ECMA/TC39/99/7
Standard ECMA-xxx
April 1999

ECMA

Standardizing Information and Communication Systems

ECMAScript Components Specification

Phone: +41 22 849.60.00 - **Fax:** +41 22 849.60.01 - **URL:** http://www.ecma.ch - **Internet:** helpdesk@ecma.ch

Standard ECMA-xxx
April 1999

ECMA

Standardizing Information and Communication Systems

ECMAScript Components Specification

Phone: +41 22 849.60.00 - **Fax:** +41 22 849.60.01 - **URL:** http://www.ecma.ch - **Internet:** helpdesk@ecma.ch

MB 9t39-007.doc 03-05-99 14,54

Brief History

ECMA Technical Committee TC-39 was formed to standardize the syntax and semantics of a general purpose, cross platform, vendor-neutral scripting language, known as ECMAScript. This Standard builds upon the ECMA-262 Standard to provide a functional specification for the componentisation and reuse of ECMAScript. It is based upon the practical experience of ECMA member companies. The development of this Standard began in April 1998.

Table of contents

1 Scope

This Standard defines conventions for the componentisation and reuse of ECMAScript code.

2 Conformance

A conforming implementation of ECMAScript Components must provide and support all the component interface descriptors and hosting environment conventions detailed in this specification. Additionally, a conforming implementation of this International standard shall interpret characters in conformance with the Unicode Standard, Version 2.0, and ISO/IEC 10646-1 with UCS-2 as the adopted encoding form, implementation level 3. If the adopted ISO/IEC 10646-1 subset is not otherwise specified, it is presumed to be the BMP subset, collection 300.

A conforming implementation of ECMAScript Components is permitted to provide additional component interface descriptors beyond those described in this specification.

3 References

ECMA-262:1998 -- ECMAScript Language Specification.

W3C REC-xml-19980210 -- Extensible Markup Language (XML) 1.0.

W3C REC-rdf-syntax-19990222 -- Resource Description Framework (RDF) Model and Syntax Specification.

ISO/IEC 10646-1:1993 Information Technology -- Universal Multiple-Octet Coded Character Set (UCS), including amendments 1 through 9 and technical corrigendum 1.

ISO/IEC 646.IRV:1991 -- Information Processing -- ISO 7-bit Coded Character Set for Information Interchange.

Unicode Inc. (1996), The Unicode Standard™, Version 2.0. ISBN: 0-201-48345-9, Addison-Wesley Publishing Co., Menlo Park, California.

4 Overview

4.1 ECMAScript Component Overview

This section contains an informal overview of ECMAScript Components.

Component-based development has been a key software development trend, as demonstrated by the widespread popularity and acceptance of several component models (e.g., Java Beans, COM, CORBA). Components allow developers to treat code modules as “black boxes”, thus increasing programming efficiency and making code reuse and management easier. However, despite the growing popularity of ECMAScript as a scripting language, script authors have been largely unable to take advantage of component-based development from within ECMAScript. This technology gap has made scripting less efficient and code reuse more difficult.

In lieu of an open component standard, many script authors have chosen to utilize .js include files in an effort to reuse their code. While .js files provide a better solution for reuse than simply embedding script code into HTML files, they are far less optimal than a component-based approach to application and site development.

The ECMAScript Component Standard fills this void, and builds on the ECMA-262 standard to provide a commonly agreed upon means of componetizing ECMAScript code.

An ECMAScript Component is defined by creating an ECMAScript object (defined in Section 4.3.3 of ECMA-262), placing it inside of a .js include file, and adding a public interface written in XML (Extensible Markup Language) to describe the component. By following these conventions, script authors can encapsulate their ECMAScript code into a component wrapper. Other script developers can reuse these components in their Web pages and server-side applications, working with the component's public interface instead of dealing with the underlying implementation details. See Figure 1.

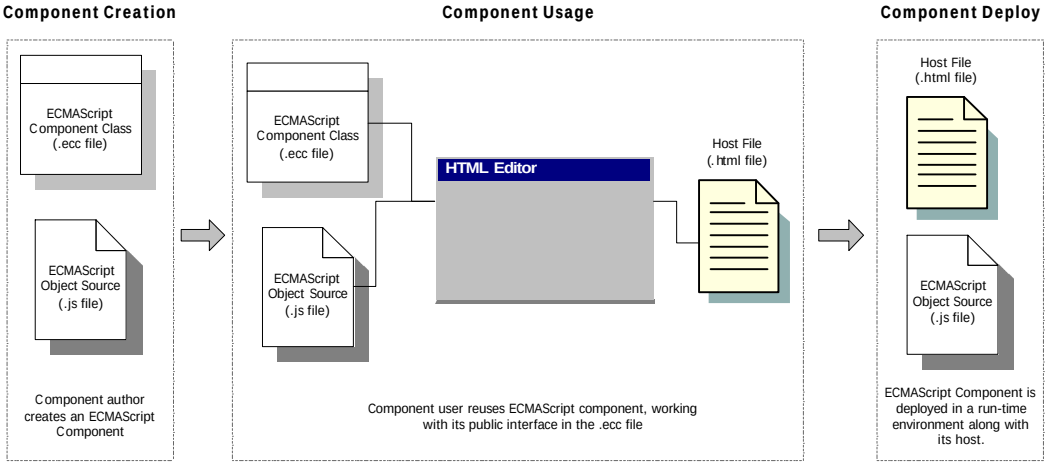

Figure 1: Component Lifecycle

The ECMAScript Component specification uses an XML-based vocabulary to define the public interface for the component. The interface definition is enclosed by a **`<component></component>`** tag pair. Within this element, the component author can add additional tags that define meta-information for the component. This definition is then housed in a ECMAScript Component Class (.ecc) file and linked to the implementation source code in a standard .js include file. This external interface is normally for design-time only and can be "detached" at runtime to simplify deployment and minimize bandwidth considerations.

4.3 Definitions

The following are informal definitions of key terms associated with ECMAScript Components.

4.3.1 .JS Include File

A ***.js include file*** is an ASCII text file that contains ECMAScript source code and is typically referenced by an external file, such as an HTML document.

4.3.2 Type

A ***type*** is a set of data values.

4.3.3 Constructor

A ***constructor*** is a function object that creates and initialises objects. Each constructor has an associated prototype object that is used to implement inheritance and shared properties.

5 ECMAScript Component Structure

ECMAScript Components consist of three main parts:

- **ECMAScript Component Interface**. The external interface for a ECMAScript Component is defined using XML tags and housed in a ECMAScript Component Class (.ecc) file.
- **ECMAScript Implementation**. The implementation of a ECMAScript Component is provided in a .js include file.
- **Additional Resource Files**. Support files may optionally be included with the component to provide fuller capabilities at design time.

5.1 ECMAScript Component Interface

ECMAScript Components use an XML-based vocabulary to define the external interface. The following is the skeleton structure of a typical ECMAScript Component:

```
<component name="com.mycompany.MyComponent" displayname="MyComponent"
   env="client" src="SourceFile.js" hint="Component hint">
   <help url="helpFile"/>
```

```
   <icon url16="iconFile"/>
   <property name="propertyName" displayname=" nameToDisplay"
       hint="propertyHint">
      <get name="getPropertyMethod"/>
      <set name="setPropertyMethod"/>
   </property>
   <method name="methodName" displayname="nameToDisplay">
      <parameter name="paramName"/>
   </method>
   <event name="onEvent"/>
</component>
```

The “fully qualified” name of the ECMAScript Component Class file corresponds to the NAME attribute of the COMPONENT element (the “.” character in the NAME is used to delimit the storage “directories” relative to a well known root). The component name also corresponds to the name of the ECMAScript object constructor by substituting “.” for “_” within the constructor function's name. To illustrate, a component named *com.MyCompany.MyComponent* would be contained inside of com\MyCompany\MyComponent.ecc and have an associated constructor function named **`com_MyCompany_MyComponent`**.

The ECMAScript Component Class specification supports the notion of implicit referencing to be as simple and flexible as possible for the ECMAScript Component author. As a result, certain elements are not required to be explicitly defined within the .ecc definition. For example, the SRC attribute of the **`<component>`** is optional. If present, then the specified .js file will be its value. If not present, then the host environment assumes the associated .js file is the same file name as the .ecc file (except with a .js extension) and in the same directory location. A second example is if the **`<help>`** element is not explicitly specified in the .ecc file, the host will look for an .html file bearing the same file name prefix in the .ecc file directory.

5.2 ECMAScript Object Implementation

ECMAScript employs a prototype object to serve as the basis for object creation. A prototype object is essentially a template from which to get initial properties for a new object as well as to define its methods.

In class-based languages, a class is defined with a class definition section. Included in a class definition are specific methods called constructors that are used to create new instances of the class. ECMAScript merges these two ideas into its object constructor function, which both defines the object and creates a new instance when called with the new operator. Any ECMAScript function can potentially be a ECMAScript object constructor.

The implementation section of the ECMAScript Component is an ECMAScript object constructor defined within a .js include file. For users with existing .js file libraries, no explicit changes to their code are necessarily needed to turn this code into ECMAScript Components. However, to prevent name space collisions, the component constructor should follow the hierarchical convention of *domainName_subdomainName_category1_…categoryN_componentName* or *category1_…categoryn_componentName*.

The following is a sample implementation of an ECMAScript Component:

```
function com_myCompany_myName( s ) {
        this.Name = s;
        this.getName = _getName;
        this.setName = _setName;
        this.nameChanged = _nameChanged;
        this.makeNameBold = _makeNameBold;
        function _getName() {
                return this.Name;
        }
```

```
        function _setName( s ) {
                this.Name = s;
                this.nameChanged();
        }
        function _makeNameBold() {
                this.Name = "<B>" + this.Name + "</B>";
        }
        function _nameChanged() {
        }
}
```

5.3 Additional Resource Files

For complete design-time support, additional files may be included as part of a component package. Examples include: Graphic (.gif) file for visual representation of the component within the design-time host environment and a Help (.html) file to support the component user.

6 ECMAScript Component Interface Definition

The public interface to ECMAScript Components is defined using XML elements. The following section describes the available elements. The DTD is provided in Section 11.

6.1 COMPONENT Element

Syntax

```
<component
        name="componentname"
        displayname="displayname"
        src="url"
        env="client" | "server" | "both" | "either"
        hint="hinttext"
        version="versionnumber"
        needsform="yes" | "no"
        ishidden="yes" | "no">
</component>
```

Purpose

Encapsulates the entire ECMAScript Component interface definition. All XML elements that define the component must be enclosed inside of this element. (Required)

Attributes

Name	***Description***	***Required***
name="componentname"	Full name of component, which is the same name as the constructor function (substituting "." for "_")	Y
displayname="displayname"	Display name of property (NAME value is used if not specified)	N
src="url"	URL of .js file containing implement code (if not present, looks for .js file with same name as .ecc file)	N
env="client" | "server" | "both" | "either"	Environment where the component runs (defaults to BOTH)	N
hint="hinttext"	Provides a brief description of the component that can be displayed as a "hint" in the design-time host environment.	N
version="versionnumber"	Major/minor version number in the form n.n	N
needsform = "yes" | "no"	Indicates that component must be contained by form (defaults to NO)	N
ishidden = "yes" | "no"	Hide in design-time WYSIWYG view (defaults to	N

	NO)	

Child Of

None

Example

```
<component
        name="com.mycompany.selectnav"
        displayname="SelectNav"
        src="selectnav.js"
        env="client"
        hint="Navigational Select List"
        version="1.0"
        needsform="yes">
</component>
```

6.2 HELP Element

Syntax

```
<help src="url"/>
```

Purpose

Defines an HTML-based Help file for using the component. If specified, design-time environments can display this Help information to component users.

Attributes

Name	***Description***	***Required***
src="url"	URL of associated HTML help file	Y

Child Of

component

Example

```
<help src="http://www.mycompany.com/esc/help/selectnav.hmtl"/>
```

6.3 ICON Element

Syntax

```
<icon src16="url" src32="url"/>
```

Purpose

Specifies the graphic file used to represent the component in the design-time host environment. Graphic files must be in .gif format and either 16x16 or 32x32 pixels per inch.

Attributes

Name	***Description***	***Required***
src16="url"	URL of associated 16x16 .gif file	N
src32="url"	URL of associated 32x32 .gif file	N

Child Of

component

Example

```
<icon src16="http://www.mycompany.com/esc/icon/selectnav.gif"/>
```

6.4 CUSTOMIZER Element

Syntax

```
<customizer
        type="ecmascript" | "html"
        class="classname"/>
```

Purpose

Defines a customizer of the component. The customizer would be triggered at design-time to enable users to have a customized means of working with the component.

Attributes

Name	***Description***	***Required***
type="ecmascript" | "html"	Type of customizer (defaults to ECMASCRIPT)	Y
class="classname"	Name of customizer	Y

Child Of

component

Example

```
<customizer type="ecmascript"
   url="http://www.mycompany.com/esc/selectnav_customizer.html"/>
```

6.5 USES Element

Syntax

```
<uses file="filename"/>
```

Purpose

Specifies a file that is used by the component at run-time. Design-time environments can use this information to bundle files when an ECMAScript Component is deployed.

Attributes

Name	***Description***	***Required***
file="filename"	Name of file used by component (relative to component)	Y

Child Of

component

Example

```
<uses file="rollover_top.gif"/>
<uses file="rollover_bottom.gif"/>
```

6.6 META Element

Syntax

```
<meta
    name="custommetatag"
    value="value"/>
```

Purpose

Describes customized component meta data. Component authors can add optional meta-data information to describe the component.

Attributes

Name	*Description*	*Required*
name="custommetatag"	Name of meta data attribute	Y
value="value"	Value of meta data attribute	Y

Child Of

component

Example

```
<meta name="copyright" value="copyright ©1998, mycompany, inc."/>
```

6.7 PROPERTY Element

Syntax

```
<property
        name="propertyname"
        displayname="displayname"
        hint="shortdescription"
        valueset="valueset"
        type="string" | "number" | "boolean" | "unknown"
        env="client" | "server" | "both"
        defaultvalue="defaultvalue"
        runtimeaccess="readwrite" | "readonly" | "writeonly" | "none"
        isinitializable="yes" | "no"
        isexpert="yes" | "no"
        isbound="yes" | "no"
        isdefault="yes" | "no">
</property>
```

Purpose

Describes a public property of a component.

Attributes

Name	*Description*	*Required*
name="propertyname"	Name of property	Y
displayname="displayname"	Display name of property (NAME value is used if not specified)	N
hint="shortdescription"	Short description of component	N
valueset="valueset"	Range of allowed values for a property	N
type="string" | "number" | "boolean" | "unknown"	Data type for the property (defaults to STRING)	N
env="client" | "server" | "both"	Environment where property can be accessed (defaults to BOTH)	N
defaultvalue="defaultvalue"	Default value of property	N
runtimeaccess="readwrite" | "readonly" | "writeonly" | "none	Indicates accessibility of property at runtime (defaults to READWRITE)	N
isinitializable="yes" | "no"	Indicates whether property is initializable (defaults to YES)	N
isexpert="yes" | "no"	Denotes property is for advanced uses (defaults to NO)	N
isbound="yes" | "no"	Specifies that a onChange event is fired when changed (defaults to NO)	N
isdefault="yes" | "no"	Specifies that property is default property in	N

	design-time environment (defaults to NO)	

Child Of

component

Example

```
<property name="id" hint="Business ID" type="string" env="client"
          defaultvalue="10010" runtimeaccess="readwrite"
          isbound="yes" isdefault="yes">
            <get name="get_BusinessID"/>
            <set name="set_BusinessID"/>
</property>
```

6.8 GET Element

Syntax

```
<get name="getmethodname"/>
```

Purpose

Specifies the name of a function used to retrieve a value of the property.

Attributes

Name	***Description***	***Required***
name="getmethodname"	Name of Get method	Y

Child Of

property

Example

See property

6.9 SET Element

Syntax

```
<set name="setmethodname"/>
```

Purpose

Specifies the name of a function used to set the value of the property.

Attributes

Name	***Description***	***Required***
name="setmethodname"	Name of Set method	Y

Child Of

property

Example

See property

6.10 PROPERTYEDITOR Element

Syntax

```
<propertyeditor
        type=""ecmascript"" | "html"
        class="classname"/>
```

Purpose

Defines custom property editor for component.

Attributes

Name	*Description*	*Required*
type="ecmascript" | "html"	Type of custom property editor (defaults to ECMASCRIPT)	N
class="classname"	Name of custom property editor	Y

Child Of

property

6.11 METHOD Element

Syntax

```
<method
        name="methodname"
        displayname="displayname"
        hint="shortdescription"
        env="client" | "server" | "both"
        isexpert="yes" | "no">
</method>
```

Purpose

Defines a method of a component.

Attributes

Name	*Description*	*Required*
name="methodname"	Name of method to expose	Y
displayname="displayname"	Display name of method (NAME value is used if not specified)	N
hint="shortdescription"	Short description of method	N
env="client" | "server" | "both"	Environment where method can be called (defaults to BOTH)	N
isexpert="yes" | "no"	Denotes method is for advanced uses (defaults to NO)	N

Child Of

component

Example

```
<method name="render" env="client" hint="Renders Component">
        <parameter name="delay" type="boolean"/>
</method>
```

6.12 PARAMETER Element

Syntax

```
<parameter
        name="paramename"
        displayname="displayname"
```

```
        hint="shortdescription"
        valueset="valueset"
        type="string" | "number" | "boolean" | "unknown"/>
```

Purpose

Specifies a parameter of a method.

Attributes

Name	*Description*	*Required*
name="parametername"	Name of parameter	Y
displayname="displayname"	Display name of parameter (NAME value is used if not specified)	N
hint="shortdescription"	Short description of parameter	N
type="string" | "number" | "boolean" | "unknown"	Data type for the parameter (defaults to STRING)	N

Child Of

method

Example

See method

6.13 EVENT Element

Syntax

```
<event
        name="methodname"
        displayname="displayname"
        hint="shortdescription"
        isexpert="yes" | "no"
        isdefault="yes" | "no">
</event>
```

Purpose

Event of a component.

Attributes

Name	*Description*	*Required*
name="eventname"	Name of event	Y
displayname="displayname"	Display name of event (NAME value is used if not specified)	N
hint="shortdescription"	Short description of event	N
isexpert="yes" | "no"	Denotes event is for advanced uses (defaults to NO)	N
isdefault="yes" | "no"	Specifies that event is default property in design-time environment (defaults to NO)	N

Child Of

component

Example

```
<event
    name="onRenderDone"
```

```
    hint="Called when render is completed"/>
```

6.14 ?XML Element

Syntax

```
<?xml
        version="versionnumber"
        standalone="yes" | "no" ?>
```

Purpose

Indicates that the ECMAScript Component interface definition file is an XML document and should be parsed as XML. This XML processor instruction should defined just once and be the first element defined in the .ecc file.

Attributes

Name	***Description***	***Required***
version="versionnumber"	Major/minor version of XML in n.n format (default is "1.0")	N
standalone="yes" | "no"	Specifies whether the file should be validated against a DTD	N

Child Of

none

Example

```
<?xml version="1.0" ?>
```

7 Hosting ECMAScript Components

This part describes how ECMAScript Components are persisted and hosted in an ECMAScript enabled environment.

7.1 Meta-information Specification

The XML meta-information describing the ECMAScript Component is available to design tools targeting an ECMAScript enabled environment. This meta-information shall be in an ASCII file.

7.2 Component Implementation

Components implementation consists of ECMAScript source. This code must be available to the context from which an instance of the component will be created. The component must adhere to the meta-information specification. Specifically, the component must implement the following:

7.2.1 A Constructor

The component must have a constructor function. In order to avoid name space collisions, component names ought to follow the hierarchical convention of ***<domainName>_<subdomainName>_<category1>_<category2>_...<component_name>*** or ***<category1>_<category2>_...<component_name>***.

Occurrences of "." in the **<component>** element's **name** are converted to "_" to ensure a valid ECMAScript identifier for the constructor function. Tools have the option of organizing ECMAScript Component implementations in a hierarchical manner, grouped by identifiers delimited by the "." character in the **<component>** element's **name** attribute.

The constructor function must return an object which represents an instance of the component. This returned object must implement the below features.

7.2.2 Specified Functions

The ECMAScript object returned by the constructor function must have functions corresponding to each of **<method>** tags in the meta-information specification.

Not all ECMAScript functions need to be represented by **<method>** tags in the meta-information.

7.2.3 Specified Properties

The ECMAScript object returned by the constructor function must have properties corresponding to each of the **<property>** tags in the meta-information specification. This means that external access, as specified by the **<property>** tag's **runtimeaccess** attribute, must be available either through direct ECMAScript property manipulation, or through the use of the Set/Get functions specified within the **<property>** tag.

Not all ECMAScript properties need to be represented by **<property>** tags in the meta-information.

7.2.4 Specified Events

The ECMAScript object returned by the constructor function must implement events as specified in the **<event>** tags in the meta-information specification. Events are conditions specific to an ECMAScript Component which can be made to trigger the calling of ECMAScript code, typically external to the component. The **name** attribute of the **<event>** tag determines the name of an ECMAScript function on the object returned by the constructor, which is called by the component when it determines that that event has occurred.

In order for an ECMAScript function to be invoked upon the occurrence of an event, it must be assigned to the ECMAScript object returned by the constructor. This “wiring” must occur after the constructor has been called, but prior to the occurrence of the event.

For example, a component might have code:

```
if ( old != new ) {
        this.onPropertyChange( name, old, new );
}
```

which calls an external function **dateWidget7_onPropertyChange** because the wiring code:

```
dateWidget7.onPropertyChange = dateWidget7_onPropertyChange;
```

was called after the instance **dateWidget7** was created by a call to the component's constructor function.

7.3 Initialization Code

Initialization code is ECMAScript code which calls an ECMAScript Component's constructor. The host environment shall execute this code once when the context in which the component “lives” is initialized. Typically all components having a similar lifetime are instantiated prior to usage of any of these components.

The ECMAScript code which calls an ECMAScript Component's constructor function passes as a parameter an object which may contain initial values for properties of the component (as specified by the **<property>** tags in the meta-information specification). It is the responsibility of the component to initialize itself based upon the values passed to it in the call to its constructor function.

7.4 Wiring Code

Code which assigns event handling functions to properties of ECMAScript Components is considered wiring code. (See Section 7.2.4 Specified Events above.) This code ought to be embedded in such a way that the host environment executes it following all the “Initialization Code” but prior to all the “Usage Code”.

7.5 Usage Code

Usage code refers to ECMAScript code which is executed by the host after the component is fully initialized, and wired. This is code which exercises the actual functionality of an ECMAScript Component.

For example a component may have a "render" function which employs host services to generate “output” or host specific objects having a visual representation.

7.6 Standard Code Format

While the usage of an ECMAScript Component could be entirely “hand coded” using tools unaware of ECMAScript Components, a standard for "generated" code is of great benefit to users of tools which are aware of ECMAScript Components. This section defines some standards to which such tools which parse and generate such code ought to adhere.

7.6.1 Initialization Code

The following shows a format for the code which instantiates and initializes an ECMAScript Component:

```
_param_ = new Object();
_param_.<propName1> = <propValue1>;
_param_.<propName1> = <propValue1>;
_param_.<propName1> = <propValue1>;
_param_.<propName1> = <propValue1>;
_param_.<propName1> = <propValue1>;
<instanceName> = new <constructorName>( _param_ );
```

or, for example:

```
_param_ = new Object();
_param_.defaultValue = false;
_param_.value = false;
_param_.textToDisplay = "Check Box Text";
com_acme_widgets_CheckBox1 = new com_acme_widgets_CheckBox(_param_);
```

7.6.2 Wiring Code

Wiring code assigns an ECMAScript function to a property of the component instance. The function name shall be the concatenation of ECMAScript identifier corresponding to the instance of the component and “_” and the event function name. This will guarantee uniqueness. For example:

```
dateWidget7.onPropertyChange = dateWidget7_onPropertyChange;
```

7.6.3 Usage Code

Unlike initializing and wiring code, usage code may be completely "free form" (hand written), even in a tool which is ECMAScript Component aware. Only standardized functions (such as render()) must be generated and parsed by ECMAScript Component aware tools. A tool generated usage might therefore look like:

```
dateWidget7.render();
```

8 HTML Hosting

This section describes how ECMAScript Components are hosted in an HTML document. This implementation is compatible with ECMAScript enabled browsers. Standard hosting specifications for target run-time environments (such as HTML browsers) allow for compliant tools to share source files and recognize embedded ECMAScript Components.

8.1 Component Implementation

The implementation of the component resides in a .js include file. This .js file is identified by the **`source`** attribute of the **`<component>`** element of the meta-information specification.

The meta-information resides in an .ecc file. This .ecc file is referenced by an extended attribute of **`<script>`** tag which includes the component source into the host .html file (see Section 8.2 Inclusion Code below). This allows a tool to discover the interface for a component used by an existing HTML file.

8.2 Inclusion Code

Inclusion code is ECMAScript code which makes the component implementation code accessible from a page which contains an instance of an ECMAScript Component. This code has the following format:

```
<script
  language="javascript"
  type="text/javascript"
  src="widgets.js"
  purpose="component"
  classname="com.acme.widgets.fooBar"
</script>
```

The **classname** is the same value as the **name** attribute of the **<component>** element in the .ecc file. It also corresponds to the .ecc file's location, which can be derived from this value. The .ecc extension is added to this string, and the "." character delimits the storage "directories" relative to a well-known root. For example, a tool in stalled into directory *x* may keep its components in *x/components*, and the above components .ecc file would be located at *x/components/com/acme/widgets/foobar.ecc*.

Inclusion code shall be placed within the **<head>** tag of the document.

8.3 Event Handling Code

Event handling code is ECMAScript code, not part of the action component source, which is executed in response to ECMAScript Component events. This code is recognizable by design tools because it follows naming conventions for event handling code, as well as being specified by the Event Wiring code (see below) this code also resides within the **<head>** tag of the document.

ECMAScript Component event handling function names are generated by appending "_" and the event name to the component instance name. For example, if component whose ECMAScript identifier is **dateWidget7** and whose **onClick** event has an event handler, then the event handler name would be **dateWidget7_onClick**:

```
<script language="javascript" type="text/javascript"
  purpose="eventhandler">
function dateWidget7_onClick() {
   alert( 'dateWidget was clicked' );
}
</script>
```

The signature of the event handler is obtained from the meta-information specification.

8.4 Initialization Code

Initialization code serves the purpose of instantiating and passing persisted property values to an ECMAScript Component. The actual code is specified in the previous section (see Section 7.6.1). This code is contained in a function whose name is generated by appending "_init" to the component instance name. This ECMAScript function is embedded in a **<script purpose="intialization"....>** tag, with a call to that function immediately following the function. Using the example from the previous section we have:

```
<script language="javascript" type="text/javascript"
  purpose="initialization>
function aCheckBox1_init {
  _param_ = new Object();
  _param_.defaultValue = false;
  _param_.value = false;
  _param_.textToDisplay = "Check Box Text";
  aCheckBox1 = new com_acme_widgets_CheckBox(_param_);
}
aCheckBox1_init();
function dateWidget7_init {
  _param_ = new Object();
  _param_.shortFormat = true;
  dateWidget7 = new com_foo_bar_dateWidget(_param_);
}
dateWidget7_init();
```

```
aCheckBox1.onChange = aCheckBox1_onChange;
dateWidget7.onClick = dateWidget7_onClick;
dateWidget7.onChange = dateWidget7_onChange;
</script>
```

8.5 Wiring Code

Wiring code is the code which assigns to ECMAScript Components their event handlers. For this code to work, and for maximal flexibility, this code is executed after all the ECMAScript Components have been initialized, and after all the event handlers have been parsed. For HTML hosting wiring code is contained in the **`<script purpose="initialization"...>`** tag. Event wiring begins after all components have been created.

8.6 Usage Code

Usage code is code which references an initialized ECMAScript Component. While this code can often be hand coded, the standard function **`render()`** is often the only necessary usage code. For this reason, calls to **`render()`**are standardized as follows:

```
<script language="javascript" type="text/javascript" purpose="render">
  dateWidget7.render();
</script>
```

9 ECMAScript Component Example

This section contains a non-normative example of an ECMAScript Component.

9.1 ECMAScript Component Class File (SelectNav.ecc)

```
<?xml version="1.0"?>
<!doctype component system "ecmascriptcomponent.dtd">
<component name="com.myCompany.SelectNav" displayname="SelectNav"
  env="client" src="SelectNav.js" version="1.0"
  hint="Creates an HTML select element and populates it with URL's">
  <help path="http://www.mycompany.com/sdihelp/SelectNav.html"/>
  <icon src16="SelectNav.gif"/>
  <meta name="COPYRIGHT" VALUE="Copyright ©1998, MyCompany, Inc."/>
  <property name="name" hint="Enter Name of SelectNav" type="string">
  </property>
  <property name="size" valueset="1|2|3|4" type="number">
  </property>
  <property name="firstLabel" type="string">
  </property>
  <property name="visible" valueset="true|false" type="boolean">
  </property>
  <method name="addLink">
   <parameter name="url" type="string"/>
   <parameter name="label" type="string"/>
  </method>
  <method name="deleteLink">
   <parameter name="linkidx" type="number"/>
  </method>
  <method name="gotoURL">
   <parameter name="linkidx" type="number"/>
  </method>
  <method name="render">
   <parameter name="useOpener" type="boolean"/>
  </method>
  <event name="onRender"/>
```

```
</component>
```

9.2 ECMAScript Include File (SelectNav.js)

```
function com_myCompany_SelectNav (name, size) {
 // "Public" Properties - can be modified by outside world
 this.name = (name) ? name : "SelectNav";
 this.size = (size) ? size : 1;
 // "Private" Properties - should not be modified except by
// object methods
 this.links = new Array();
 this.labels = new Array();
 this.numItems = 0;
 // "Public" Methods - can be called by outside world
 this.addLink = snAddLink;
 this.deleteLink = snDeleteLink;
 this.gotoURL = snGotoURL;
 this.render = snRender;
 // Events
 this.onRender = snOnRender;

 function snAddLink(url, label) {
  this.numItems++;
  this.links[this.numItems-1] = url;
  // Add the label. If no label was given,
  //make the label the same as the link
  this.labels[this.numItems-1] =
     (label != null || label+"" != "undefined") ? label : url;
 } // end snAddLink
 function snDeleteLink(linkidx) {
  var tempLinks = new Array();
  var tempLabels = new Array();
  var count = 0;
  if (linkidx+"" == "undefined" || linkidx == null) {
   this.numItems = 0;
   linkidx = 0;
  }
  // Copy all but the deleted link & label into new arrays
  for(var i=0; i < this.numItems; i++)
  if (i != linkidx) {
   document.write(i + ", " + linkidx);
   tempLinks[count] = this.links[i];
   tempLabels[count] = this.labels[i];
   count++;
  }
  this.numItems = count;
  this.links = tempLinks;
  this.labels = tempLabels;
 } // end snDeleteLink
 function snGotoURL(linkidx) {
  // redirect to the selected location
  window.location.href = this.links[linkidx];
 } // end snGotoURL
 function snRender( useOpener ) {
  // default to using same window instead of opening window
```

```
  if (useOpener+"" == "undefined" || useOpener == null)
   useOpener = false;
   document.write('<FORM NAME="' + this.name + '">');
   if (useOpener)
    document.write('<SELECT NAME="' + this.name
    + '" SIZE=' + this.size +
    ' onChange="window.opener.location.href =
         this.options[this.selectedIndex].value">');
   else
    document.write('<SELECT NAME="' +
    this.name + '" SIZE=' + this.size +
    ' onChange="window.location.href =
         this.options[this.selectedIndex].value">');
    // Write out all of the options in the list
  for (var i=0; i < this.numItems; i++)
   document.write('<OPTION VALUE="' + this.links[i] + '">' +
   this.labels[i]);
   document.write('</SELECT>');
   document.write('</FORM>');
  // Calls event hander
  this.onRender();
 } // end snRender
 // Empty event handler
 function snOnRender() {
 } // end onRender()
} // end SelectNav constructor
```

10 HTML Hosting Example

The following non-normative example demonstrates two components hosted in a single HTML page. The intent of this example is provide a demonstration of fairly complex components which exercise all facets of ECMAScript Components. The following files are listed:

- **example.html** This page has all four types of tool managed code: Initialization, Event Handling, Wiring, and Usage. Instances of two ECMAScript Components are created, and rendered in a form. These components emit the HTML for a select list and a group of radio buttons. Because both of these components have bound value properties an allow for programmatic setting of their “value” they can

- **cJSSelectBox.js, cRadioGroup.js** These are the source files for the two components. They both support a **`render()`** function, a bound property, and the firing of events.

10.1 HTML Host File (example.html)

```
<html><head>
 <script language="javascript" type="text/javascript"
  src="cradiogroup.js"
  purpose="component" classname="radiogroup">
 </script>
 <script language="javascript" type="text/javascript"
  src="cjsselectbox.js"  purpose="component" classname="selectbox">
 </script>
</head><body>
 <script language="javascript" type="text/javascript"
  purpose="eventhandler">
 function aSelectBox1_onChange(propName,oldValue,newValue) {
  if (propName=="value") {
   aRadioGroup1.setValue(newValue);
```

```
  }
}
function aRadioGroup1_onChange(propName,oldValue,newValue) {
  if (propName=="value") {
   aSelectBox1.setValue(newValue);
  }
}
</SCRIPT>

<script language="javascript" type="text/javascript"
  purpose="initializer">
 function _init_aRadioGroup1() {
  _param_ = new Object();
  _param_.value = "";
  _param_.valueOptions = "1;2;3";
  _param_.defaultValue = "2";
  _param_.textOptions = "one;two;three";
  _param_.id = "aRadioGroup1";
  aRadioGroup1 = new com_acme_widgets_RadioGroup(_param_);
}
_init_aRadioGroup1();
 function _init_aSelectBox1() {
  _param_ = new Object();
  _param_.size = 5;
  _param_.value = "";
  _param_.valueOptions = "1;2;3";
  _param_.defaultValue = "2";
  _param_.textOptions = "one;two;three";
  _param_.id = "aSelectBox1";
  aSelectBox1 = new com_acme_widgets_SelectBox(_param_);
}
_init_aSelectBox1();
aSelectBox1.onChange=aSelectBox1_onChange
aRadioGroup1.onChange=aRadioGroup1_onChange
</SCRIPT>

<hr size=2>
<form name="form1">
First component:<BR>
<script language="javascript" type="text/javascript" purpose="render">
aRadioGroup1.render();
</script>
<hr>
Second component:<BR>
<script language="javascript" type="text/javascript" purpose="render">
aSelectBox1.render();
</script>
</form>
<hr size=2>
</body></html>
```

10.2 ECMAScript Include File (cJSSelectBox.js)

```
//-------------------------------------------------
// JavaScript select list component
```

```
// external functions -----------------------------------------
function select_OnChange( oElement ) {
  oJS = window[oElement.name];
  if ( null != oJS.onChange ) {
    oJS.onChange( "value", null, oJS.getValue() );
  }
}
function select_OnBlur( oElement ) {
  oJS = window[oElement.name];
  if ( null != oJS.onBlur ) {
    oJS.onBlur();
  }
}
function select_OnFocus( oElement ) {
  oJS = window[oElement.name];
  if ( null != oJS.onFocus ) {
    oJS.onFocus();
  }
}
// constructor function ---------------------------------------
function com_acme_widgets_SelectBox(params) {
  this.getObj = _select_getObj;
  this.setValue = _select_setValue;
  this.getValue = _select_getValue;
  this.unEncodeList = _select_unEncodeList;
  this.unEncodeJS = _select_unEncodeJS;
  this.id = params.id;
  this.defaultValue = params.defaultValue;
  this.multiple = params.multiple;
  this.render = _select_render;
  this.id = params.id;
  this.size = params.size;
  // now generate the client object
  if ( ""==params.textOptions ) {
    params.textOptions = null;
  }
  if ( ""==params.valueOptions ) {
    params.valueOptions = null;
  }
  this.aValues = this.unEncodeList( params.valueOptions );
  this.aTexts = this.unEncodeList( params.textOptions );
  if ( null == this.aValues ) {
    this.aValues = this.aTexts;
  }
  if ( null == this.aTexts ) {
   this.aTexts = this.aValues;
  }
  // internal functions -----------------------------------------
  function _select_setValue( sNewValue ) {
    obj = this.getObj();
    nOptions = obj.options.length;
    for ( iO = 0; iO < nOptions; iO++ ) {
      oO = obj.options[iO];
      if ( oO.value == sNewValue ) {
```

```
      oO.selected = true;
    } else {
      oO.selected = false;
    }
  }
}
function _select_getValue() {
  obj = this.getObj();
  i = obj.selectedIndex;
  if ( i == -1 ) {
    return null;
  } else {
    return obj.options[i].value;
  }
}
function _select_getObj() {
  if ( null != this.obj ) {
    return this.obj;
  }
  nForms = document.forms.length;
  for ( nF = 0; nF < nForms; nF++ ) {
    oF = document.forms[nF];
    nElements = oF.elements.length;
    for ( nE = 0; nE < nElements; nE++ ) {
      oE = oF[nE];
      if ( oE.name == this.id ) {
        this.obj = oE;
        return oE;
      }
    }
  }
  return null;
}
function _select_unEncodeJS(input) {
  if (null == input)
    return null;
  if (input == "null" )
    return null;
  output = "";
  i=0;
  while (i<input.length) {
    ch = input.charAt(i++);
    if ('\\' == ch) {
      ch = input.charAt(i++);
      if ('n' == ch) {
        output += '\n';
      } else if (';' == ch) {
        output += ';';
      } else if ('r' == ch) {
        //supress this character
      } else if ('t' == ch) {
        output += '\t';
      } else {
        output += ch;
```

```
      }
    } else {
      output += ch;
    }
  }
  return output;
}
function _select_unEncodeList( sList ) {
  aRes = null;
  if ( null != sList && ""!=sList ) {
    aRes = new Array();
    iStart = 0;
    iIndex = 0;
    j=0;
    while (j<sList.length) {
      ch = sList.charAt(j++);
      if ('\\' == ch) {
        ch = sList.charAt(j++);
      } else {
        if ( ';' == ch ) {
          sRaw = sList.substring( iStart, j-1 );
          aRes[iIndex++] = this.unEncodeJS( sRaw );
          iStart = j;
        }
      }
    }
    if ( iStart != j ) {
      sRaw = sList.substring( iStart, j );
      aRes[iIndex++] = this.unEncodeJS( sRaw );
    }
  }
  return aRes;
}
function _select_render() {
  if ( this.aValues == null ) {
    document.write( "JSSelectBox error:
      Value Options or TextOptions must be specified." );
  } else {
    sName = " name='"+this.id+"'";
    sSize = " size='"+this.size+"'";
    sOnChange = " onchange='select_OnChange(this)'";
    sOnBlur = " onblur='select_OnBlur(this)'";
    sOnFocus = " onfocus='select_OnFocus(this)'";
    sMultiple = "";
    if ( null != this.multiple &&
     "" != this.multiple && this.multiple == "true" ) {
      sMultiple = " multiple";
    }
  }
  document.write( "<select" + sName + sSize + sOnChange +
     sOnBlur + sOnFocus + sMultiple + ">\n" );
  if ( this.aValues != null ) {
    for ( iIndex = 0; iIndex < this.aValues.length; iIndex++ ) {
      sDefault = ( null != this.defaultValue &&
```

```
          this.defaultValue == this.aValues[iIndex]) ? " SELECTED" : "";
        document.write("<option value='"+
         this.aValues[iIndex]+"'" + sDefault + ">" + this.aTexts[iIndex]
          + "\n" );
      }
    }
    document.writeln( "</select>\n" );
  }
}
```

10.3 ECMAScript Include File (cRadioGroup.js)

```
//-------------------------------------------------
// JavaScript radio group component

// external functions ----------------------------------------
function radio_OnClick( oElement ) {
  oJS = window[oElement.name];
  if ( null != oJS.onChange ) {
    oJS.onChange( "value", null, oJS.getValue() );
  }
}
function radio_OnBlur( oElement ) {
  oJS = window[oElement.name];
  if ( null != oJS.onBlur ) {
    oJS.onBlur();
  }
}
function radio_OnFocus( oElement ) {
  oJS = window[oElement.name];
  if ( null != oJS.onFocus ) {
    oJS.onFocus();
  }
}
// constructor function ----------------------------------------
function com_acme_widgets_RadioGroup(params) {
  this.getButtons = _radio_getButtons;
  this.setValue = _radio_setValue;
  this.getValue = _radio_getValue;
  this.unEncodeList = _radio_unEncodeList;
  this.unEncodeJS = _radio_unEncodeJS
  this.id = params.id;
  this.render = _radio_render;
  if ( ""==params.textOptions ) {
    params.textOptions = null;
  }
  if ( ""==params.valueOptions ) {
    params.valueOptions = null;
  }
  this.aValues = this.unEncodeList( params.valueOptions );
  this.aTexts = this.unEncodeList( params.textOptions );
  if ( null == this.aValues ) {
    this.aValues = this.aTexts;
  }
  if ( null == this.aTexts ) {
   this.aTexts = this.aValues;
```

```
}
this.defaultValue = params.defaultValue;
this.id = params.id;

// internal functions ------------------------------------------
function _radio_setValue( sNewValue ) {
  obj = this.getButtons();
  for (i in obj) {
    if ( obj[i].value == sNewValue ) {
      obj[i].checked = true;
    }
  }
}
function _radio_getValue() {
  obj = this.getButtons();
  for (i in obj) {
    if ( obj[i].checked ) {
      return obj[i].value;
    }
  }
  return null;
}
function _radio_getButtons() {
  if ( null != this.obj ) {
    return this.obj;
  }
  nForms = document.forms.length;
  for ( nF = 0; nF < nForms; nF++ ) {
    oF = document.forms[nF];
    oRet = oF[this.id];
    if ( null != oRet ) {
      this.obj = oRet;
      return oRet;
     }
  }
  return null;
}
function _radio_unEncodeJS(input) {
  if (null == input)
    return null;
  if (input == "null" )
    return null;
  output = "";
  i=0;
  while (i<input.length) {
    ch = input.charAt(i++);
    if ('\\' == ch) {
      ch = input.charAt(i++);
      if ('n' == ch) {
        output += '\n';
      } else if (';' == ch) {
        output += ';';
      } else if ('r' == ch) {
        //supress this character
```

```
        } else if ('t' == ch) {
          output += '\t';
        } else {
          output += ch;
        }
      } else {
        output += ch;
      }
    }
    return output;
  }
  function _radio_unEncodeList( sList ) {
    aRes = null;
    if ( null != sList && ""!=sList ) {
      aRes = new Array();
      iStart = 0;
      iIndex = 0;
      j=0;
      while (j<sList.length) {
        ch = sList.charAt(j++);
        if ('\\' == ch) {
          ch = sList.charAt(j++);
        } else {
          if ( ';' == ch ) {
            sRaw = sList.substring( iStart, j-1 );
            aRes[iIndex++] = this.unEncodeJS( sRaw );
            iStart = j;
          }
        }
      }
      if ( iStart != j ) {
        sRaw = sList.substring( iStart, j );
        aRes[iIndex++] = this.unEncodeJS( sRaw );
      }
    }
    return aRes;
  }
  function _radio_render() {
    if ( this.aValues == null ) {
      document.write( "RadioGroup error: Value Options or TextOptions must
be specified." );
    } else {
      sName = " name='"+this.id+"'";
      sOnClick = " onclick='radio_OnClick(this)'";
      sOnBlur = " onblur='radio_OnBlur(this)'";
      sOnFocus = " onfocus='radio_OnFocus(this)'";
      for( i = 0; i < this.aValues.length; i++ ) {
        sValue = " value='"+this.aValues[i]+"'";
        sChecked = (this.defaultValue == this.aValues[i]) ? " CHECKED" :
"";
        document.write( "<input type='radio' " + sName + sValue + sOnClick
+ sOnBlur + sOnFocus + sChecked + ">" + this.aTexts[i] );
      }
    }
```

```
    }
}
```

11 ECMAScript Component DTD

```
<!—ecmascriptcomponent.dtd -->
<!element component (
        help?
        | icon?
        | customizer?
        | visual?
        | uses*
        | meta*
        | property*
        | method*
        | event*)*>
<!attlist component
env (client | server | both | either) "both"
src CDATA #required
name CDATA #required
hint CDATA #implied
version CDATA #implied
needsform (yes | no) "no"
ishidden (yes | no) "no"
displayname CDATA #implied>

<!-- help -->
<!element help empty>
<!attlist help
path CDATA #required>

<!-- icon -->
<!element icon empty>
<!attlist icon
src16 CDATA #required
src32 CDATA #implied>

<!-- customizer -->
<!element customizer empty>
<!attlist customizer
type (ecmascript | html) "ecmascript"
class CDATA #required>

<!-- visual -->
<!element visual empty>
<!attlist visual
type (ecmascript | html) "ecmascript"
class CDATA #required>

<!-- uses -->
<!element uses empty>
<!attlist uses
file CDATA #required>

<!-- meta -->
<!element meta empty>
<!attlist meta
name CDATA #required
value CDATA #required>

<!-- property -->
<!element property (get | set | propertyeditor)*>
<!attlist property
name CDATA #required
displayname CDATA #implied
hint CDATA #implied
```

```
valueset CDATA #implied
type (string | number | boolean | unknown) "string"
env (client | server | both) "both"
defaultvalue CDATA #implied
runtimeaccess (readwrite | readonly | writeonly | none) "readwrite"
isinitializable (yes | no) "yes"
isexpert (yes | no) "no"
isbound (yes | no) "no"
isdefault (yes | no) "no">

<!-- property get -->
<!element get empty>
<!attlist get
name CDATA #required>

<!-- property set -->
<!element set empty>
<!attlist set
name CDATA #required>

<!-- property editor -->
<!element propertyeditor empty>
<!attlist propertyeditor
type (ecmascript | html) "ecmascript"
class CDATA #required>

<!-- method -->
<!element method (parameter)*>
<!attlist method
name CDATA #required
displayname CDATA #implied
hint CDATA #implied
env (client | server | both) "both"
isexpert (yes | no) "no">

<!-- method parameter -->
<!element parameter empty>
<!attlist parameter
name CDATA #required
displayname CDATA #implied
hint CDATA #implied
valueset CDATA #implied
type (string | number | boolean | unknown) "string">

<!-- event -->
<!element event empty>
<!attlist event
name CDATA #required
displayname CDATA #implied
hint CDATA #implied
isexpert (yes | no) "no"
isdefault (yes | no) "no">
```

12 Appendix I: Relationship to RDF

This non-normative appendix, which will change over time, describes the relationship between ECMAScript Components and Resource Description Framework (RDF).

RDF is a general mechanism for describing structured related data. XML grammars can be used to represent RDF resource descriptions. According to the RDF Model and Syntax Specifications:

> *The broad goal of RDF is to define a mechanism for describing resources that makes no assumptions about a particular application domain, nor defines (a priori) the semantics of any application domain. The definition of the mechanism should be domain neutral, yet the mechanism should be suitable for describing information about any domain.*

While RDF is particularly useful in cases where the structure of the data is not fixed, its generality means that it could be used to describe any data structure and content. Therefore, while RDF could potentially be used to model an ECMAScript component's metadata, the ECMAScript Component specification does not employ RDF to describe component metadata for two principle reasons.

First, ECMAScript Component metadata is much easier to work with than RDF, even in its abbreviated forms. Consequently, usage of RDF would go against one of the fundamental goals of ECMAScript Components: to be easily usable by ECMAScript scripters. An ECMAScript scripter, which is typically distinct from a lower-level “programmer”, will often have a working knowledge limited to that of ECMAScript and HTML; leveraging this existing knowledge base will enable the ECMAScript Component standard to be more easily adopted by this community. As a result, the metadata schema uses the “vernacular” that a scripter will already feel comfortable with, such as <property>, <method>, and <event>. It follows then that, because of the nature of the scripting community, adding the requirement to learn RDF prior to creating or utilizing ECMAScript Components would have an impact on the usefulness of this standard to its target audience.

Second, generality is one of the main benefits of RDF, but this generality has less practical value within the context of describing ECMAScript Component metadata. Tools which create, edit, and interpret ECMAScript Component descriptions will necessarily need to know the appropriate structure of the metadata.

The TC39 working group will continue to evaluate RDF and other possible representations of ECMAScript Component metadata for future versions of the ECMAScript Component specification.

Printed copies can be ordered from:

ECMA
114 Rue du Rhône
CH-1204 Geneva
Switzerland

Fax: +41 22 849.60.01
Internet: documents@ecma.ch

Files can be downloaded from our FTP site, **ftp.ecma.ch**. This Standard is available from library **ECMA-ST** as a compacted, self-expanding file in MSWord 6.0 format (file Exxx-DOC.EXE) and as an Acrobat PDF file (file Exxx-PDF.PDF). File Exxx-EXP.TXT gives a short presentation of the Standard.

Our web site, http://www.ecma.ch, gives full information on ECMA, ECMA activities, ECMA Standards and Technical Reports.

ECMA

114 Rue du Rhône
CH-1204 Geneva
Switzerland