme()

Returns the theme object associated with the Omnis instance. Once you have the Theme instance, there are a variety of methods you can use, documented in the Theme Instance Methods section.

Parameters:

- None

Returns:

- (Object): Theme object.

Example

```
const theme = this.getTheme();
```

resolveDefaultColor()

resolveDefaultColor(color, attribute)

Returns the passed color into a concrete color. If it's kColorDefault, turns it into the appropriate theme color constant. Otherwise, returns the color unchanged. Especially useful for calculating the correct text color when on a theme color background – see getTextColorString().

Parameters:

- **color** (Number): The color to normalise.
- **Attribute** (ColorAttribute): The name of a key within the control's DefaultColors object from which to get the default color.

Returns:

- (Number): The passed color, if it was not kColorDefault, else the matched theme color constant from DefaultColors (or 0 if not found).

Example

```
const backColor = this.resolveDefaultColor(backgroundColor, BaseDefaultColorAttributes.BACKGROUND);
```

Control - Properties

.form

Type: Form

The control's containing Form instance.

.mEventFunction

Type: function

You should set this as the handler for your control's HTML events.

Then, when the event is triggered it will pass through the Omnis JavaScript framework, and will call into your class' handleEvent method, where you should handle the event.

Example

```
ctrl.init_ctrl_inst = function(form, elem, rowCtrl, rowNumber)
{
...
client_elem.onclick = this.mEventFunction;
client_elem.onblur = this.mEventFunction;
...
};
```

.objName

Type: Character

The name of the control.

(Typically something like js1_button_1059)

.objNumber

Type: Integer

This is the Object Number of the control. A unique identifier within the form.

allowDefaultDropHandling

Type: Boolean

If true, the browser's default drag & drop drop handling will not be prevented for this control.

Defaults to false.

baseClassName

Type: String

Controls can override this to provide a space-separated list of css classes which should always be applied to the Client Element.

Set before calling super init_ctrl_inst in order to automatically apply to Client Element initially.

DefaultColors

Type: Object

This object contains theme constant values to be used for default theme colors for when a color property is set to kColorDefault. Use with BaseDefaultColorAttributes (found in Constants of the JavaScriptControl Reference) to access the correct member.

See Theme Instance Methods getColorString() and getTextColorString() for examples of how these are used.

mAcceptsReturn

Type: Boolean

If this is set to true, pressing the Enter key while the control has focus will not trigger the form's okkeyobject (if there is one). This should be set if the control performs an action when Enter is pressed.

mAcceptsEscape

Type: Boolean

If this is set to true, pressing the Escape key while the control has focus will not trigger the form's cancelkeyobject (if there is one). This should be set if the control performs an action when Escape is pressed.

Form

beginAnimations()

beginAnimations(pDuration, pCurve)

Begins an animation block for the form.

Individual animations are added using your control's doSetProperty() method.

The animation block is run by calling commitAnimations().

Parameters:

- **pDuration** (Integer): The animation duration, in milliseconds.
- **pCurve** (Integer): The animation curve - One of:
  0 for "ease-in-out"
  1 for "ease-in"
  2 for "ease-out"
  3 for "linear"
  4 for "ease"

Returns:

- None

Example

```
var formInst = this.form;
formInst.beginAnimations(1000, 4); // Begin animation block
this.doSetProperty(eBaseProperties.left, 300); // Add property change to animation block.
formInst.commitAnimations(); // Run the animations.
```

callMethod()

callMethod(pMethodName, ...)

Calls a form’s instance method.

Can be client or server executed.

Parameters:

- **pMethodName** (String): The name of the method to call.
- **...** : Any extra parameters will be passed as arguments to the method call.

Returns:

- (Var): The method’s return value (If it’s a client method - server methods are called asynchronously, and return into another client method).

Example

```
var ret = formInst.callMethod("$echo", "Hello"); // Calls the method $echo passing "Hello" as its first parame
```

callMethodEx()

callMethodEx(pMethodName, pFlags, ...)

Calls a form instance method

Can be a server or client method.

Parameters:

- **pMethodName** (String): The name of the method to call.
- **pFlags** (Interface): Bit mask of eDoMethodFlags constants.
- **...** : Any extra parameters will be passed as arguments to the method call.

Returns:

- (Var): The method’s return value (If it’s a client method - server methods are called asynchronously, and return into another client method)

Example

```
var ret = formInst.callMethodEx("$echo", eDoMethodFlags.clientOnly | eDoMethodFlags.noOverlay, "Hello");
// Calls the form method $echo, only if it's a client-executed method, and prevents the overlay from displayin
```

commitAnimations()

commitAnimations()

Commit an animation block for the form.

Note that nothing will occur until control returns to the browser, which then applies the CSS transitions.

Animation blocks are initiated using beginAnimations() and individual animations added using a Control’s doSetProperty() method.

Parameters:

- None

Returns:

- None

Example

```
var formInst = this.form;
formInst.beginAnimations(1000, 4); // Begin animation block
this.doSetProperty(eBaseProperties.left, 300); // Add property change to animation block.
formInst.commitAnimations(); // Run the animations.
```

findChild()

findChild(pObjNumber, pRowNumber)

Gets a control on the form by its Object Number.

Parameters:

- **pObjNumber** (Integer): The Object Number of the control to find.
- **pRowNumber** (Integer): If non-zero the object belongs to this row inside a complex grid.

Returns:

- (Object): The Control (or null if not found).

Example

```
var ctrl = formInst.findChild(3,0) ;
```

findChildByName()

findChildByName(pObjName)

Gets a Control instance by its Object Name.

Parameters:

- **pObjName** (String): The Object Name of the control to find.

Returns:

- (Object): The Control instance, or null if not found.

Example

```
var ctrl = formInst.findChildByName("jsform_button_1056");
```

getData()

getData(pDataIndex, pWhat, pRow, pCol, pFmt, pFormatString)

Returns the data from one of the form's instance variables.

Parameters:

- **pDataIndex** (Integer): The data index to the instance variable. See getDataIndex()
- **pWhat** (String): Specifies which part of the data to get. One of:
  "" - Return all of the data.
  "#COL" - Return data for a list's column specified by pRow and pCol, or a row's column specified by pCol.
  "#1COL" - Same as #COL for this method.
  "#LSEL" - Return a list's line selection flags (pRow can specify the row).
  "#LSEL_ALL" - Return the list selection array for a list.
  "#L" - Return a list's current line.
  "#LCNT" - Return a list's line count.
  "#CCNT" - Return a list's column count.
- **pRow** (Optional) (Integer): If specified, returns the data only for that row in a list (range = 1..n)
- **pCol** (Optional) (Integer): If specified, returns the data only for that column in a list row (range = 1..n)
- **pFmt** (Optional) (String): If specified,the format required for the return value.
  null means return the value unchanged.
  s means return a string.
  sz means return a string which is empty if the value is zero.
- **pFormatString** (Optional) (String) If pFmt is "s" or "sz", the Number or Date formatting string (depending on data type).

Returns:

- (Var): The requested data.

Example

```
// Get the value of the second col from iRow, in a custom date format:
var bday = formInst.getData(formInst.getDataIndex("iRow",
"#COL", 1, 2, "s", "D/M/y");
```

getDataIndex()

getDataIndex(pDataName, pCtrl, pMapControlToData)

Returns the index to an instance data variable.

Also allows you to map the data variable to the control.

Parameters:

- **pDataName** (String): The name of the instance variable.
  Can also be formatted as “variable.column” to refer to an individual column in a row/list variable. The column can be specified as a number, a number prefixed with “C”, or a column name. E.g. “iRowVar.C3”
- **pCtrl** (Object): A control to be associated with the data index if pMapControlToData is true (or omitted). Pass as null if you don’t want to map the data to a control.
- **pMapControlToData** (Optional) (Boolean): If true, associate the control with the data index. If omitted, defaults to true.

Returns:

- (Integer): Data index for the variable.

Example

```
var dataIndex = formInst.getDataIndex("iList", null, false);
```

getNextPrevTabObject()

getNextPrevTabObject(pStartObj, pIsPrev, pNoWrap)

Gets the next or previous active Control in the tab order.

Parameters:

- **pStartObj** (Object): An Omnis Control instance, relative to which next or previous control is required.
- **pIsPrev** (Boolean): True if previous control is required, false to get the next control.
- **pNoWrap** (Boolean): (Optional) If true and the start object is the final object in the tab order, the function will return null.

Returns:

- (Object): The next (or previous) active Control instance in the tab order.

Example

```
var nextCtrl = formInst.getNextPrevTabObject(this, false);
```

registerForFormBuilt()

registerForFormBuilt(pCtrl)

Call this from your control's init_ctrl_inst() method, if you wish to be notified when the form has finished building.

Your control's formBuilt() method will be called when the form has finished constructing everything.

Parameters:

- **pCtrl** (Object): An Omnis Control instance, which you wish to be notified when the form has finished constructing.

Returns:

- None

Example

```
var formInst = jOmnis.getOmnisForm(0,0);
formInst.registerForFormBuilt(this);
```

setData()

setData(pDataIndex, pNewData, pWhat, pRow, pCol, pCanRecord, pFmt, pFormatString)

Assigns data to a form instance variable.

If the instance variable is mapped to a control, that control's updateCtrl() method will also be called, after setting the data, with the values of pWhat, pRow & pCol set here.

Parameters:

- **pDataIndex** (Integer): The data index to the instance variable. See getDataIndex()
- **pNewData** (Var): The new data to assign to the variable.
- **pWhat** (String): Specifies which part of the data to assign to. One of:
  "" - Change the data (row data if a list).
  "#COL" - Change the data for a list's column specified by pRow and pCol, or a row's column specified by pCol.
  "#1COL" - Same as #COL for this method.
  "#LSEL" - Change a list's line selection flags for the row specified by pRow.
  "#LSEL_ALL" - Change the complete list selection array for a list.
  "#L" - Change a list's current line.
- **pRow** (Optional) (Integer): If specified, changes the data only for that row in a list (range = 1..n)
- **pCol** (Optional) (Integer): If specified, changes the data only for that column in a list row (range = 1..n)
- **pCanRecord** (Optional) (Boolean): If true, we can record the change to be sent to the server the next time we communicate. (Defaults to true)
- **pFmt** (Optional) (String): If specified, the source format for the new data.
  null means keep the format of the new data
  "s" means the new data is a string which is to be converted to the correct data type for the instance variable if possible.
  "a" means the new data is always to be used, and not to be compared with the oldData (as it has been changed in place by a client method).

- **pFormatString** (Optional) (String) If pFmt is “s”, the Number or Date formatting string (depending on data type) to format the new data.

Returns:

- (Boolean): Success

Example

```
// Set a list's current line to 5:
formInst.setData(dataIndex, 5, "#L");
// Set the value of the 4th column in the 3rd row to be "Bert":
formInst.setData(dataIndex, "Bert", "#COL", 3, 4);
```

jIcons

This is a global object which has replaced some of the methods within jOmnis and added further functionality to work with icons.

getCheckedImage()

getCheckedImage(normalUrl)

Gets the checked image URL corresponding to a normal image URL. According to icon set naming rules. *(Previously a jOmnis method)*

Parameters:

- **normalUrl** (String): The url to the normal (unchecked) image.

Returns:

- (String): The url to the checked version of the image.

Example

```
var checkedIcon = jIcons.getCheckedImage("icons/myset/myCheckBox_90001_32x32_2x.png");
//checkedIcon becomes "icons/myset/myCheckBox_90001_32x32c_2x.png"
```

getIconBackSize()

getIconBackSize(iconUrl)

Returns the dimensions of the image, as a string suitable to be applied to an element’s background-size style value. I.e. “<width>px <height>px”

Parameters:

- **iconUrl** (String): The URL to the image.

Returns:

- (String): A string suitable to be applied to an element’s background-size style value. I.e. “<width>px <height>px”

Example

```
myImg.style.backgroundSize = jIcons.getIconBackSize("icons/myset/myIcon_90000_32x32.png");
```

getIconSize()

getIconSize(iconUrl, cssStyle, isSvg)

Returns either an object with width and height members indicating the size of the icon for the url, or a CSS style to set the background size.

If pCssStyle is false, and the size is not available (such as for a stand-alone page not in an ICON set) returns an empty object.

The icon name must conform to Omnis Icon Set naming syntax.

Parameters:

- **iconUrl** (String): The url to the icon image.
- **cssStyle** (Boolean): Whether to return a CSS style string. e.g. “background-size: 50px 80px;”
- **isSvg** (Boolean): Either null (or omitted) meaning detect if SVG from iconUrl, or Boolean indicating if the image is SVG

Returns:

- (String or Object): Either an object with width and height members indicating the size of the icon for the url or a CSS style to set the background size. If the size is not available (such as for a stand-alone page not in an ICON set), return an empty object:

If pCssStyle=true, returns a CSS style string. e.g. “background-size: 50px 80px;”
If pCssStyle=false, returns a JavaScript Object containing width and height properties.

Example

```
const iconWidth = jIcons.getIconSize("icons/myset/myIcon_90000_32x32.png", false).width;
```

replaceOrAddToElem()

replaceOrAddToElem(parentElem, iconUrl, loadFunc, sizeObj, alwaysCreate, needWrapper)

Replaces if present, or adds if absent, an icon to an element.

Parameters:

- **parentElem** (HTMLElement): The parent element to which the specified icon is to be added; if the parent already contains an icon it is replaced,otherwise this method adds the icon after any existing children in the parent.
- **iconUrl** (String): The URL to the image.
- **loadFunc** (Function) (Optional): The function to call when the image has loaded (null or omitted if onload is not required).

- **sizeObj** (Object) (Optional): If not null, a size object to override the size specified by the **iconUrl**.
- **alwaysCreate** (Boolean) (Optional): If true, always create and append (so we do not replace an existing element)
- **needWrapper** (Boolean) (Optional): If true, create a wrapper element for SVG icons. The wrapper allows the icon to have a badge

Returns:

- (HTMLElement): The element being used for the icon or null if there is no icon or no useful icon URL

Example

```
const myElement = document.getElementById("myElement"); // The element to append the icon to.
const sizeObj = {width: 16, height: 16, viewBox: "0 0 24 24"}; // The size object to give a width and height o
jIcons.replaceOrAddToElem(myElement, "icons/myicon.svg?", ()=>{window.alert("Icon loaded");}, sizeObj, false);
```

iconUrl()

iconUrl(imageElem)

Returns the icon URL for the supplied element.

Parameters:

- imageElem (HTMLElement): The img or svg element

Returns:

- (String) The icon URL or null if the element does not have a URL.

Example:

```
const iconElem = this.clientElem.querySelector("img,svg");
const iconUrl = jIcons.iconUrl(iconUrl);
```

setImageUrl()

setImageUrl(imageElem, imageUrl)

Replaces the image URL in the supplied image element (svg or img). This is used to switch between checked and normal images, so the type of the image supplied is the same as the current image (img, svg or themed svg).

Parameters:

- imageElem (HTMLElement): The image element
- imageUrl (String): The normal image URL

Returns:

- None

Example:

```
const iconElem = this.clientElem.querySelector("img,svg");
const checkedImage =
jIcons.getCheckedImage("icons/myset/myCheckBox_90001_32x32_2x.png");
jIcons.setImageUrl(iconElem, checkedImage);
```

jOmnis

addClass()

addClass(pElem, pClassName, addSuffix)

Adds the passed CSS class name(s) to the specified element, if it doesn't already have it.

Parameters:

- **pElem** (Object): The html element.
- **pClassName** (String): A space-separated list of CSS class names to add.
- **addSuffix** (String) (Optional): A suffix to add to all 'classes' added to the element.

Returns:

- None

Example

```
jOmnis.addClass(elem, "myClass myOtherClass");
```

removeClass()

removeClass(pElem, pClassName, withSuffix)

Removes the passed CSS class name(s) from the element.

Parameters:

- **pElem** (Object): The html element.
- **pClassName** (String): A space-separated list of CSS class names to remove.
- **withSuffix** (String) (Optional): A suffix to add to all 'classes' to remove.

Returns:

- None

Example

```
jOmnis.removeClass(elem, "myClass myOtherClass");
```

hasClass()

hasClass(pElem, pClassName)

Checks whether the passed element has the specified CSS class (amongst all of its CSS classes).

Parameters:

- **pElem** (Object): The html element.
- **pClassName** (String): The CSS class name to search for.

Returns:

- (Boolean): True if the element has the specified CSS class applied.

Example

```
var hasMyClass = jOmnis.hasClass(client_elem, "myclass");
```

addStyleSheet()

addStyleSheet(pCssText)

Adds a CSS style sheet to the page.

Parameters:

- **pCssText** (String): The CSS text to add as a style sheet.

Returns:

- (Object): The style sheet added - this must be removed with removeStyleSheet() before replacing it due to a property change etc.

Example

```
var myStyleSheet = jOmnis.addStyleSheet(".myClass{ color: #FF0000;}");
```

removeStyleSheet()

removeStyleSheet(pStyleSheet)

Removes a CSS style sheet from the page.

Parameters:

- **pStyleSheet** (Object): The style sheet returned by addStyleSheet().

Returns:

- None

Example

```
var myStyleSheet = jOmnis.addStyleSheet(".myClass{ color: #FF0000;}");
jOmnis.removeStyleSheet(myStyleSheet);
```

registerStyleSheetObject()

registerStyleSheetObject(pObject)

Registers the passed object (generally a control) to be notified when stylesheets are updated dynamically (using addStyleSheet etc).

The object must implement a method called “styleSheetsUpdated”, which will be called when the stylesheets have updated.

Parameters:

- **pObject** (Object): The object to notify whenever

Returns:

- None

Example

```
ctrl.init_ctrl_inst = function() {
    ...
    jOmnis.registerStyleSheetObject(this);
    ...
}

ctrl.styleSheetsUpdated = function() {
    // React to stylesheets being loaded.
}
```

unregisterStyleSheetObject

unregisterStyleSheetObject(pObject)

Unregisters an object (generally a control) from receiving notifications when stylesheets are updated dynamically.

Parameters:

- **pObject** (Object): The object to remove from the list of objects to notify when stylesheets are updated.

Returns:

- None

Example

```
jOmnis.unregisterStyleSheetObject(this);
```

loadResource()

loadResource(pURL)

Loads a resource (.js or .css file) dynamically.

Returns a Promise, which will be fulfilled when the resource loads.

(Handles multiple calls to load the same resource)

If you are loading multiple resources, you are advised to instead use loadResources()

Parameters:

- **pURL** (String): A URL to the resource to load. Should be on the same Domain unless you've configured CORS.

Returns:

- (Promise): A Promise, which will be fulfilled once the resource loads.

Example

```
function onSuccess() {
  // Handle successful load of resource
}
function onFail() {
  // Handle failure to load resource
}
jOmnis.loadResource("scripts/newscript.js").then(onSuccess).catch(onFail);
```

loadResources()

loadResources(pURLs, pOnSuccess, pOnFail)

Asynchronously loads multiple resources (css/js files) in order. Then calls the provided callback function once complete.

Parameters:

- **pURLs** (String[]): An Array of URL strings to the resource files.
- **pOnSuccess** (function): Function to call when all resources have been loaded.
- **pOnFail** (function): Function to call if an error occurs with any resource.

Returns:

- None

Example

```
jOmnis.loadResources([ "css/mycss.css", "scripts/myscript.js"], resourcesReady, resourcesLoadFailure);
```

addSwipeListener()

addSwipeListener(pElem, pStartTouchOK, pMoveHandler, pEndListener, pAnimationDuration)

Adds a swipe listener to handle touch events swiping horizontally across a control.

Parameters:

- **pElem** (Object): The html element to which the listener is to be attached.
- **pStartTouchOK** (Function): Callback function called when a touchStart event occurs. This function should return true if swipe processing can start for the event. The callback function receives the event object as a parameter.
- **pMoveHandler** (Function): Callback function to move the content while the swipe is in progress. The function is passed the x offset from the starting position and the duration for any animation to move the contents (in milliseconds).
- **pEndListener** (Function): Called when the swipe has ended. The function is passed an object containing the original element, the final offset and a cancelled flag.
- **pAnimationDuration** (Integer): The duration, in milliseconds, of any content movement that is not to occur instantaneously.

Returns:

- None

Example

```
var ctrl = this;
var startTouchFunc = function() { return true; };
var moveHandlerFunc = function(dx, duration) {ctrl.scrollToOffset(startOffset + dx, duration, false);};
var endListenerFunc = function(obj) {alert("Swipe complete");};
jOmnis.addSwipeListener(elem, startTouchfunc, moveHandlerFunc, endListenerFunc, 500);
```

animate()

animate(pElem, pProperties, pTime, pTransitionEndedHandler)

Animates multiple CSS properties of an element, over the given time. Uses CSS transitions to animate.

Parameters:

- **pElem** (Object): The html element to which the animation will be applied.
- **pProperties** (Object): A JavaScript object containing CSS property names & values. e.g: {left: “10px”, width: “100px”}
- **pTime** (Integer): Number of milliseconds the animation should occur over.
- **pTransitionEndedHandler** (Function) If provided, a callback function to call when the transition completes.

Returns:

- None

Example

```
var props = {left: "10px"; width: "100px"};
jOmnis.animate(elem, props, 1000, function() {
  alert("Animation complete!");
});
```

clearAnimations()

clearAnimations(pElem, pTransitionEndedHandler)

Removes any animations from the passed element.

Parameters:

- **pElem** (Object): The html element.
- **pTransitionEndedHandler** (Function) (Optional): The transistionEnded handler function to remove. If not provided, the global 'transitionEnd' function will be used.

Returns:

- None

Example

```
jOmnis.clearAnimations(elem, this.myAnimationEndedListener);
```

applyAlphaValueToColorPair()

applyAlphaValueToColorPair(pColorPair, pAlphaValue)

DEPRECATED in 10.2

Color Pairs are no longer needed (all browsers now support alpha), and they do not support Themed Colors. The functions in Theme Instance Methods should be used instead.

Applies a new alpha value to a Color Pair.

Parameters:

- **pColorPair** (String): A pair of color strings, separated by ";" (semicolon). The first for non-alpha, the second for alpha.
- **pAlphaValue** (Integer): The new alpha value, 0-255.

Returns:

- (String): The updated Color Pair.

Example

```
var colorPair = jOmnis.applyAlphaValueToColorPair("rgb(255,0,0);rgba(255,0,0,0.6)", 0.5);
```

applyColorValueToColorPair()

applyColorValueToColorPair(pColorPair, pColorValue)

DEPRECATED in 10.2

Color Pairs are no longer needed (all browsers now support alpha), and they do not support Themed Colors. The functions in Theme Instance Methods should be used instead.

Applies a new color value to both components of a Color Pair.

Parameters:

- **pColorPair** (String): A pair of color strings, separated by ";" (semicolon). The first for non-alpha, the second for alpha.
- **pColorValue** (Integer): The new color value, in Omnis rgb format.

Returns:

- (String): The updated Color Pair.

Example

```
var colorPair = jOmnis.applyColorValueToColorPair("rgb(255,0,0);rgba(255,0,0,0.6)", 65280); //Make the red col
```

disableTextSelection()

disableTextSelection(pElem, pReEnable, pReEnableValue)

Disables/re-enables the ability to select text in an element.

Parameters:

- **pElem** (Object): The html element.
- **pReEnable** (Boolean): true if text selection is to be re-enabled. false to disable text selection.
- **pReEnableValue** (String): The new value used to re-enable. One of the CSS user-select values: none, text, all or element.

Returns:

- (String): the old text selection value.

Example

```
var oldSelection = jOmnis.disableTextSelection(elem, false); //Disable all selection.
jOmnis.disableTextSelection(elem, true, "text"); //Re-enable text selection.
```

extractColor()

extractColor(pCololorPair)

DEPRECATED in 10.2

Color Pairs are no longer needed (all browsers now support alpha), and they do not support Themed Colors. The theme's getColorString() and getTextColorString() should be used instead.

Extracts the color to use from a Color Pair, according to whether the browser supports RGBA

Parameters:

- **pColorPair** (String): A pair of color strings, separated by ";" (semicolon). The first for non-alpha, the second for alpha.

If you want the color to be completely transparent, pass "0.00" for the alpha component. If the function finds this, it maps to "transparent", for better cross-browser support.

Returns:

- (String): The color string appropriate for the client's browser.

Example

```
var colorString = jOmnis.extractColor("rgb(255,0,0);rgba(255,0,0,0.6)");
```

extractSolidColor()

extractSolidColor(pColorPair)

DEPRECATED in 10.2

Color Pairs are no longer needed (all browsers now support alpha), and they do not support Themed Colors. The theme's getColorString() and getTextColorString() should be used instead.

Extracts the solid color from a Color Pair, as a numeric color string prefixed with #.

Parameters:

- **pColorPair** (String): A pair of color strings, separated by ";" (semicolon). The first for non-alpha, the second for alpha.

Returns:

- (String): The solid (first) color string, converted to #FFFFFF format.

Example

```
var colorString = jOmnis.extractSolidColor("rgb(255,0,0);rgba(255,0,0,0.6)");
```

getAttributeAsBool()

getAttributeAsBool(pElem, pAttributeName, pDefaultValue)

Gets an element’s attribute value as a Boolean.

Parameters:

- **pElem** (Object): The html element.
- **pAttributeName** (String): The name of the attribute to query (It should have a numeric value).
- **pDefaultValue** (Var): The value to return if the attribute doesn’t exist.

Returns:

- (Boolean): false if the attribute=0, otherwise returns true. (or pDefaultValue if the attribute doesn’t exist)

Example

```
var myBool = jOmnis.getAttributeAsBool(elem, "data-custom", false);
```

getAttributeList()

getAttributeList(pElem, pAttributeName, pPrimarySeperator)

Initializes a simple array variable from a comma-separated list in an attribute. The comma-separated list may include a primary separator for a “two-dimensional” array.

Parameters:

- **pElem** (Object): The html element.
- **pAttributeName** (String): The attribute name. The value of this attribute should be a comma-separated list.
- **pPrimarySeperator** (Optional) (String): A separator character used for two-dimensional arrays (see example below).

Returns:

- (Array): An array containing the values from the attribute’s comma-separated list.

Example

```
var myArray = jOmnis.getAttributeList(elem, "data-mylist", "~");
/* Where data-mylist = "0,12,13~2,14,5~9,100,23"
This returns a "2D" array - the first element of the array is itself an array containing 3 values -
0, 12 & 13. The second element is an array containing 2, 14 & 5, and so on...*/
```

getCheckedImage()

getCheckedImage(pNormalUrl)

REMOVED in 10.2

Icon related methods have been moved to jIcons. See getCheckedImage().

Gets the checked image URL corresponding to a normal image URL. According to icon set naming rules.

Parameters:

- **pNormalUrl** (String): The url to the normal (unchecked) image.

Returns:

- (String): The url to the checked version of the image.

Example

```
const checkedIcon = jIcons.getCheckedImage("icons/myset/myCheckBox_90001_32x32_2x.png");
//checkedIcon becomes "icons/myset/myCheckBox_90001_32x32c_2x.png"
```

getClientLocale()

getClientLocale()

Gets the client's locale information.

Parameters:

- None

Returns:

- (String): The client's locale string, language and country code, as defined by ISO 639 and 3166.

Example

```
var locString = jOmnis.getClientLocale();
```

cToChar()

cToChar(pValue)

Converts a value to a string, obeying Omnis' rules (e.g. using locale's dp character).

This is the best way to turn an omnis_date type into a string.

Parameters:

- **pValue** (*): The value to convert to a string.

Returns:

- (String): A String conversion of the passed value.

Example

```
var omDate = new omnis_date(new Date());
var dateString = jOmnis.cToChar(omDate);
```

rgbFromCssColor()

rgbFromCssColor(pCssColor)

Gets the Omnis rgb color value of a CSS color string.

Parameters:

- **pCssColor** (String): A CSS Color string (e.g. "#FFFF00", "rgb(255,255,0)" or "rgba(255,255,0,0.5)")

Returns:

- (Integer): An Omnis rgb color integer.

Example

```
var rgb = this.rgbFromCssColor("#0000FF"); //rgb becomes 16711680
```

getColorString()

getColorString(pRGB)

DEPRECATED in 10.2

Themes require the use of various constants and the functions in Theme Instance Methods have been developed to handle these. The theme's instance methods getColorString() and getTextColorString() should be used instead.

Gets a color value, suitable for use with styles.

Parameters:

- **pRGB** (Integer): An Omnis rgb color value.

Returns:

- (String): "rgb()" formatted color string.

Example

```
var blueString = jOmnis.getColorString(16711680);
```

getColorStringWithAlpha()

getColorStringWithAlpha(pRGB, pAlpha)

DEPRECATED in 10.2

Themes require the use of various constants and the functions in Theme Instance Methods have been developed to handle these. The theme's instance methods getColorString() and getTextColorString() should be used instead.

Gets a color value, suitable for use with styles, including alpha when the browser supports it.

Parameters:

- **pRGB** (Integer): Omnis rgb color value.
- **pAlpha** (Integer): The alpha value (0-255).

Returns:

- (String): "rgba()" formatted color string. Or "rgb()" formatted if the browser does not support RGBA.

Example

```
var blueStringAlpha = jOmnis.getColorStringWithAlpha(16711680,100);
```

getColorStringForElement()

getColorStringForElement(parentElem, className, attrib, colorPair)

Adds a temporary hidden element with the passed className as a child of parentElem. Uses this to create a color or colorPair string from the computed CSS rules which would be applied to the element. The hidden element is removed again once the color has been calculated.

Parameters:

- **parentElem** (HTMLElement): The parent element to add the temporary element to.
- **className** (String): The class name to apply to the element.
- **attrib** (String): The style attribute (e.g. backgroundColor) whose colour to query.
- **colorPair** (Boolean): If true, returns a colorPair string. Otherwise just a solid color string.

Returns:

- (String): A color or color pair string. E.g. rgb(125,125,125) or rgb(125,125,125);rgba(125,125,125,0.5)

Example

```
var backgroundColor = jOmnis.getColorStringForElement(this.clientElem, "myclass", "backgroundColor", true);
```

getEventCurrentTarget()

getEventCurrentTarget(pEvent)

A browser-agnostic function to return the current target of an event object. This may not be the original target when events are sent to multiple elements.

Parameters:

- **pEvent** (Object): The event object. (Standard JavaScript event object)

Returns:

- (Object): The element which is currently handling the event.

Example

```
var evCurrTarget= jOmnis.getEventCurrentTarget(event);
```

getEventTarget()

getEventTarget(pEvent)

A browser-agnostic function to return the target of an event object. This may not be the current target when events are sent to multiple elements.

Parameters:

- **pEvent** (Object): The event object. (Standard JavaScript event object)

Returns:

- (Object): The element which triggered the event.

Example

```
var evTarget = jOmnis.getEventTarget(event);
```

getIconBackSize()

getIconBackSize(pUrl)

REMOVED in 10.2

Icon related methods have been moved to jIcons. See getIconBackSize().

Returns the dimensions of the image, as a string suitable to be applied to an element's background-size style value. I.e. "<width>px <height>px"

Parameters:

- **pUrl** (String): The URL to the image.

Returns:

- (String): A string suitable to be applied to an element's background-size style value. I.e. "<width>px <height>px"

Example

```
myImg.style.backgroundSize = jOmnis.getIconBackSize("icons/myset/myIcon_90000_32x32.png");
```

getIconSize()

getIconSize(pUrl, pCssStyle)

REMOVED in 10.2

Icon related methods have been moved to jIcons. See getIconSize().

Returns either an object with width and height members indicating the size of the icon for the url, or a CSS style to set the background size.

If pCssStyle is false, and the size is not available (such as for a stand-alone page not in an ICON set) returns an empty object.

The icon name must conform to Omnis Icon Set naming syntax.

Parameters:

- **pUrl** (String): The url to the icon image.
- **pCssStyle** (Boolean): Whether to return a CSS style string. e.g. "background-size: 50px 80px;"

Returns:

- (String or Object): Depends on value of pCssStyle:

If pCssStyle=true, returns a CSS style string. e.g. "background-size: 50px 80px;"
If pCssStyle=false, returns a JavaScript Object containing width and height properties.

Example

```
var iconWidth = jOmnis.getIconsize("icons/myset/myIcon_90000_32x32.png", false).width;
```

getOmnisForm()

getOmnisForm(pInstIndex, pFormIndex)

Gets a reference to an Omnis Form.

Parameters:

- **pInstIndex** (Integer): The local index of the Omnis Instance (zero-based).
- **pFormIndex** (Integer): The local index of the Omnis Form (zero-based).

Returns:

- (Omnis Form): Omnis Form object.

Example

```
var omForm1 = jOmnis.getOmnisForm(0,0);
```

getOmnisFormControl()

getOmnisFormControl(pInstIndex, pFormIndex, pObjNumber, pObjRowNumber)

Gets a reference to an individual Control on a form.

Parameters:

- **pInstIndex** (Integer): The local index of the omnis instance (zero-based).
- **pFormIndex** (Integer): The local index of the omnis form (zero-based).
- **pObjNumber** (Integer):The $order number of the control.
- **pObjRowNumber** (Integer): If non-zero the object belongs to the row inside a complex grid. In which case, this should be the row number of the complex grid in which the control resides.

Returns:

- (Control): Omnis Control instance.

Example

```
var omnisCtl1 = jOmnis.getOmnisFormControl(0,0,1,0);
```

getOmnisHeight()

getOmnisHeight(pElem, pClientArea)

Gets the height of the element, according to Omnis rules, for the frame or client areas.

Parameters:

- **pElem** (Object): The html element.
- **pClientArea** (Boolean): If true, the size is returned excluding borders. If false, it includes the borders.

Returns:

- (Integer): The height of the element.

Example

```
var height= jOmnis.getOmnisHeight(elem, false);
```

getOmnisInst()

getOmnisInst(pInstIndex)

Gets a reference to an Omnis Instance.

Parameters:

- **pInstIndex** (Integer): The local index of the Omnis Instance (zero-based).

Returns:

- (Omnis Instance): Omnis Instance object.

Example

```
var omInst1 = jOmnis.getOmnisInst(0);
```

getOmnisLeft()

getOmnisLeft(pElem)

Gets the left position of the element, according to Omnis rules. This is the left position within the parent element, left of CSS border.

Parameters:

- **pElem** (Object): The html element.

Returns:

- (Integer): The left pixel coordinate.

Example

```
var leftPos = jOmnis.getOmnisLeft(elem);
```

getOmnisTop()

getOmnisTop(pElem)

Gets the top position of the element, according to Omnis rules. This is the top position within the parent element, top of CSS border.

Parameters:

- **pElem** (Object): The html element.

Returns:

- (Integer): The top pixel coordinate.

Example

```
var topPos = jOmnis.getOmnisTop(elem);
```

getOmnisWidth()

getOmnisWidth(pElem, pClientArea)

Gets the width of the element, according to Omnis rules, for the frame or client areas.

Parameters:

- **pElem** (Object): The html element.
- **pClientArea** (Boolean): If true, the size is returned excluding borders. If false, it includes the borders.

Returns:

- (Integer): The width of the element.

Example

```
var width = jOmnis.getOmnisWidth(elem, false);
```

getStyle()

getStyle(pElem, pStyleName, pDoParseInt)

Returns the current specified style attribute. This works regardless of how the style was specified, i.e. via style sheets or the style attribute.

Parameters:

- **pElem** (Object): The html element.
- **pStyleName** (String): The style property name (e.g. “padding-left”).
- **pDoParseInt** (Boolean): True if you wish the result to be parsed as an integer.

Returns:

- (Var): The style’s value.

Example

```
var leftPad = jOmnis.getStyle(elem, "padding-left", true);
```

getTotalHeight()

getTotalHeight(pElem)

Gets the total CSS height of the element.

Parameters:

- **pElem** (Object): The html element.

Returns:

- (Integer): The total height.

Example

```
var totalHeight = jOmnis.getTotalHeight(elem);
```

getTotalWidth()

getTotalWidth(pElem)

Gets the total CSS width of the element.

Parameters:

- **pElem** (Object): The html element.

Returns:

- (Integer): The total width.

Example

```
var totalWidth = jOmnis.getTotalWidth(elem);
```

getWindowClientScroll()

getWindowClientScroll()

Gets the X and Y scroll offsets of the window client area in a JavaScript object .

Parameters:

- None

Returns:

- (Object): JavaScript object with XOffset and YOffset member properties.

Example

```
var topPos = jOmnis.getWindowClientScroll().YOffset;
```

getWindowClientSize()

getWindowClientSize()

Gets the width and height of the window client area in a JavaScript object.

Parameters:

- None

Returns:

- (Object): JavaScript object with width and height member properties.

Example

```
var clientWidth = jOmnis.getWindowClientSize().width;
```

inAppWrapper()

inAppWrapper()

Returns whether the client is running in a wrapper app.

Parameters:

- None

Returns:

- (Boolean): Returns true if the client is running in a wrapper app, false if running in a standard browser.

Example

```
this.inWrapper = jOmnis.inAppWrapper();
```

okMessage()

okMessage(pTitle, pMessage)

Opens an OK message dialog. Same as $showmessage() from Omnis code.

Parameters:

- **pTitle** (String): The dialog title.
- **pMessage** (String): The dialog message.

Returns:

- None

Example

```
jOmnis.okMessage("My Title", "My Message");
```

showDialogEx()

showDialogEx(pTitle, pMessage, pType, pWidth, pHeight, pAutohide, pAtCursor, pModal, pHasClose, pButtonList, pFormInst)

Opens a dialog, with several customizable options.

Parameters:

- **pTitle** (String): The dialog title.
- **pMessage** (String): The dialog message.
- **pType** (String): Dialog style: 'error'(red), 'warning'(yellow), 'success'(green), 'prompt'(blue), 'message'(blue), 'query'(blue)
- **pWidth** (Integer): Width of dialog box, in pixels. Special values: 0 = autosize, null = default fixed size.
- **pHeight** (Integer): Height of dialog box, in pixels. Special values: 0 = autosize, null = default fixed size.
- **pAutoHide** (Integer): Number of seconds before autoclose. Set to null for no autoclose.
- **pAtCursor** (Boolean): true = open at cursor position. false = centered.
- **pModal** (Boolean): Whether the dialog is modal (disables the rest of the form while it's shown).
- **pHasClose** (Boolean): Whether the dialog has a close box.
- **pButtonList** (Array): Array of button objects.
- **pFormInst** (Object): A form instance, required for callback to server on button presses.

Returns:

- None

Example

```
var buttonList = new Array( new button(jOmnis.yesString, true, $yesEvent, jOmnis.yesAccel),
          new button(jOmnis.noString, false, "$noEvent", jOmnis.noAccel),
          new button(jOmnis.cancelString, false, "$cancelEvent", null) );
jOmnis.showDialogEx("myTitle", "My Message", "message", 0, 0, null, false, true, false, buttonList, formInst);
```

parseInteger()

parseInteger(pString)

Converts a string to an integer.

Parameters:

- **pString** (String): The string containing the number.

Returns:

- (Integer): The parsed int, or 0 if parsing failed.

Example

```
var myInt = jOmnis.parseInteger("123");
```

setBackgroundFromOmnisDefinition()

setBackgroundFromOmnisDefinition(pElem, pAttName, pDestElem)

Sets the element's background color and alpha from the element's "data-backgroundcolor" attribute (which specifies a Color Pair).

Parameters:

- **pElem** (Object): The html element which is queried for the color pair.
- **pAttName** (optional) (String): The name of the attribute to query for the Color Pair (defaults to "data-backgroundcolor").
- **pDestElem** (optional) (Object): The html element whose background color & alpha is set (defaults to pElem).

Returns:

- None

Example

```
jOmnis.setBackgroundFromOmnisDefinition(elem);
```

setBorderRadius()

setBorderRadius(pElem, pValue)

Sets the border radius of the specified element.

Parameters:

- **pElem** (Object): The html element.
- **pValue** (String): The new value. A string of 1 to 4 numbers separated by hyphens "n-n-n-n" (number and ordering of hyphen-separated values corresponds with the CSS border-radius standard. Numbers will be interpreted as "px" units).

Returns:

- None

Example

```
jOmnis.setBorderRadius(elem, "10-5");
```

setOmnisHeight()

setOmnisHeight(pElem, pNewHeight)

Sets the height of the element, according to Omnis rules. The height includes the borders and everything inside.

Parameters:

- **pElem** (Object): The html element.
- **pNewHeight** (Integer): The new height, in pixels.

Returns:

- None

Example

```
jOmnis.setOmnisHeight(elem, 150);
```

setOmnisLeft()

setOmnisLeft(pElem, pNewLeft)

Sets the left position of the element, according to Omnis rules. This is the left position within the parent element, left of CSS border.

Parameters:

- **pElem** (Object): The html element.
- **pNewLeft** (Integer): The new left coordinate, in pixels.

Returns:

- None

Example

```
jOmnis.setOmnisLeft(elem, 120);
```

setOmnisRect()

setOmnisRect(pElem, pNewLeft, pNewTop, pNewWidth, pNewHeight)

Sets the position and dimensions of the given element, according to Omnis rules.

Parameters:

- **pElem** (Object): The html element.

- **pNewLeft** (Integer): Left coordinate, in pixels.
- **pNewTop** (Integer): Top coordinate, in pixels.
- **pNewWidth** (Integer): Width, in pixels.
- **pNewHeight** (Integer): Height, in pixels.

Returns:

- None

Example

```
jOmnis.setOmnisRect(elem, 120, 10, 200, 150);
```

setOmnisTop()

setOmnisTop(pElem, pNewTop)

Sets the top position of the element, according to Omnis rules. This is the top position within the parent element, top of CSS border.

Parameters:

- **pElem** (Object): The html element.
- **pNewTop** (Integer): The new top coordinate, in pixels.

Returns:

- None

Example

```
jOmnis.setOmnisTop(elem, 10);
```

setOmnisWidth()

setOmnisWidth(pElem, pNewWidth)

Sets the width of the element, according to Omnis rules. The width includes the borders and everything inside.

Parameters:

- **pElem** (Object): The html element.
- **pNewWidth** (Integer): The new width, in pixels.

Returns:

- None

Example

```
jOmnis.setOmnisWidth(elem, 200);
```

setPropertyBackground()

setPropertyBackground(pElem, pPropNumber, pValue)

Sets the element's background color or alpha.

Parameters:

- **pElem** (Object): The html element.
- **pPropNumber** (Integer): To set the color, pass eBaseProperties.backcolor. To set the alpha, pass eBaseProperties.backalpha.
- **pValue** (Integer or Float): If setting color, this should be an Omnis rgb color value, if setting alpha, it should be a float, between 0 and 1.

Returns:

- (Boolean): Whether the value was successfully set.

Example

```
var success = jOmnis.setPropertyBackground(elem, eBaseProperties.backcolor, 65280);
success = jOmnis.setPropertyBackground(elem, eBaseProperties.backalpha, 0.5);
```

stopEventNow()

stopEventNow(pEvent)

Immediately stops the browser from processing the event any further. (i.e. prevents event bubbling, etc.)

Parameters:

- **pEvent** (Object): The event object. (Standard JavaScript event object)

Returns:

- None

Example

```
jOmnis.stopEventNow(event);
```

setFontSize()

setFontSize(pElem, pSize)

Sets the point size of a font for an element. Allows for special processing for point size zero. (Zero defaults to point size 9)

Parameters:

- **pElem** (Object): The html element.
- **pSize** (Integer): The size, in points.

Returns:

- None

Example

```
jOmnis.setFontSize(elem, 12);
```

getElemPosRelativeToAncestor()

getElemPosRelativeToAncestor(pElem, pAncestor)

Gets the top and left CSS coordinates of an element relative to its ancestor.

Parameters:

- **pElem** (Object): The html element.
- **pAncestor** (Object): The ancestor to calculate the position from (if null then uses document.body)..

Returns:

- (Object): JavaScript Object containing x and y properties. {x:(left), y:(top)}

Example

```
var relPos = jOmnis.getElemPosRelativeToAncestor(elem, ancestor);
```

getDefaultDateFormatCustom()

getDefaultDateFormatCustom()

Gets the default value for the custom format string.

Parameters:

- None

Returns:

- (String): The default custom format string.

Example

```
var defDateFmt = jOmnis.getDefaultDateFormatCustom();
```

setDefaultDateFormatCustom()

setDefaultDateFormatCustom(pFormat)

Sets the default value for the custom format string.

Parameters:

- **pFormat** (String): The new default date format string. (See Date Formatting)

Returns:

- None

Example

```
var fmt = "D/M/y"
jOmnis.setDefaultDateFormatCustom(fmt);
```

getLocaleTimeFormat()

getLocaleTimeFormat()

Gets the client's localized time format string.

Parameters:

- None

Returns:

- (String): The client's localized time format string.

getLocaleShortDateFormat()

```
getLocaleShortDateFormat()
```

Gets the client’s localized short date format string.

Parameters:

- None

Returns:

- (String): The client’s localized short date format string.

getLocaleShortDateTimeFormat()

```
getLocaleShortDateTimeFormat()
```

Gets the client’s localized short datetime format string.

Parameters:

- None

Returns:

- (String): The client’s localized short datetime format string.

getLocaleMediumDateFormat()

```
getLocaleMediumDateFormat()
```

Gets the client’s localized medium date format string.

Parameters:

- None

Returns:

- (String): The client’s localized medium date format string.

getLocaleMediumDateTimeFormat()

getLocaleMediumDateTimeFormat()

Gets the client’s localized medium datetime format string.

Parameters:

- None

Returns:

- (String): The client’s localized medium datetime format string.

getLocaleLongDateFormat()

getLocaleLongDateFormat()

Gets the client’s localized long date format string.

Parameters:

- None

Returns:

- (String): The client’s localized long date format string.

getLocaleLongDateTimeFormat()

getLocaleLongDateTimeFormat()

Gets the client’s localized long datetime format string.

Parameters:

- None

Returns:

- (String): The client’s localized long datetime format string.

getLocaleFullDateFormat()

getLocaleFullDateFormat()

Gets the client’s localized full date format string.

Parameters:

- None

Returns:

- (String): The client’s localized full date format string.

getLocaleFullDateTimeFormat()

getLocaleFullDateTimeFormat()

Gets the client’s localized full datetime format string.

Parameters:

- None

Returns:

- (String): The client’s localized full datetime format string.

trim()

trim(pString, pChars)

Trims any leading or trailing characters which match those defined in pChars, from the string.

Use ltrim() to trim only leading characters.

Use rtrim() to trim only trailing characters.

Parameters:

- **pString** (String): The string to be trimmed.
- **pChars** (String): The characters to trim. These characters are inserted into square brackets in a regular expression, so this should match that format.E.g:
  “abc”: will match a, b or c.
  “a-z”: will match any character between a & z.
  “^abc” will match any character EXCEPT a, b or c.
  Passing an empty string will strip leading & trailing whitespace.

Returns:

- (String): The trimmed string.

Example

```
var str = jOmnis.trim("aaaaaaaHelloaaa","a"); //Returns "Hello"
```

ltrim()

ltrim(pString, pChars)

Same as trim(), but only trims leading chars. See trim() for details.

rtrim()

rtrim(pString, pChars)

Same as trim(), but only trims trailing chars. See trim() for details.

inFirefox()

inFirefox()

Returns whether the client is running in Firefox.

Parameters:

- None

Returns:

- (Boolean): true if running in Firefox.

inOpera()

inOpera()

Returns whether the client is running in Opera.

Parameters:

- None

Returns:

- (Boolean): true if running in Opera.

inChrome()

inChrome()

Returns whether the client is running in Chrome.

Parameters:

- None

Returns:

- (Boolean): true if running in Chrome.

inSafari()

inSafari()

Returns whether the client is running in Safari.

Parameters:

- None

Returns:

- (Boolean): true if running in Safari.

inEdge()

inEdge()

Returns whether the client is running in Edge.

Parameters:

- None

Returns:

- (Boolean): true if running in Edge.

getMethod()

getMethod(pObj, pOmnisMethodName, pInhLevel)

Gets the name of a method in the object, from the Omnis method name.

Parameters:

- **pObj** (Object): The object (Control or Form).
- **pOmnisMethodName** (String): The Omnis method name.
- **pInhLevel** (Integer): Inheritance level for a private method, or passed as -1 for instance/field methods where the inheritance level does not affect the method called.

Returns:

- (String): Method name.

Example

```
var methodName = jOmnis.getMethod(this, "$myMethod", -1);
this[methodName].call(this, "p1",123); // First argument to call() is what the method will use as "this".
```

doMethod()

doMethod(pCallType, pObj, pOmnisMethodName, pFlags, ...)

Calls a method relative to object pObj (“Do method” command implementation).

For calling a method of your control, see callMethod()

Parameters:

- **pCallType** (Integer): An eAnums value specifying how the call is to be made. One of: eAnums.priv, eAnums.$cfield, eAnums.$cinst or eAnums.$cwind.
- **pObj** (Object): The object making the call.
- **pOmnisMethodName** (String): The Omnis method name.
- **pFlags** (Integer): Bitmap of eDoMethodFlags values. (Or’d together)
- **...** (Any Extra Parameters): Any extra parameters will be passed as parameters to the method you are calling.

Returns:

- (Var): The method’s return value.

Example

```
// Call an instance method of your form:
var rtn = jOmnis.doMethod(eAnums.$cinst, this, "$myMethod", eDoMethodFlags.completionEvent,"param1",123);
```

defaultParameter()

defaultParameter(pParam, pDefaultValue)

A helper method to help ensure parameters are initialized with valid values. E.g. Especially useful for optional parameters.

Checks if pParam has been passed (i.e. not null) - if so, it is returned.

If the parameter is null, pDefaultValue is returned.

Parameters:

- **pParam** (Var): The variable whose value to check.
- **pDefaultValue** (Var): The value to return if pParam is null.

Returns:

- (Var): pParam if not null, else pDefaultValue.

Example

```
ctrl_generic.prototype.myMethod = function(p1) {
  p1 = jOmnis.defaultParameter(p1, 0); // Ensure p1 is not null.
  var myVal = p1 * 100; // would cause an error if p1 was null.
};
```

clientPlatform

Type: Integer

An eClientPlatforms value specifying the platform the client is running on.

inFirefoxiOS()

inFirefoxiOS()

Returns true if running in Firefox on iOS.

Parameters:

- None

Returns:

- (Boolean): True if running in Firefox on iOS.

Example

```
if (jOmnis.inFirefoxiOS()) {
  // Some workaround
}
```

inChromeiOS()

inChromeiOS()

Returns true if running in Chrome on iOS.

Parameters:

- None

Returns:

- (Boolean): True if running in Chrome on iOS.

Example

```
if (jOmnis.inChromeiOS()) {
  // Some workaround.
}
```

deepCopy()

deepCopy(pObj)

Returns a deep copy of the passed object.

Parameters:

- **pObj** (Object): The object to make a copy of.

Returns:

- (Object): A deep copy of the passed object.

Example

```
var clone = jOmnis.deepCopy(myObj);
```

getURLParameters()

getURLParameters()

Returns a JSON object containing the current URL's query parameters' keys & values.

All values will be treated as strings.

Parameters:

- None

Returns:

- (Object): An object containing the names & values of the current document URL's query parameters.

Example

```
// User opened the form by visiting http://mysite.com/omnisform?Name=Bert&Age=12
var urlParams = jOmnis.getURLParameters();
// urlParams = {Name: "Bert", Age: "12"}
```

sendDOMEvent()

sendDOMEvent(pTargetElem, pEventType, pBubbles)

Sends an event of the specified type to the passed element.

Parameters:

- **pTargetElem** (Element): Element to send the event to.
- **pEventType** (String): The event type (e.g. "click").
- **pBubbles** (Boolean)(Optional): If true, allow the event to bubble.

Returns:

- None

Example

```
jOmnis.sendDOMEvent(theElem, "click", true);
```

getContainerControl()

getContainerControl(pElem)

Gets the specified element's container control. If none exists, return null.

Parameters:

- **pElem** (Object): The HTML element.

Returns:

- (Object): The container control.

Example

```
var container = jOmnis.getContainerControl(client_elem);
```

jStyles

This is a global object which allows for more efficient style queries on the same element.

If you are using jOmnis.getStyle() multiple times on the same element, you can improve efficiency by using jStyles.

setElem()

setElem(pElem)

Sets up the object for the element whose styles you wish to query.

You must make this call before using get() or getInt().

Parameters:

- **pElem** (Element): The html element whose styles you wish to query.

Returns:

- None

Example

```
jStyles.setElem(elem); //sets up the object to get styles for elem.
var style1 = jStyles.get(styleName); // Get the value for styleName style.
var style2 = jStyles.getInt(styleName); // Get the value for styleName style parsed as an integer.
jStyles.done(); // Make sure to call to perform necessary clean up.
```

get()

get(pStyleName)

Gets the value of the passed style name from the element defined in setElem().

Once you have queried the element for all of the styles you require, you must make sure to call jStyles.done().

Parameters:

- **pStyleName** (String): The style name you wish to query. This can be either a CSS style name (e.g. "background-color"), or a JavaScript style name (e.g. "backgroundColor"), the latter being more efficient.

Returns:

- (Var): The value of the queried style.

Example

```
jStyles.setElem(elem); //sets up the object to get styles for elem.
var style1 = jStyles.get(styleName); // Get the value for styleName style.
var style2 = jStyles.getInt(styleName); // Get the value for styleName style parsed as an integer.
jStyles.done(); // Make sure to call to perform necessary clean up.
```

getInt()

getInt(pStyleName)

Gets the Integer value of the passed style name from the element defined in setElem().

Once you have queried the element for all of the styles you require, you must make sure to call jStyles.done().

Parameters:

- **pStyleName** (String): The style name you wish to query. This can be either a CSS style name (e.g. "background-color"), or a JavaScript style name (e.g. "backgroundColor"), the latter being more efficient.

Returns:

- (Integer): The value of the queried style, parsed as an integer.

Example

```
jStyles.setElem(elem); //sets up the object to get styles for elem.
var style1 = jStyles.get(styleName); // Get the value for styleName style.
var style2 = jStyles.getInt(styleName); // Get the value for styleName style parsed as an integer.
jStyles.done(); // Make sure to call to perform necessary clean up.
```

done()

done()

You must call done() on jStyles after you have finished querying the element for styles. This will perform clean up logic, and remove any cloned elements created during this process from the DOM.

It is not necessary to call done() if you are about to call setElem() again on a different element.

Parameters:

- None

Returns:

- None

Example

```
jStyles.setElem(elem); //sets up the object to get styles for elem.
var style1 = jStyles.get(styleName); // Get the value for styleName style.
var style2 = jStyles.getInt(styleName); // Get the value for styleName style parsed as an integer.
jStyles.done(); // Make sure to call to perform necessary clean up.
```

Theme - Static Properties

The Theme has some static properties that are accessed without requiring the need to get the instance first.

These are accessed via window.Theme (or simply Theme).

.COLORS

Type: Object

The key-value pairs of theme constants, which match with their respective kJSThemeColor… constant in Omnis.

Example

```
console.log(Theme.COLORS.primary); // logs to the console: -2147483462 (the same value as kJSThemeColorPrimary
```

.COLOR_NAMES

Type: Object

The key-value pairs of theme constant names.

Theme - Instance Methods

The Theme instance can be obtained from any control by calling getTheme() as documented in the Control – Instance Methods section.

The Theme contains methods to get the correct theme colors to apply to elements. These methods replace a lot of the color functionality, which have been deprecated, from jOmnis.

In Studio 11, static versions of the theme instance methods **getColorString()** and **getTextColorString()** were added. The static versions are the same as the instance methods except you don't need a theme instance to call them.

The existing instance methods are retained for backwards compatibility, but they are deprecated in Studio 11 and you should use the new static versions.

getColorString()

getColorString(color, defaultColor, alpha)

Gets string suitable for assigning to a css style.

Parameters:

- **color** (Number): The input color value (either an RGB Int or a Theme.COLORS constant).
- **defaultColor** (optional) (Number): A default color to use, in the case that 'color' is kColorDefault or an unknown constant. Can also be a Theme.COLORS constant.
- **alpha** (optional) (Number): An alpha value between 0 and 1. Defaults to 1.

Returns:

- (String) An RGB color string.

Example

```
const theme = this.getTheme(); // Called from within a control to get the theme object.
const colorStr = theme.getColorString(newColor, this.DefaultColors[BaseDefaultColorAttributes.BACKGROUND], 1);
this.client_elem.style.backgroundColor = colorStr; // Assign the background color of the client element to col
```

getTextColorString()

getTextColorString(textColor, onColor, defaultColor)

Gets string suitable for assigning to a css style similar to getColorString() but automatically calculates the text color based on the onColor if textColor is kColorDefault and onColor is a theme color with an associated text color.

Parameters:

- **textColor** (Number): The color (either an RGB integer or a Theme.COLORS constant) for the text.
- **onColor** (Number): The background color on which the text will be displayed.
- **defaultColor** (Number): The color to fall back to in the case that textColor is kColorDefault & onColor is not a valid theme color with associated text color.

Returns:

- (String): An RGB color string.

Example

```
const theme = this.getTheme(); // Called from within a control to get the theme object.
backColor = this.resolveDefaultColor(backColor, BaseDefaultColorAttributes.BACKGROUND); // ensure backColor is
const textColorStr = theme.getTextColorString(newTextColor, backColor, this.DefaultColors[BaseDefaultColorAttr
this.client_elem.style.color = textColorStr; // Assign the color of the client element to textColorStr.
```

getColorRGB()

getColorRGB(color, defaultColor)

Gets the resolved Omnis RGB Integer of the specified color (resolves color constants to their true RGB values).

Parameters:

- **color** (Number): The input color value (either an Omnis RGB Int or a color constant).
- **defaultColor** (optional) (Number): A default color to use, in the case that 'color' is kColorDefault or an unknown constant. Can also be a Theme.COLORS constant, but must not be kColorDefault (-1).

Returns:

- (Number) An Omnis RGB Integer representing the color.

Example

```
const theme = this.getTheme(); // Called from within a control to get the theme object.
const rgbValue = theme.getColorRGB(this.getProperty(eBaseProperties.backcolor));
```

OMColor

The **OMColor** class provides **static** helper methods for common color operations.

TintRGB()

TintRGB(rgbInt, tintFactor, tintType)

A **static** function which creates a new Omnis RGB Integer which is tinted to become a lighter or darker shade.

Parameters:

- **rgbInt** (Number): The input color value (an Omnis RGB Int), which you wish to 'tint'.
- **tintFactor** (optional) (Number): The 'strength' of the tint (0-255). This can be an OMColor.eTintFactor enum value.
- **tintType** (optional) (Number): An OMColor.eTintType enum value, specifying whether to lighten, darken, or determine automatically based on the initial color's 'luminance' (effectively how 'light' it is). Defaults to AUTO.

Returns:

- (Number) A new Omnis RGB Integer representing the tinted color.

Example

```
const theme = this.getTheme();
const bgColor = theme.getColorRGB(this.getProperty(eBaseProperties.backcolor));
const tintedColor = OMColor.TintRGB(bgColor);
```

TintRGBWithColor()

TintRGBWithColor (baseRGB, tintRGB, tintFactor)

A **static** function which creates a new Omnis RGB Integer which is tinted towards the passed 'tintRGB' color.

Parameters:

- **baseRGB** (Number): The input color value (an Omnis RGB Int), which you wish to 'tint'.
- **tintRGB** (Number): The color (an Omnis RGB Int) to use to 'tint' the base color.
- **tintFactor** (optional) (Number): The 'strength' of the tint (0-255). This can be an OMColor.eTintFactor enum value.

Returns:

- (Number) A new Omnis RGB Integer representing the tinted color.

Example

```
const theme = this.getTheme();
const bgColor = theme.getColorRGB(this.getProperty(eBaseProperties.backcolor));
const redTintColor = 0x0000FF;
const tintedColor = OMColor.TintRGBWithColor(bgColor, redTintColor, OMColor.eTintFactor.SMALL); // Returns the
```

eTintFactor

A **static** enum representing some default tint factors, which can be applied to OMColor's tinting functions.

Values

- DEFAULT
- SMALL
- MEDIUM

eTintType

A **static** enum representing whether to lighten or darken the color, when used with OMColor.TintRGB).

Values

- AUTO
- LIGHTEN
- DARKEN

jOmnisEffects

This class provides ways of adding visual effects.

It is available globally, on the *window*.

addRippleToElem()

```
addRippleToElem(elem, theme, hasContainer, elemColor, keys, keyElement, directColor)
```

Adds a ripple effect to the passed element.

Only add once - once added, the ripple effect will be triggered whenever the element is clicked (or optional keys are pressed while it has focus).

Parameters:

- **elem** (Element): The HTML element to apply the ripple to.
- **theme** (Theme): The theme instance to use to resolve themed colors.
- **hasContainer** (optional) (Boolean): If true, the ripple is applied to the container element. If false, it is applied to elem.
- **elemColor** (optional) (Omnis RGB Int): The color of the element to which the ripple will be applied. A tinted color will be generated from this for the ripple (unless *directColor* is provided).
- **keys** (optional) ([String]): An array of keyboard key names, which should fire the ripple when pressed. E.g. ["Enter", " "]
- **keyElement** (optional) (Element): The element which receives keyboard events. Defaults to *elem*.
- **directColor** (optional) (Omnis RGB Int): If specified, a color to apply directly to the ripple, rather than calculated from the *'elemColor'*.

Returns

- (OmnisRipple) - the ripple instance. Currently no need to hold on to this.

Example

```
const theme = this.getTheme();
jOmnisEffects.addRippleToElem(
this.getClientElem(),
theme,
false,
this.getProperty(eBaseProperties.backcolor),
["Enter", " "], // trigger the ripple effect if Enter or spacebar are pressed
null,
0x0000ff // Set the ripple color directly to red.
);
```

setRippleColor()

setRippleColor(elem, theme, elemColor, directColor)

Updates the color of a previously created ripple element.

Parameters:

- **elem** (Element): The HTML element which already has a ripple applied.
- **theme** (Theme): The theme instance to use to resolve themed colors.
- **elemColor** (Omnis RGB Int): The color of the element to which the ripple will be applied. A tinted color will be generated from this for the ripple (unless *directColor* is provided).
- **directColor** (optional) (Omnis RGB Int): If specified, a color to apply directly to the ripple, rather than calculated from the *'elem-Color'*.

Example

```
const theme = this.getTheme();
jOmnisEffects.setRippleColor(
this.getClientElem(),
theme,
this.getProperty(eBaseProperties.backcolor), // this could be null as 'directColor' is specified
0x00ff00 // Set the ripple color directly to green.
);
```