

# Web Forms 2.0

## Working Draft — 1 September 2005



### This version:

<http://www.whatwg.org/specs/web-forms/2005-09-01/>

### Latest version:

<http://www.whatwg.org/specs/web-forms/current-work/>

### Previous versions:

<http://www.whatwg.org/specs/web-forms/2005-07-03/> (diffs)

<http://www.whatwg.org/specs/web-forms/2005-04-11-call-for-comments/>  
(diffs)

<http://www.w3.org/Submission/2005/SUBM-web-forms2-20050411/>

<http://www.whatwg.org/specs/web-forms/2005-01-28-call-for-comments/>

<http://www.whatwg.org/specs/web-forms/2004-12-10-call-for-comments/>

<http://www.whatwg.org/specs/web-forms/2004-06-27-call-for-comments/>

<http://www.hixie.ch/specs/html/forms/web-forms-3>

<http://www.hixie.ch/specs/html/forms/web-forms-2>

<http://www.hixie.ch/specs/html/forms/xforms-basic-1>

<http://lists.w3.org/Archives/Member/w3c-archive/2003Sep/att-0014/hfp.html> (W3C member-only link)

### Editor:

Ian Hickson, Opera Software, [ian@hixie.ch](mailto:ian@hixie.ch)

© Copyright 2004, 2005 Apple Computer, Inc., Mozilla Foundation, and Opera Software ASA.  
You are granted a license to use, reproduce and create derivative works of this document.

---

## Abstract

This specification defines Web Forms 2.0, an extension to the forms features found in HTML 4.01's Forms chapter and the corresponding DOM 2 HTML interfaces. Web Forms 2.0 applies to both HTML and XHTML user agents. It provides new strongly-typed input fields, new attributes for defining constraints, a repeating model for declarative repeating of form sections, new DOM interfaces, new DOM events for validation and dependency tracking, and XML submission and initialization of forms. It also standardises and codifies existing

practice in areas that have not been previously documented, and clarifies some of the interactions of HTML form controls and CSS.

HTML4, XHTML1.1, and the DOM are thus extended in a manner that has a clear migration path from existing HTML forms, leveraging the knowledge authors have built up with their experience with HTML so far.

## Status of this document

This document is the result of a loose collaboration between interested parties in the context of the [Web Hypertext Application Technology Working Group](#).

This is an archive copy of a working draft of Web Forms 2.0. If you wish to make comments regarding this document, please send them to [whatwg@whatwg.org](mailto:whatwg@whatwg.org). All feedback is welcome.

**This document is in the very final stages and will very shortly become a call for implementations.**

To find the latest version of this specification, please follow the "Latest version" link above.

## Table of contents

- [1. Introduction](#)
  - [1.1. Scope](#)
  - [1.2. Relationship to HTML](#)
  - [1.3. Relationship to XHTML](#)
  - [1.4. Relationship to the W3C DOM](#)
  - [1.5. Relationship to XForms](#)
  - [1.6. Relationship to XForms Basic](#)
  - [1.7. Relationship to CSS](#)
  - [1.8. Missing features](#)
  - [1.9. Conformance requirements](#)
  - [1.10. Terminology](#)
  - [1.11. Extensibility](#)
  - [1.12. Security](#)
- [2. Extensions to form control elements](#)
  - [2.1. Introduction for authors](#)
  - [2.2. Existing controls](#)

### [2.3. Changes to existing controls](#)

### [2.4. Extensions to the `input` element](#)

#### [2.4.1. Ranges](#)

#### [2.4.2. Precision](#)

### [2.5. Extensions to existing attributes](#)

### [2.6. The `pattern` attribute](#)

### [2.7. The `required` attribute](#)

### [2.8. The `form` attribute](#)

### [2.9. The `autocomplete` attribute](#)

### [2.10. The `autofocus` attribute](#)

### [2.11. The `inputmode` attribute](#)

### [2.12. The `datalist` element and the `list` attribute](#)

### [2.13. The `output` element](#)

### [2.14. Extensions to the `textarea` element](#)

### [2.15. Extensions to file upload controls](#)

### [2.16. Extensions to the `form` element](#)

### [2.17. Extensions to the submit buttons](#)

### [2.18. Handling unexpected elements and values](#)

## [3. The repetition model for repeating form controls](#)

### [3.1. Introduction for authors](#)

#### [3.1.1. More features](#)

#### [3.1.2. Suggestions for authors](#)

#### [3.1.3. What the repetition model can't do](#)

### [3.2. Definitions](#)

#### [3.2.1. Repetition templates](#)

#### [3.2.2. Repetition blocks](#)

### [3.3. New form controls](#)

### [3.4. The `repeat-min` and `repeat-max` attributes](#)

### [3.5. Event interface for repetition events](#)

### [3.6. The repetition model](#)

#### [3.6.1. Addition](#)

#### [3.6.2. Removal](#)

#### [3.6.3. Movement of repetition blocks](#)

#### [3.6.4. Initial repetition blocks](#)

#### [3.6.5. Notes for assistive technologies](#)

### [3.7. Examples](#)

#### [3.7.1. Repeated rows](#)

#### [3.7.2. Nested repeats](#)

## [4. The forms event model](#)

### [4.1. The `click` event and `input` controls](#)

### [4.2. The `change` and `input` events](#)

### [4.3. Events to enable simpler dependency tracking](#)

#### [4.4. Form validation](#)

#### [4.5. Receiving the results of form submission](#)

#### [4.6. The `DOMControlValueChanged` event](#)

### [5. Form submission](#)

#### [5.1. Successful form controls](#)

#### [5.2. Handling characters outside the submission character encoding](#)

##### [5.2.1. The `\_charset\_` field](#)

#### [5.3. `application/x-www-form-urlencoded`](#)

#### [5.4. `application/x-www-form+xml`: XML submission](#)

#### [5.5. `text/plain`](#)

#### [5.6. Submitting the encoded form data set](#)

##### [5.6.1. For `http:` actions](#)

##### [5.6.2. For `ftp:` actions](#)

##### [5.6.3. For `data:` actions](#)

##### [5.6.4. For `file:` actions](#)

##### [5.6.5. For `mailto:` actions](#)

##### [5.6.6. For `smsto:` and `sms:` actions](#)

##### [5.6.7. For `javascript:` actions](#)

### [6. Fetching data from external resources](#)

#### [6.1. Filling `select` elements](#)

#### [6.2. Seeding a form with initial values](#)

### [7. Extensions to the HTML Level 2 DOM interfaces](#)

#### [7.1. Additions specific to the `HTMLFormElement` interface](#)

#### [7.2. Additions specific to the `HTMLSelectElement` interface](#)

#### [7.3. The `HTMLDataListElement` interface](#)

#### [7.4. Changes to the `HTMLOptionElement` interface](#)

#### [7.5. Additions specific to the `HTMLFieldsetElement` interface](#)

#### [7.6. The `HTMLOutputElement` interface](#)

#### [7.7. Validation APIs](#)

#### [7.8. New DOM attributes for new content attributes](#)

#### [7.9. Additions specific to the `HTMLInputElement` interface](#)

#### [7.10. The `defaultValue` DOM attribute](#)

#### [7.11. Labels](#)

#### [7.12. Firing change events](#)

#### [7.13. Repetition interfaces](#)

### [8. Presentation and rendering](#)

#### [8.1. Styling form controls](#)

#### [8.2. Relation to CSS selectors](#)

#### [8.3. Interaction of the form processing model with CSS](#)

#### [8.4. Interaction of the form processing model with SMIL](#)

[A. XHTML module definition](#)

[B. Attribute summary](#)

[C. Deprecated features](#)

[D. Requirements for declaring interoperability](#)

[References](#)

[Acknowledgements](#)

---

## 1. Introduction

This is an update to the forms features found in HTML 4.01's [Forms chapter](#), which are informally referred to as Web Forms 1.0.

Authors have long requested enhancements to HTML4 to support some of their more common needs. Such requests in mailing lists and other forums were examined, and from these sources a set of requirements and design goals were derived:

- Backwards compatibility (where possible).
- Ease of authoring for authors who are familiar with commonly used languages such as HTML and ECMAScript but have limited knowledge about XML, data models, etc.
- Basic data typing, providing new controls for commonly used types so that authors do not need to repeatedly design complicated widgets such as calendars.
- Simpler validation on the client side (while recognizing that server side validation will still be required), with declarative solutions for the common cases, but strong DOM support so that less common cases can easily be handled using scripting.
- Dynamic addition of fields (repeating structures) on the client side without scripting.
- XML submission (although not necessarily arbitrarily structured XML submission).
- The ability to initialize forms from external data sources, so that authors do

not have to dynamically rewrite the form content itself to prefill forms, but can instead use static pages with scripts that dynamically generate only the data part.

- This specification should be implementable in full on devices with limited resources.

Not all the desired features have been included in this specification. Future versions may be introduced to address further needs.

This specification does not describe the complete behaviour of an HTML or XHTML user agent. Readers are expected to refer to the existing specifications for the definitions of features that this specification does not change.

## 1.1. Scope

This specification is limited specifically to incremental improvements to existing wide-spread technologies, namely HTML4 and the DOM, as implemented by browsers prevalent in 2004. It is also intended to be a small step, implementable without overwhelming effort.

Large sweeping changes or new markup languages are therefore out of scope for this specification.

## 1.2. Relationship to HTML

This specification clarifies and extends the semantics put forth in [\[HTML4\]](#) for form controls and form submission. It is expected to be implemented in ordinary HTML user agents alongside existing forms technology, and indeed, some of the features described in this draft have been implemented by user agents as ad-hoc, non-standard extensions for many years due to strong market need.

## 1.3. Relationship to XHTML

This specification can also be viewed as an extension to [\[XHTML1\]](#). In particular, some of the features added in this module only apply to XHTML documents; for example, features allowing mixed namespaces.

## 1.4. Relationship to the W3C DOM

This specification clarifies and extends the semantics put forth in [\[DOM2HTML\]](#) for the form control interfaces. These extensions are expected to be

implemented in HTML and XHTML user agents that support the DOM.

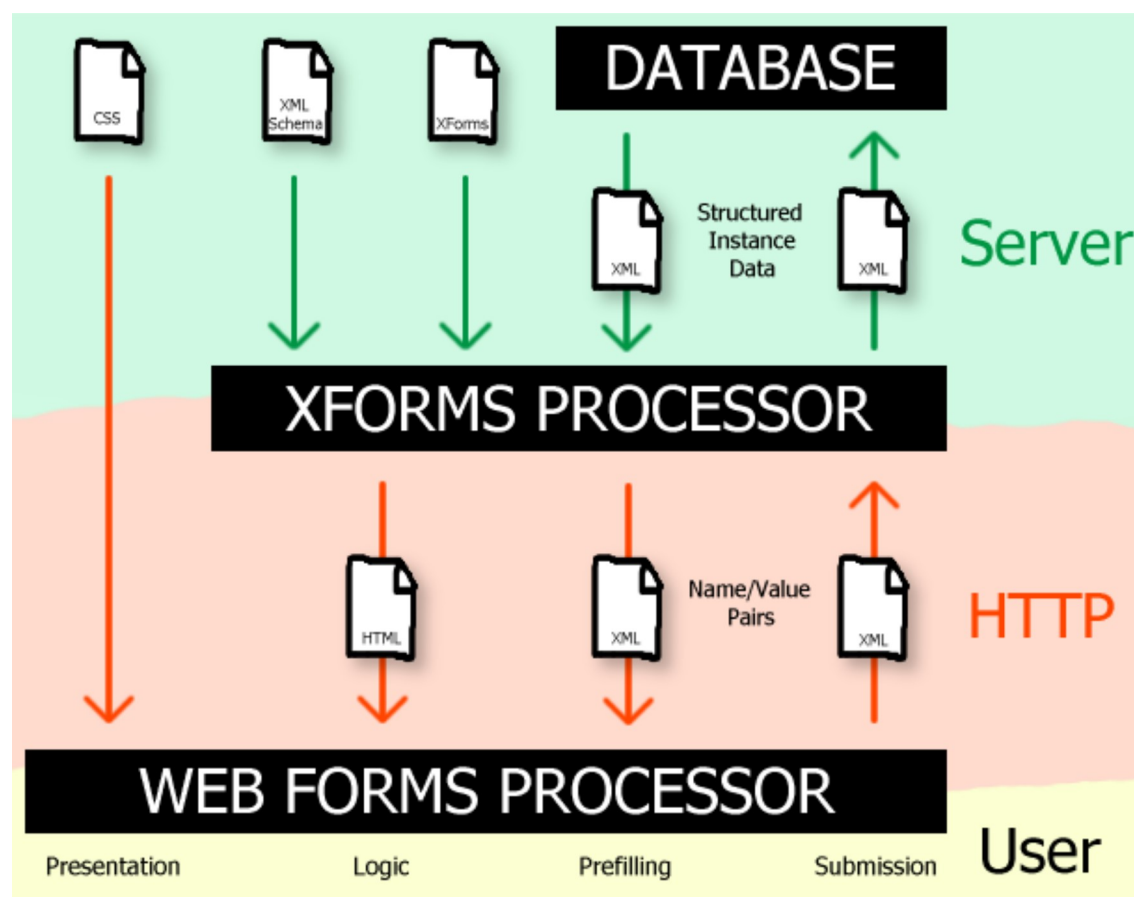
## 1.5. Relationship to XForms

***Note: This section is aimed at XForms authors and implementors. If you do not plan to use XForms, you may prefer to skip ahead to the next section. Knowledge of XForms is not required to use Web Forms.***

This specification is in no way aimed at replacing XForms 1.0 [\[XForms\]](#), nor is it a subset of XForms 1.0.

XForms 1.0 is well suited for describing business logic and data constraints. Web Forms 2.0 aims to simplify the task of transforming XForms 1.0 systems into documents that can be rendered on HTML Web browsers that do not support XForms.

In this transformation model, the XForms processor is a server-side process that converts XForms and XML Schema documents, according to the XForms specification, into HTML and Web Forms documents, which are then processed by the client side Web Forms processor, along with a style sheet for presentation.



The structured XML instance data stored on the server side (e.g. in a database) is converted by the XForms processor into name/value pairs that are then used by the UA to prefill the form. Submission follows the opposite path, with the UA generating name/value pairs and sending them to the XForms processor on the server, which converts them back into structured XML for storage or further processing.

In order to simplify this transformation process, this specification attempts to add some of the functionality of XForms with a minimum impact on the existing, widely implemented forms model. Where appropriate, backwards compatibility, ease of authoring, and ease of implementation have been given priority over theoretical purity.

The following features of XForms have *not* been addressed:

- The separation of the instance data model, data typing, field interdependencies, and submission information from the content model and interface elements.
- The ability to create arbitrary XML fragments to be filled in before submission.
- The ability to edit local XML files directly. (While technically not defined by the XForms 1.0 specification, UAs have generally implemented such a feature since it is easy to extend the XForms model in that way.)
- Compound data type definitions (schemas).

Many of the less-used features that XForms supports using declarative syntax are, in this specification, handled by using scripting. Some new interfaces are introduced to simplify some of the more tedious tasks.

## 1.6. Relationship to XForms Basic

This specification is unrelated to the XForms Basic profile.

***Note: A previous version of this draft was called "XForms Basic". This name has been changed so as to avoid confusion with the similarly named draft from the W3C.***

## 1.7. Relationship to CSS

This specification does not extend CSS, but it does clarify some of the interactions between HTML's form features and CSS.



## 1.8. Missing features

This draft does not address all needs. In addition to the features of XForms that have not been addressed (see above), the following features were considered but rejected for this version of the specification:

- Digital signatures for submissions. This is currently not covered by this specification due to patent concerns. However, it would still be considered for future inclusion if suggestions of how to support it without infringing on known patents were provided.
- DOM interfaces for the creation of new controls that are still able to interact with form submission. This will probably be addressed by the [Web Controls 1.0 specification](#).
- Elements or properties to create tabbed interfaces or "wizard" interfaces. These needs will probably be addressed by the [Web Applications 1.0 specification](#).
- A rich text editing or HTML editing control. This will probably be addressed by the [Web Applications 1.0 specification](#).
- A grid or spreadsheet editing control. This will probably be addressed by the [Web Applications 1.0 specification](#).
- A declarative way of specifying that one list should filter the view of a second list. This need may be addressed in a future version of this specification.

## 1.9. Conformance requirements

Conformance to this specification is defined for user agents (UAs, implementations) and documents (authors, authoring tools, markup generators). Clauses specify whether they apply to user agents or documents.

User agents could include graphical Web browsers, voice-based mobile devices, automated agents, content indexing robots, and inference tools. In certain user agents, it may be impossible to determine whether a particular conformance criteria is followed or not. For instance, whether indexing robots mark the first option in a `select` element as selected or not is not detectable. When it is impossible to tell if a UA complies with a particular conformance requirement, that UA is exempt from conforming to that requirement.

Authoring tools and markup generators are conformant if they only produce conformant documents.

As well as sections marked as non-normative, all diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in the normative parts of this document are to be interpreted as described in [\[RFC2119\]](#). For readability, these words do not appear in all uppercase letters in this specification.

This specification includes by reference the form-related parts of the HTML4, XHTML1.1, DOM2 HTML, DOM3 Core, and DOM3 Events specifications ([\[HTML4\]](#), [\[XHTML1\]](#), [\[DOM2HTML\]](#), [\[DOM3CORE\]](#), [\[DOM3EVENTS\]](#)). Compliant UAs must implement all the forms-related requirements of those specifications, except those modified by this specification, to claim compliance with this one. Implementations may optionally implement only one of HTML4 and XHTML1.1.

Implementations and documents must comply to the W3C Character Model specification. [\[CHARMOD\]](#)

Implementations that do not support scripting (or which have their scripting features disabled) are exempt from supporting the events and DOM interfaces mentioned in this specification. For the parts of this specification that are defined in terms of an events model or in terms of the DOM, such user agents must still act as if events and the DOM were supported.

***Note: Scripting can form an integral part of an application. User agents that do not support scripting, or that have scripting disabled, might be unable to fully convey the author's intent.***

This specification introduces attributes for setting the maximum size or range of certain values. While user agents should support all possible values, there may be implementation-specific limits.

HTML documents that use the new features described in this specification and that are served over HTTP must be sent as `text/html` and must use the following DOCTYPE: `<!DOCTYPE html>`.

XML documents using elements from the XHTML namespace that use the new features described in this specification and that are served over HTTP must be sent using an XML MIME type such as `application/xml` or `application/xhtml+xml` and must not be served as `text/html`. [\[RFC3023\]](#)

These XML documents may contain a `DOCTYPE` if desired, but this is not required.

**Note: Documents that use the new features described in this specification cannot be strictly conforming XHTML1.1 or HTML4 documents, since they contain features not defined in those specifications.**

## 1.10. Terminology

This specification refers to both HTML and XML attributes and DOM attributes, often in the same context. When it is not clear which is being referred to, they are referred to as **content attributes** for HTML and XML attributes, and **DOM attributes** for those from the DOM. Similarly, the term "properties" is used for both ECMAScript object properties and CSS properties. When these are ambiguous they are simply qualified as object properties and CSS properties respectively.

Generally, when the specification states that a feature applies to HTML or XHTML, it also includes the other. When a feature specifically only applies to one of the two languages, it is called out explicitly, as in:

*Similarly, `form` elements in XHTML may now be nested (this does not apply to HTML).*

Unless otherwise stated, all XML elements defined or mentioned in this specification are in the `http://www.w3.org/1999/xhtml` namespace, and all attributes defined or mentioned in this specification have no namespace (they are in the per-element partition).

(There are elements from other namespaces in this specification, in particular the XML submission format uses the `http://n.whatwg.org/formdata` namespace.)

The term **form control** refers to `input`, `output`, `select`, `textarea` and `button` elements. It does not include `form`, `label`, `datalist`, `option`, `optgroup`, or `fieldset` elements.

Form controls are **valid** when they comply with their constraints, and **invalid** when they don't. This is distinct from whether form controls have their `willValidate` attribute set to true, which relates to whether they will be validated, or whether they are [\*successful\*](#), which relates to whether they will be submitted.

The terms URI and IRI in normative contexts are used as defined by [\[RFC3986\]](#) and [\[RFC3987\]](#).

When a comparison is said to be **case-insensitive**, the comparison must be performed using case folding, as described in Unicode. See Unicode 4.0, section 5.18 "Case Mappings", subsection "Caseless Matching". [\[UNICODE\]](#)

## 1.11. Extensibility

Vendor-specific proprietary extensions to this specification are strongly discouraged. Documents must not use such extensions, as doing so reduces interoperability and fragments the user base, allowing only users of specific user agents to access the content in question.

If markup extensions are needed, they should be done using XML, with elements or attributes from custom namespaces. If DOM extensions are needed, the members should be prefixed by vendor-specific strings to prevent clashes with future versions of this specification. Extensions must be defined so that the use of extensions does not contradict nor cause the non-conformance of functionality defined in the specification.

For example, while strongly discouraged to do so, an implementation "Foo Browser" could add a new DOM attribute "fooTypeTime" to a control's DOM interface that returned the time it took the user to select the current value of a control (say). On the other hand, defining a new control that appears in a form's `elements` array would be in violation of the above requirement, as it would violate the definition of `elements` given in this specification.

User agents must treat elements and attributes that they do not understand as semantically neutral; leaving them in the DOM (for DOM processors), and styling them according to CSS (for CSS processors), but not inferring any meaning from them.

## 1.12. Security

The delivery of Web Forms 2 documents to the user from a remote host and the submission of data from the user to a remote host may be performed using a number of different protocols, and therefore no specific statements can be made regarding the security of those operations.

In general, authors are urged to use a secure transport layer such as TLS when information of a confidential nature is to be transmitted.

On the client side, implementors must be aware of a number of potential attacks. Since it is relatively easy for a hostile Web site to trick users into loading hostile content, for example by sending e-mails claiming to include links

to photos of naked girls, users must be confident that a hostile site cannot access confidential information, perform denial-of-service attacks, or hijack the client's host to perform actions on behalf of the user that the user may not approve of.

Confidential information can be stored in several places. Documents from other servers loaded into other browsing contexts (e.g. other windows), documents from other servers that the hostile page has caused to be loaded (of particular concern being pages that include user-specific information using out-of-band authentication and/or authorisation information such as HTTP cookies, HTTP authentication, or the origin host), files on the local system, as well as details of the user's configuration are all potential sources of confidential information.

User agents must therefore implement security mechanisms to block cross-domain accesses (where local files are considered a separate domain). Such mechanisms are referred to as **cross-domain scripting security** mechanisms. Unfortunately, since it is difficult to predict exactly what attack vectors may exist in such a complex system, and in particular because it depends on the exact feature set of the implementation, this specification does not define the exact mechanism that must be implemented.

In practice user agents implement quite comprehensive cross-domain scripting security mechanisms. Implementation experience has shown that such security mechanisms must, at a minimum, prevent scripts originating from a site at one domain from accessing the properties and methods of any object (in particular, DOM nodes) associated with a page from another domain. Typically, such an access would cause an exception to be raised.

Denial of service attacks are naturally hard to prevent, since they frequently are hard to distinguish from legitimate behaviour. Implementors are encouraged to set arbitrary (although high) limits on what an author can do. For instance, user agents might place a limit on the length of the regular expression pattern allowed in the `pattern` attribute, if a long expression could be made to take unacceptably long to execute.

Implementations are also asked to consider how otherwise-legitimate UI could be abused by a hostile page. Naturally, since implementations are not restricted in how they implement their interface, no specific guidelines can be given. One example, however, would be the `mailto:` submission feature. Since a script can artificially submit a form, it is important that the UA not cause each submission to create a new mail window, since this would allow authors to overwhelm the user with windows containing author-specified text, which could act as both a denial-of-service attack, and an annoying advertising technique.

Finally, user agent implementors should prevent pages and scripts in those

pages from performing potentially harmful or embarrassing actions on behalf of the user without the user's knowledge.

For example, it is recommended that user agents limit the ports to which forms may be submitted, excluding, in particular, ports of well-known protocols like SMTP or telnet. The SMTP port in particular has been used by hostile pages in the past as a target of form submissions for the relaying of spam by unsuspecting users.

Certain actions, including submitting a form to a third-party site and making HTTP GET requests to remote sites (both of which would be blind attacks, assuming the UA implements a [cross-domain scripting security](#) mechanism) have been historically allowed, and many sites depend on these features for quite legitimate uses. User agents should allow them.

Servers therefore must also consider security. Servers should never perform non-idempotent actions in response to GET requests, as discussed by the HTTP specification. Servers should also check the `Referer` header to ensure that only requests from trusted hosts are honoured.

Servers should also consider the client to be untrusted, since in most scenarios requests can be made to hosts by hostile parties directly, bypassing any security logic included in the page nominally intended to perform the submission. Thus servers should perform validation on all submitted data, whether such validation is expected to be performed on the client or not.

Further specific security considerations are called out where relevant.

## 2. Extensions to form control elements

This section describes how Web Forms 2.0 expands the traditional HTML form model to support new types and features.

### 2.1. Introduction for authors

*This subsection is not normative.*

One of the big additions to the Web Forms model introduced with this specification is primitive type and validity checking.

Authors can use these new features in various ways. To indicate that a form control expects a particular type of input, authors can specify the types using the `type` attribute:

```
<label>E-mail address: <input type="email" name="addr"></label>
<label>Start date: <input type="date" name="start"></label>
```

To mark a field as required, the required attribute can be used:

```
<label>Quantity: <input type="number" required="required" name="qty"></label>
```

To set the range of values that are allowed, the min and max attributes can be used:

```
<label>Meeting time: <input type="time" min="09:00" max="17:00" name="mt"></label>
```

Once such constraints have been specified, the user agent will automatically guide the user through any errors he may have made before allowing the form to be submitted.

Authors can hook into this validation system with their scripts. There are several ways to do this.

At any point, scripts can check a control's validity DOM attribute for up to date information on whether a control is valid:

```
with (document.forms[0]) {
  if (qty.validity.valueMissing) {
    // the quantity field is required but not filled in
  } else if (qty.validity.typeMismatch) {
    // the quantity field is filled in, but it is not a number
  }
}
```

An author can explicitly set a control as being invalid ("invalid" means that the control's value is not acceptable):

```
var myControl = document.forms[0].addr;
if (myControl.value == 'a@b.c') {
  myControl.setCustomValidity('You must enter your real address.');
```

Authors can also override the normal user agent error reporting behaviour by hooking into the invalid event:

```
<label>Home page: <input type="url" name="hp" required="required"
oninvalid="alert('You must enter a valid home page');"
</label>
```

## 2.2. Existing controls

HTML `input` elements use the `type` attribute to specify the data type. In [\[HTML4\]](#), the types (as seen by the server) are as follows:

**text**

A free-form text field, nominally free of line breaks.

**password**

A free-form text field for sensitive information, nominally free of line breaks.

**checkbox**

A set of zero or more values from a predefined list (in the limiting case of the list only containing one value, this is equivalent to a boolean).

**radio**

An enumerated value.

**submit**

An enumerated value, with the extra semantic that it must be the last value selected and initiates form submission.

**file**

An arbitrary file with a MIME type and optionally a file name.

**image**

A coordinate, relative to a particular image's size, with the extra semantic that it must be the last value selected and initiates form submission.

**hidden**

An arbitrary string that is not normally displayed to the user.

In addition, HTML also provides a few alternate elements that convey typing semantics similar to the above types, but use different content models:

**select**

An enumerated value, much like the radio type.

**select multiple**

A set of zero or more values from a predefined list, much like the checkbox type.

**textarea**

A free-form text field, nominally with no line break restrictions.

**button**

An enumerated value, with the extra semantic that it must be the last value selected and initiates form submission, much like the submit type but with a richer content model.

The difference between the checkbox and radio types and their `select` and



`select` `multiple` counterparts is that for the `select` variants the values are only available through a single composite control, whereas for the `checkbox` and `radio` types the controls representing each value may be individually placed around the document.

There are also two button types (available on both `input` and `button` elements) that are never submitted: `button` and `reset`.

This specification includes all of these types, their semantics, and their processing rules, by reference, for backwards compatibility.

These types are useful, but limited. This specification expands the list to cover more specific data types, and introduces attributes that are designed to constrain data entry or other aspects of the UA's behaviour.

### 2.3. Changes to existing controls

In addition to the attributes described below, some changes are made to the content model of HTML form elements to take into account scripting needs. Specifically, the `form`, `legend`, `select`, and `optgroup` elements may now be empty. However, with the exception of the `form` element, authors should avoid allowing any of these elements to be both empty and visible for any noticeable period, as it is likely to confuse users. ***In HTML4, those elements always required at least one element child, or, in the case of `legend`, at least one character of text.***

Also, as [controls no longer need to be contained within their `form` element](#) to be associated with it, authors may prefer to declare their forms in advance, at the top of their documents. The `form` element is therefore allowed in the `head` element of XHTML documents, although only when the `form` element is empty. (This does not apply to HTML, where a `<form>` tag has always implied the end of any unclosed `head` element and the beginning of the `body`.)

Similarly, `form` elements in XHTML may now be nested (this does not apply to HTML, where a `<form>` tag is interpreted by UAs as implying the end of any unclosed `form` elements). Form controls by default associate with their nearest form ancestor. Forms are not semantically related to ancestor forms in any way, and do not share attributes or form controls or events (except insofar as events bubble up the DOM).

The children of a `form` element must be block-level elements, unless one of the ancestors of the `form` element is an element other than `div` whose content model includes both block- and inline-level content, in which case either block-level or inline-level content is allowed (but not both). `input` elements of type `hidden` may be placed anywhere (both in inline contexts and block contexts).

The `form` and `select` elements are extended with [data attributes](#) for fetching values and options from external resources.

Radio buttons in sets where none of the buttons are marked as checked must all be initially left unchecked by the UA (which differs from the behavior described in [RFC1866](#), but more accurately represents common implementation and author needs). Authors are recommended to always have one radio button selected. Having no radio buttons selected is considered very poor UI.

Radio buttons in sets where more than one button is marked as checked must all be initially left unchecked by the UA except for the last radio button marked as checked. Each time a checked radio button is inserted into the document, the UA must uncheck all the other radio buttons in that set in the document. Authors must not mark more than one radio button per set as being checked.

Previous versions of Web Forms were inconsistent about whether the first `option` element of a single-select `select` element with no otherwise-selected items should be automatically selected. According to [RFC1866](#), it should be, and according to [HTML4](#) it was undefined. User agents implementing this specification must select the first (non-disabled) `option` element of a single-select `select` element with no otherwise-selected items. (If all the items are disabled or there are no items, then no item will be selected.)

The `optgroup` element may now be nested inside other `optgroup` elements.

The `label` element's exact default presentation and behaviour should match the platform's label behaviour. For example, on platforms where clicking a checkbox label checks the checkbox, clicking a `label` element must cause a `click` event to be synthesised and fired at the checkbox. In any case, events targeted at form controls (or other interactive elements, e.g. links) within a label must not be handled by the label itself.

User agents may establish a button in each form as being the form's default button. (This should be the first submit button in the form, but UAs may pick another button if another would be more appropriate for the platform.) If the platform supports letting the user submit a form implicitly (for example, on some platforms hitting the "enter" key while a text field is focused implicitly submits the form), then when doing so the form's default submit button must be the one used to initiate form submission (and it will therefore probably be [successful](#)). To initiate for submission in such a case, the user agent must fire a `click` event at the button's element, as if the user had clicked the button himself.

Consequently, if the default button is disabled, the form must not be submitted when such an implicit submission mechanism is used. (The default action of a click on a disabled button is to do nothing.)

If there is no submit button, then the implicit submission mechanism must submit the form as if there was an enabled, unnamed, default button. No `click` event is fired in this case.

**Note:** Submit buttons can be associated with multiple forms, but only ever submit to the first form they are associated with. A default button for one form, therefore, could submit a different form when implicitly invoked than the form for which it is a default button. (This, however, is an edge case.)

For `checkbox` and `radio` form controls, the `value` attribute defaults to the literal string `on`, so that if the `value` content attribute is not specified then the `value` DOM attribute (and the value used for submission when the controls are checked) is `"on"`. For other controls the default is the empty string.

The attributes defined in this specification as accepting a fixed set of values (e.g. `type`) must be compared to those values using a case-insensitive literal comparison. Whitespace must not be trimmed from attribute values to make that comparison.

Whitespace must also not be trimmed from any other attributes (e.g. the `value` attribute).

**Note:** Whitespace can get trimmed by the parser for other reasons; e.g. if an XML DTD is used, the XML specification can require certain attributes to have whitespace trimmed.

## 2.4. Extensions to the `input` element

Several new values are introduced for the `type` attribute. As with the older types, UAs are recommended to show specialized widgets for these types, instead of requiring that the user enter the data into a text field.

*The formats described below are those that UAs must use in the DOM and when submitting the data. They do not necessarily represent what the user is expected to type. User agents are expected to show suitable user interfaces for each of these types (e.g. using the user's locale settings). It is the UA's responsibility to convert the user's input into the specified format.*

For most of these types, `min`, `max` and `step` attributes can be applied to restrict the range of numbers that apply.

**`datetime`**

A date and time (year, month, day, hour, minute, second, fractions of a

second) encoded according to ISO 8601 [\[ISO8601\]](#) with the time zone set to UTC: four or more digits (0 to 9) representing the year, a hyphen (U+002D), two digits for the month, a hyphen, two digits for the day, a literal "T", two digits for the hour, a colon (U+003A), two digits for the minute, optionally a colon and two digits for the second, optionally (if the seconds are present) a period (U+002E) and one or more digits for the fraction of a second, and finally a mandatory literal "Z". All the numbers must be in base ten and zero-padded if necessary. If the seconds are omitted, they must be assumed to be zero. If the fraction is omitted, it must be assumed to be zero as well. For instance:

1995-12-31T23:59:59.99Z or, representing the time 10 milliseconds later, 1996-01-01T00:00Z. The step attribute specifies the precision in seconds, defaulting to 60 (one minute).

User agents are expected to show an appropriate widget. UAs may display the time in whatever time zone is appropriate for the user, but should be clear to the user that the time is globally defined, not time-zone dependent. The submitted date and time must be in the UTC time zone.

#### **datetime-local**

A date and time (year, month, day, hour, minute, second, fractions of a second) encoded according to ISO 8601 [\[ISO8601\]](#), with no time zone information: four or more digits (0 to 9) representing the year, a hyphen (U+002D), two digits for the month, a hyphen, two digits for the day, a literal "T", two digits for the hour, a colon (U+003A), two digits for the minute, optionally a colon and two digits for the second, and optionally (if the seconds are present) a period (U+002E) and one or more digits for the fraction of a second. All the numbers must be in base ten and zero-padded if necessary. If the seconds are omitted, they must be assumed to be zero. If the fraction is omitted, it must be assumed to be zero as well. For instance: 1995-12-31T23:59:59.99 or, representing the time 10 milliseconds later, 1996-01-01T00:00. The step attribute specifies the precision in seconds, defaulting to 60 (one minute).

#### **date**

A date (year, month, day) encoded according to ISO 8601 [\[ISO8601\]](#): four or more digits (0 to 9) representing the year, a hyphen (U+002D), two digits for the month, a hyphen, and two digits for the day. All the numbers must be in base ten and zero padded if necessary. For instance: 1995-12-31. The step attribute specifies the precision in days, defaulting to 1. User agents are expected to show an appropriate widget, such as a datepicker.

#### **month**

A date consisting of a year and a month encoded according to ISO 8601

[\[ISO8601\]](#): four or more digits (0 to 9) representing the year, a hyphen (U+002D), and two digits for the month, zero-padded if necessary. All the numbers must be in base ten. For instance: 1995-12. The `step` attribute specifies the precision in months, defaulting to 1. This type is used most frequently for credit card expiry dates.

#### **week**

A date consisting of a year and a week number encoded according to ISO 8601 [\[ISO8601\]](#): four or more digits (0 to 9) representing the year, a hyphen (U+002D), a literal "W", and two digits for the week, zero-padded if necessary. All the numbers must be in base ten. The week number must be a number greater than or equal to 01. Week 01 of a given year is the week containing the 4th of January; weeks start on Monday. For instance: 2005-W52 is the week that ends on Sunday the first of January, 2006. The `step` attribute specifies the precision in weeks, defaulting to 1. This type is used most frequently for dates in European industry.

#### **time**

A time (hour, minute, seconds, fractional seconds) encoded according to ISO 8601 [\[ISO8601\]](#) with no time zone: two digits (0-9) for the hour, a colon (U+003A), two digits for the minute, optionally a colon and two digits for the second, and optionally (if the seconds are present) a period (U+002E) and one or more digits for the fraction of a second. All the numbers must be in base ten and zero-padded if necessary. If the seconds are omitted, they must be assumed to be zero. If the fraction is omitted, it must be assumed to be zero as well. For instance:

23:59:00.00000 or 00:00:05. The `step` attribute specifies the precision in seconds, defaulting to 60. Times must be greater than or equal to 0 and must be less than 24 hours, in addition to any tighter restrictions placed on the field by the `min` and `max` attributes. Note that this type is not an elapsed time data type.

User agents are expected to show an appropriate widget, such as a clock. UAs should make it clear to the user that the time does not carry any time zone information.

#### **number**

A numerical value. The `step` attribute specifies the precision, defaulting to 1.

Numbers must be submitted as a significand followed by an optional exponent. The significand is an optional minus sign (U+002D, "-"), an integer, and optionally a decimal point (U+002E, ".") and an integer representing the fractional part. The exponent is a lowercase literal letter "e", an optional minus sign, and an integer representing the index of a power of ten with which to multiply the significand to get the actual

number. Integers are one or more decimal digits. If the exponent part is omitted its index of a power of ten must be assumed to be zero.

For example, negative-root-two, to 32 significant figures, would be `-1.4142135623730950488016887242097e0`, the radius of the earth given in furlongs, to an arbitrary precision, would be `3.17e4`, and the answer to the life, the universe and everything could be any of (amongst others) `42`, `0042.000`, `42e0`, `4.2e1`, or `420e-1`.

This format is designed to be compatible with `scanf(3)`'s `%f` format, ECMAScript's `parseFloat`, and similar parsers while being easier to parse than some other floating point syntaxes that are also compatible with those parsers.

The strings `+0`, `0e+0`, and `+1e+3` are all invalid numbers (the minus sign cannot be replaced by a plus sign for positive numbers, it must simply be omitted). Similarly, `.42e2` is invalid (there must be at least one digit before the decimal point). UAs must not submit numbers in invalid formats (whatever the user might enter).

The submission format is not intended to be the format seen and used by users. UAs may use whatever format and UI is appropriate for user interaction; the description above is simply the submission format.

#### range

Same as number, but indicates that the exact value is not important, letting UAs provide a simpler interface than they do for number. For instance, visual UAs may use a slider control. The step, min, and max attributes still apply. For this type, step defaults to 1, min defaults to 0, max defaults to 100, and value defaults to the min value.

Volume controls and brightness controls would be good examples of "range" data controls.

#### email

An e-mail address, following the format of the `addr-spec` token defined in RFC 2822 section 3.4.1 [\[RFC2822\]](#), but excluding the `CFWS` subtoken everywhere, and excluding the `FWS` subtoken everywhere except in the `quoted-string` subtoken. UAs could, for example, offer e-mail addresses from the user's address book. (See below [for notes on IDN](#).)

#### url

An IRI, as defined by [\[RFC3987\]](#) (the `IRI` token, defined in RFC 3987 section 2.2). UAs could, for example, offer the user URIs from his bookmarks. (See [below](#) for notes on IDN.) The value is called url (as



opposed to `iri` or `uri`) for consistency with CSS syntax and because it is generally felt authors are more familiar with the term "URL" than the other, more technically correct terms.

***Note: Relative URIs and IRIs do not match the `IRI` token mentioned above. Only absolute addresses (potentially with fragment identifiers) are valid values for this input type. Of course, this does not prevent a user agent from allowing users to enter relative or incomplete values, but such values would have to be expanded to complete addresses before the control's `isTypeMismatch` flag is cleared.***

Any string that matches the IRI token must be accepted as a valid value by user agents. For example, user agents are not required to check that given values are in full logical order.

The `email` and `url` fields may contain IDN domains. [\[RFC3490\]](#) These should be sent in their original (full-Unicode) characters, not IDNA-encoded. (Authors can use the pattern `pattern="[\x00-\x7F]+"` to indicate that only ASCII-based domain names are to be allowed.)

A control is said to have **no value selected** if its value is the empty string. File controls are said to have no value selected if no files have been selected.

By default, all of these new types (except `range`), just like the types from HTML4, must have [no value selected](#) unless a default value in a valid format is provided using the `value` attribute. For all controls, if a value is specified but it is not in a format that is valid for the type (where the valid types are the same as the valid submission types described above) then the `defaultValue` DOM attribute has the specified value, and the control is left with the value it would have had if the `value` attribute had not been specified (namely, [no value selected](#), except for `range` controls, which have the `min` value selected).

User agents may allow users to set a control to its "[no value selected](#)" state, but are not required to do so. For example, radio buttons often cannot be returned to their "[no value selected](#)" state.

Fields with [no value selected](#) do not need to match the format appropriate for their type. (Although if they are [required fields](#), they will stop submission for that reason instead.)

If a control has its `type` attribute changed to another type, then the user agent must reinterpret the current value (given by the `value` DOM attribute) and the default value (given by the `value` content attribute and the `defaultValue` DOM attribute) in light of the new type. Values that no longer match the format allowed for the control must be handled as described [in the error handling](#)

[section.](#)

The following form uses some of the types described above:

```
<form action="..." method="post" onsubmit="verify(event)">
  <p>
    <label>
      Quantity:
      <input name="count" type="number" min="0" max="99" value="1" />
    </label>
  </p>
  <p>
    <label for="time1"> Preferred delivery time: </label>
    <input id="time1" name="time1" type="time" min="08:00" max="17:00" />
    <input id="time2" name="time2" type="time" min="08:00" max="17:00" />
  </p>
  <script type="text/javascript">
    function verify(event) {
      // check that time1 is smaller than time2, otherwise, swap them
      if (event.target.time1.value >= event.target.time2.value) { //
        var time2Value = event.target.time2.value;
        event.target.time2.value = event.target.time1.value;
        event.target.time1.value = time2Value;
      }
    }
  </script>
</form>
```

If in this example the "time1" field was changed to be of type `date`, the current value (as picked by the user or as initialised by the `value` attribute), the default value (given by the `value` attribute in the markup and the `defaultValue` attribute in the DOM) and the various constraints (`min` and `max` here) would all be found to be invalid and the control would therefore become a date control with no minimum or maximum, and with no value selected.

Servers should still perform type-checking on submitted data, as malicious users or rogue user agents might submit data intended to bypass this client-side type-checking. Validation done via script may also be easily bypassed if the user has disabled scripting. Additionally, legacy user agents do not support the validation features described in this specification and will therefore submit data that has not been checked.

The `size` attribute of the `input` element is deprecated in favor of using CSS to specify the layout of the form.

### 2.4.1. Ranges

To limit the range of values allowed by some of the above types, two new attributes are introduced, which apply to the date-related, time-related, numeric,



and file upload types:

**min**

Gives the minimum value (inclusive) of the field, in the format specified for the relevant type. Values for the field less than the minimum value are out of range ([rangeUnderflow](#)). If absent, or if the minimum value is not in exactly the expected format, there is no minimum restriction, except for the [range](#) and [file](#) types, where the default is zero.

**max**

Gives the maximum value (inclusive) of the field, in the format specified for the relevant type. Values for the field greater than the maximum value are out of range ([rangeOverflow](#)). If absent, or if the maximum value is not in exactly the expected format, there is no maximum restriction (beyond those intrinsic to the type), except for the [range](#) type, where the default is 100, and the [file](#) type, where the default is 1.

For date, time and numeric fields, the values indicate the allowed range. For file upload fields, the values indicate the allowed number of files.

The [typeMismatch](#) flag is used for fields whose values do not match their types, and the [rangeUnderflow](#) and [rangeOverflow](#) flags are used for fields whose values are outside the allowed range.

A field with a [max](#) less than its [min](#) can never fulfill both conditions when it has a value (since that value will always either underflow or overflow the allowed range) and thus must block its forms from being submitted. This does not make the document non-conformant.

The exact values allowed by [min](#) and [max](#) depend on the [type](#) attribute. For numeric types ([number](#) and [range](#)) the value must exactly match the [number](#) type described above. For [file](#) types it must be a sequence of digits 0-9, treated as a base ten integer. For date and time types it must match the relevant format mentioned for that type, all fields having the right number of digits, with the right separating punctuation.

If a control has its [type](#) attribute changed to another type, then the user agent must reinterpret the [min](#) and [max](#) attributes. If an attribute has an invalid value according to the new type, then the appropriate default must be used (and not, e.g., the default appropriate for the previous type). Control values that no longer match the range allowed for the control must be handled as described [in the error handling section](#).

For example, 50.00 does not match the allowed value of [time](#), so the following control has no artificial restrictions on its values:

```
<input type="time" min="50.00">
```

Similarly, the value 2000 is not a valid value for [datetime](#), [date](#), or any of the other date or time types.

The following cases would work, though;

```
<input type="time" min="22:00">
<input type="time" min="22:00:50.0001">
```

## 2.4.2. Precision

The [step](#) attribute controls the precision allowed for the date-related, time-related, and numeric types.

For the control to be valid, the control's value must be an integral number of steps from the [min](#) value, or, if there is no [min](#) attribute, the [max](#) value, or if there is neither attribute, from the zero point.

The zero point for [datetime](#) controls is 1970-01-01T00:00:00.0Z, for [datetime-local](#) is 1970-01-01T00:00:00.0, for [date](#) controls is 1970-01-01, for [month](#) controls is 1970-01, for [week](#) controls is 1970-W01 (the week starting 1969-12-29 and containing 1970-01-01), and for [time](#) controls is 00:00.

For time controls, the value of the [step](#) attribute is in seconds, although it may be a fractional number as well to allow fractional times. The format of the [step](#) attribute is the [number format](#) described above, except that the value must be greater than zero. The default value of the [step](#) attribute for [datetime](#), [datetime-local](#) and [time](#) controls is 60 (one minute).

For the following control, the allowed values are any day of any year, with the times restricted to even minutes:

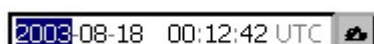
```
<input type="datetime" step="120" name="start">
```

For the following control, the allowed values are fifteen seconds and two tenths of a second past the minute, any minute of the day, i.e. 00:00:15.2, 00:01:15.2, 00:02:15.2 ... 23:59:15.2:

```
<input type="time" min="00:00:15.20" name="t">
```

This is because the default [step](#) for [time](#) controls is 60 (one minute).

How the [step](#) attribute affects the UI is not defined by this specification. For example, for a [datetime](#) control with [step](#)="1", the UI could look like this:



For date controls, the value of the `step` attribute is in days, weeks, or months, for the `date`, `week`, and `month` types respectively. The format is a non-negative integer; one or more digits 0-9 interpreted as base ten. If the step is zero, it is interpreted as the default. The default for the `step` attribute for these control types is 1.

The following control would only allow selection of any Sunday in any year from 1900 onward:

```
<input type="date" min="1900-01-07" step="7" name="sunday">
```

For numeric controls (`number` and `range`), the format of the `step` attribute is the [number format](#) described above, except that the value must be greater than zero. For numeric controls, the default value of the `step` attribute is 1.

If the step is  $25e-2$  (or 0.25, which is equivalent), and if `max` is -1.1, then the allowed values would be -1.1, -1.35, -1.60, -1.85, -2.1, ...

If you wanted a range control that allowed only even numbers, you could set:

```
<input type="range" step="2" name="evenN">
```

The following control would have a step of 1, the default, because the given value for the `step` attribute does not match the allowed values for numbers as defined above (it would need a leading zero before the decimal point):

```
<input type="range" step=".1" name="n">
```

In addition, for any of the types, the literal value `any` may be used as the value of the `step` attribute. This keyword indicates that any value may be used (within the bounds of other restrictions placed on the field).

The following control would allow any floating point number:

```
<input type="number" step="any" name="n">
```

The `stepMismatch` flag is used for fields whose values are not one of the values allowed by the `step` attribute. However, UAs may silently round the number to the nearest value allowed by the `step` attribute instead of reporting a `stepMismatch` validation error. (Such rounding may make the value out of range, causing, for instance, a `rangeOverflow` validation error.)

If the author specified step is too small for the UA to handle (for example, `1e-99999999` would probably underflow most implementations) then the UA should treat the value as `any`. If the given `step` value is not one of the allowed values, then the default is used.

User agents are recommended to never convert user- and author-supplied values to their binary numeric representation, keeping the values in string form at all times and performing comparisons in that form. This ensures that UAs are able to handle arbitrarily large numbers without risking data loss due to rounding in the decimal-to-binary conversion.

If a UA needs to round a number to its nearest binary equivalent, as, for example, when converting a user-supplied decimal number and an author-supplied minimum in order to compare them to establish validity (ignoring the suggestion above to do these comparisons in string form), algorithms equivalent to those specified in ECMA262 sections 9.3.1 ("ToNumber Applied to the String Type") and 8.5 ("The Number type") should be used (possibly after suitably altering the algorithms to handle numbers of the range that the UA can support). [\[ECMA262\]](#)

If a control has its `type` attribute changed to another type, then the user agent must reinterpret the `step` attribute. If it has an invalid value according to the new type, then the new appropriate default must be used. Control values that no longer match the precision allowed for the control must be handled as described [in the error handling section](#).

## 2.5. Extensions to existing attributes

In addition to the new attributes given in this section, some existing attributes from [\[HTML4\]](#) are clarified and extended below. These, and other attributes from HTML4, continue having the same semantics as described in HTML4 unless specified otherwise.

### `accesskey`

UAs may now support the `accesskey` attribute on `select` elements (and must at a minimum support the relevant DOM attribute).

The `accesskey` attribute on `label` elements must act the same way as it would if specified on the associated element directly.

### `disabled`

The `disabled` attribute applies to all form controls except the `output` element, and also to the `fieldset`, `option`, and `optgroup` elements.

***In HTML4 the `disabled` attribute did not apply to the `fieldset` element.***

When applied to a `fieldset` element it overrides the `disabled` attributes of any descendent form controls (regardless of whether they are associated with the same form). In other words, a form control shall be disabled if it

has its disabled attribute set, or if any of its ancestor `fieldset` elements have *their* disabled attribute set.

#### **maxlength**

This attribute applies to text, password, url, email and file input types, and `textarea` elements. In particular, it does not apply to the date-related, time-related, and numeric field types. ***In HTML4, this attribute only applied to the text and password types.***

For text input controls it specifies the maximum length of the input, in terms of numbers of code points. [\[CHARMOD\]](#).

A newline in a `textarea`'s value must count as two code points for maxlength processing (because newlines in `textareas` [are submitted as U+000D U+000A](#)). This includes the implied newlines that are added for submission when the wrap attribute has the value hard.

Authors are discouraged from using maxlength on url and email fields unless the server side processor actually has a limit on the size of data fields it can usefully process. Valid URIs and e-mail addresses in particular can often be surprisingly long.

When specified on a file upload control, it specifies the maximum size in bytes of each file (not the maximum size of the sum of all the files).

The tooLong flag is used when this attribute is specified on a text, password, url, email or `textarea` control and the control has more than the specified number of code points, or when it is specified on a file control and at least one of the selected files is longer than the specified number of bytes.

Servers should still expect to receive, and must be able to cope with, content larger than allowed by the maxlength attribute, in order to deal with malicious or non-conforming clients.

This attribute must not affect the initial value (the DOM defaultValue attribute). It must only affect what the user may enter and whether a validity error is flagged during validation.

***If the maxlength attribute has a value that is less than the length required for a valid value of the given type, for example:***

```
<input type="email" maxlength="1" name="test"/>
```

***...then the control can only be valid if it is has no value***

**selected (unless, of course, it is a required field, in which case it can never be valid).**

#### name

Some names (all starting with the string "Ecom\_") are reserved by [\[RFC3106\]](#). Authors must not use names starting with the string "Ecom\_" in ways that conflict with RFC3106.

#### readonly

This attribute applies only to `text`, `password`, `email`, `url`, date-related, time-related, and `number` input types, as well as the `textarea` element. Specifically, it does not apply to radio buttons, checkboxes, file upload fields, `range` controls, `select` elements, or any of the button types; the interface concept of "readonly" values does not apply to button-like user interfaces.

Other attributes not listed in this specification retain the same semantics as in [\[HTML4\]](#).

## 2.6. The `pattern` attribute

For the `text`, `password`, `email`, and `url` types of the `input` element, and for the `textarea` element, the `pattern` attribute specifies a pattern that the control value must match.

When specified, the `pattern` attribute contains a regular expression that the field's value must match before the form may be submitted (`patternMismatch`).

```
<label> Credit Card Number:
  <input type="text" pattern="[0-9]{13-16}" name="cc" />
</label>
```

The regular expression language used for this attribute is the same as that defined in [\[ECMA262\]](#), except that the `pattern` attribute implies a `^` at the start of the pattern and a `$` at the end (so the pattern must match the entire value, not just any subset). The pattern must be compiled with the `global`, `ignoreCase`, and `multiline` flags disabled (see ECMA262, sections 15.10.7.2 through 15.10.7.4). If the attribute is omitted then the control has no pattern restriction.

***The implicit `^` and `$` characters are inserted because it is expected that the overwhelming majority of use cases will be to require that user input exactly match the given pattern. Authors who forget that these characters are implied will immediately realise their***

***mistake during testing. Had the characters not been implied, requiring most authors to insert them themselves, it is likely that authors who forgot them would not catch their mistake as easily.***

***Authors who wish to allow for any input so long as a particular string occurs somewhere in the input should put `.*` at the start and end of their pattern. If the input is expected to allow newlines, then `[\s\S]*` or some equivalent should be used instead, since the dot character in JavaScript regular expressions does not include newlines.***

The `/` character is not special in the `pattern` attribute. The two attributes `pattern="/etc/.*"` and `pattern="\/etc\/.*"` are therefore equivalent.

The `^` and `$` characters have their usual meaning. Thus, using the `^` character anywhere other than at the start of the pattern, or the `$` character anywhere other than at the end of the pattern, prevents the pattern from matching anything (unless the characters are escaped or part of a range).

In the case of the `email` and `url` types, the `pattern` attribute specifies a pattern that must be matched *in addition* to the value matching the generic pattern relevant for the field. If the pattern given by the attribute specifies a pattern that is incompatible with the grammar of the field type, as in the example below, then the field could never be satisfied. (A document containing such a situation is not technically non-conformant, but it is of dubious semantic use.)

```
<form>
  <p>
    This form could never be submitted, as the following required field
    can never be satisfied:
    <input type="url" pattern="[^:]+" required="required" name="test"/>
  </p>
</form>
```

When the value doesn't match the field's type, the `typeMismatch` flag must be set; when the value doesn't match the pattern, the `patternMismatch` flag must be set. If the value matches neither the field's type nor the field's pattern, both flags must be set.

When the pattern is not a valid regular expression, it is ignored for the purposes of validation, as if it wasn't specified.

Fields with [no value selected](#) do not need to match their pattern. (Although if they are [required fields](#), they will stop submission for that reason anyway.)

If the `pattern` attribute is present but empty, it doesn't match any value, and

thus the `patternMismatch` flag shall be set whenever the field's value isn't [empty](#).

Authors should include a description of the pattern in the `title` attribute. User agents may use the contents of this attribute when informing the user that the pattern is not matched, or at any other suitable time, such as in a tooltip or read out by assistive technology when the control gains focus.

For example, the following snippet:

```
<label> Part number:
  <input pattern="[0-9][A-Z]{3}" name="part"
        title="A part number is a digit followed by three uppercase
  </label>
```

...could cause the UA to display an alert such as:

*A part number is a digit followed by three uppercase letters.*

*You cannot complete this form until the field is correct.*

When a control has a `pattern` attribute, the `title` attribute, if used, must describe the pattern. Additional information could also be included, so long as it assists the user in entering the field. Otherwise, assistive technology would be impaired.

For instance, if the title attribute contained the caption of the control, assistive technology could end up saying something like `The text you have entered does not match the required pattern. Birthday, which is not useful.`

UAs may still show the `title` in non-error situations (for example, as a tooltip when hovering over the control), so authors should be careful not to word titles as if an error has necessarily occurred.

## 2.7. The `required` attribute

Form controls can have the `required` attribute specified, to indicate that the user must enter a value into the form control before submitting the form.

The `required` attribute applies to all form controls except controls with the type `hidden`, image inputs, buttons (`submit`, `move-up`, etc), and `select` and `output` elements. It *can* be used on controls with the `readonly` attribute set; this may be useful in scripted environments. For disabled controls, the attribute has no effect.

The `missingValue` flag is used for form controls marked as required that do not have values.



For checkboxes, the required attribute shall only be satisfied when one or more of the checkboxes with that name in that form are checked.

For radio buttons, the required attribute shall only be satisfied when exactly one of the radio buttons in that radio group is checked.

For [file upload controls](#), the required attribute shall only be satisfied if at least one valid file is selected, regardless of the min and max attributes.

Here is a form fragment showing one required field and one optional field. A user agent would not allow the user to submit the form until the "name" field was filled in.

```
<ul>
  <li>Name: <input type="text" name="name" required="required" /></li>
  <li>Comment: <input type="text" name="comment" /></li>
</ul>
```

Any non-[empty](#) value satisfies the required condition, including a simple whitespace character.

## 2.8. The `form` attribute

All form controls as well as the `fieldset` element may have the `form` attribute specified. The `form` attribute gives a space-separated list of IDs of `form` elements that the form control is to be associated with, and overrides the relationship between the form control and any ancestor `form` element.

Any of the IDs in the space-separated list that do not identify an element in the document, or that identify an element that is not an HTML `form` element, must be ignored. Setting an element's `form` attribute to the empty string (or to a string consisting only of IDs that do not correctly identify `form` elements) just disassociates the form control from its form, leaving it unassociated with any form.

When set on a `fieldset` element, this attribute also changes the association of any descendent form controls, unless they have `form` attributes of their own or are contained inside forms that are themselves descendants of the `fieldset` element.

In other words, form controls and `fieldset`s must be associated with the forms given in their `form` attribute, or, if they don't have one, must be associated with the nearest ancestor `form` element or the forms given in the `form` attribute of the nearest `fieldset` element with a `form` attribute, whichever is the nearest. If none of those apply, the element is not associated with any form.

When forms are submitted, are reset, or have their form controls enumerated through the DOM, only those controls associated with the form are taken into account. A control can be associated with more than one form at a time. Submit buttons and image controls must only submit the first form they are associated with. Reset buttons must reset all the forms they are associated with.

A `form` attribute that specifies an ID that occurs multiple times in a document must select the same form as would be selected by the `getElementById()` method for that ID ([\[DOM3CORE\]](#)). (That is, the exact behaviour is undefined, but must be the same as if the `getElementById()` method was used.)

If a form is specified multiple times in the `form` attribute, all occurrences but the first must be ignored. (An element can only be associated with a form once.)

In this example, each row contains one form. Without the "form" attribute, it would not be possible to have more than one form per table if any of them spanned cells.

```
<table>
  <thead>
    <tr>
      <th>Name</th>
      <th>Value</th>
      <th>Action</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>
        <form id="edit1" action="/edit" method="post">
          <input type="hidden" name="id" value="1"/>
          <input type="text" name="name" value="First Row"/>
        </form>
      </td>
      <td>
        <input form="edit1" type="text" name="value"/>
      </td>
      <td>
        <input form="edit1" type="submit" name="Edit"/>
      </td>
    </tr>
    <tr>
      <td>
        <form id="edit2" action="/edit" method="post">
          <input type="hidden" name="id" value="2"/>
          <input type="text" name="name" value="Second Row"/>
        </form>
      </td>
      <td>
        <input form="edit2" type="text" name="value"/>
      </td>
```

```

        <td>
          <input form="edit2" type="submit" name="Edit"/>
        </td>
      </tr>
    </tbody>
  </table>

```

In the following example, the text control is associated with two forms.

```

<form action="test.cgi">
  <input type="text" name="q" form="fg fy">
  <input type="submit" name="t" value="Test">
</form>
<form id="fg" action="google.cgi"><input type="submit" value="Google">
<form id="fy" action="yahoo.cgi"><input type="submit" value="Yahoo">

```

There are three submit buttons. The first, "Test", submits just the "Test" button, and submits it to `test.cgi`. The text field is not submitted with that form in Web Forms 2 compliant UAs. (It *is* submitted in legacy UAs, which can be used for implementing fallback behaviour.)

The second button submits the text field to `google.cgi`, the third button submits the same text field to `yahoo.cgi`.

## 2.9. The autocomplete attribute

The autocomplete attribute applies to the text, password date-related, time-related, numeric, email, and url controls. The attribute takes two values, `on` and `off`. The default, when the attribute is not specified, is `on`.

The `off` value means that the control's input data is either particularly sensitive (for example the activation code for a nuclear weapon) or that it is a value that will never be reused (for example a one-time-key for a bank login) and indicates that the user should therefore explicitly enter the data each time, instead of being able to rely on the UA to prefill the value for him. The `on` value indicates that the value is not particularly sensitive and the user should expect to be able to rely on his UA to remember values he has entered for that field.

When a control has its autocomplete attribute set to a value other than `off`, or when the attribute is omitted, the UA may store the value entered by the user so that if the user returns to the page, the UA can prefill the form. When a control has its autocomplete attribute set to `off`, the UA should not remember that field's value.

This specification does not define the autocompletion mechanism. UAs may implement any system within the conformance criteria of this specification, taking into account security and privacy concerns.

Banks frequently do not want UAs to prefill login information:

```
<p>Account: <input type="text" name="ac" autocomplete="off" /></p>
<p>PIN: <input type="text" name="pin" autocomplete="off" /></p>
```

A UA may allow the user to disable support for this attribute. Support for the attribute should be enabled by default, and the ability to disable support should not be trivially accessible, as there are significant security implications for the user if support for this attribute is disabled.

***Note: In practice, this attribute is required by many banking institutions, who insist that UAs with auto-complete features implement it before supporting them on their Web sites. For this reason, it is implemented by most major Web browsers already, and has been for many years.***

## 2.10. The `autofocus` attribute

Any form control (except `hidden` and `output` controls) can have an `autofocus` attribute specified.

When a form control is inserted into a document, the UA must check to see if it has this attribute set. If it does, and the control is not `disabled`, and it is of a type normally focusable in the user's operating environment, then the UA should focus the control, as if the control's `focus()` method was invoked. UAs with a viewport should also scroll the document enough to make the control visible, even if it is not of a type normally focusable.

Authors should avoid setting the `autofocus` attribute on multiple enabled elements in a document. If multiple elements with the `autofocus` attribute set are inserted into a document, each one will be processed as described above, as they are inserted. This means that during document load, for example, the last focusable form control in document order with the attribute set will end up with the focus.

UAs may ignore this attribute if the user has indicated (for example, by starting to type in a form control) that he does not wish focus to be changed.

The value of the attribute, if set, should be `autofocus`. The `autofocus` DOM attribute is true when the attribute is present (regardless of its value, even if it is the empty string), and false when it is absent. Setting the DOM attribute to true sets the content attribute to the value `autofocus`. Setting the DOM attribute to false removes the content attribute.

In the following snippet, the text field would be focused when the document

was loaded.

```
<input maxlength="256" name="q" value="" autofocus="autofocus">
<input type="submit" value="Search">
```

In HTML, the minimised form may be used (just [autofocus](#) instead of `autofocus="autofocus"`).

The following would cause the [autofocus](#) attribute to be set to [autofocus](#):

```
<input autofocus>
```

***Note: Focusing the control does not imply that the UA must focus the browser window if it has lost focus.***

## 2.11. The [inputmode](#) attribute

The `inputmode` attribute applies to the `input` element when it has a `type` attribute of [text](#), [password](#), [email](#), or [url](#), and to the `textarea` element.

This attribute is defined to be exactly equivalent to the [inputmode](#) attribute [defined in the XForms 1.0 specification](#) (sections E1 through E3.2) [\[XForms\]](#).

## 2.12. The [datalist](#) element and the [list](#) attribute

For the [text](#), [email](#), [url](#), date-related, time-related, and numeric types of the `input` element, a new attribute `list` is introduced to point to a list of values that the UA should offer to the user in addition to allowing the user to pick an arbitrary value.

To complement the new [list](#) attribute, a `datalist` element is introduced. This element has two roles: it provides a list of data values, in the form of a list of `option` elements, and it may be used to provide fallback content for user agents that do not support this specification.

If the UA supports this element, it should not be displayed. In CSS-aware user agents, this should be achieved by including the following rules, or their equivalent, in the UA's user agent stylesheet:

```
@namespace xh url(http://www.w3.org/1999/xhtml);
xh|datalist { display: none; }
```

When a control has a [list](#) attribute, it specifies an element from which to derive the list of author-specified autocompletion values for the control.

The element specified is the one that would be returned when calling `getElementById()` with the value of the `list` attribute as the argument, if the returned value is an element node with either the tag name `datalist` or the tag name `select`, and (for XHTML) with the XHTML namespace. If the attribute is present but either specifies an ID that is not in the document, or specifies an element that is not an (X)HTML `datalist` or `select`, then it must be ignored.

The list of autocompletion values shall be given by the list of elements that would be found by calling `getElementsByTagName()` with the tag name `option` on the element specified, if any (or, in XHTML documents, the list of elements that would be found by calling `getElementsByTagNameNS()` with the same tag name and the XHTML namespace).

For each element in this list, if the element is not marked as `disabled`, the autocompletion value is either the value of its `value` content attribute, or, if that attribute is absent, the value of its `text` DOM attribute. The UA may use the `label` attribute to annotate the value in its interface. If the element is marked as `disabled`, if the autocompletion value is the empty string, or if the autocompletion value is not a valid value for the control's type (for example, `20` is not a valid value for a `datetime` control) then it must be ignored.

The author-specified list of predefined values may be augmented by the UA's own autocompletion logic. For example, the UA may remember previous values that the user has entered.

Authors must only use empty `option` elements or elements that would be allowed in the `datalist` element's parent as children of `datalist` elements. `datalist` elements may be used wherever block-level elements are allowed and wherever `select` elements are allowed.

Controls inside `datalist` elements must never be [successful](#). (They must still, however, be [associated](#) with their form.)

The `datalist` element may be prefilled from an external file with the `data` attribute.

The `selected` attribute and the `form`, `selected`, `defaultSelected`, and `index` DOM attributes on `option` elements must have no effect on the `input` and `datalist` elements when `option` elements are used in this context.

If a document contained the following markup:

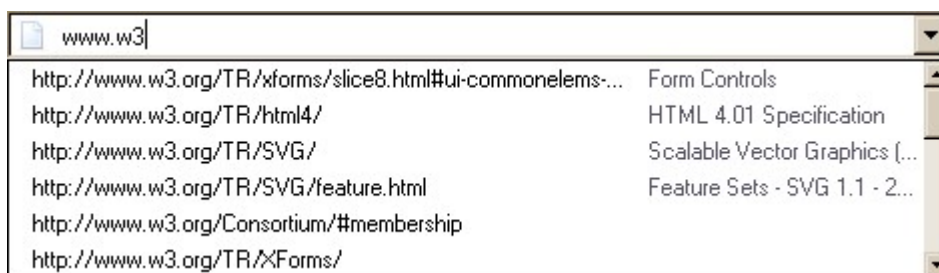
```
<input type="url" name="location" list="urls">
<datalist id="urls">
  <option label="MIME: Format of Internet Message Bodies" value="http:
  <option label="HTML 4.01 Specification" value="http://www.w3.org/T
  <option label="Form Controls" value="http://www.w3.org/TR/xforms/s
```

```

<option label="Scalable Vector Graphics (SVG) 1.1 Specification" v.
<option label="Feature Sets - SVG 1.1 - 20030114" value="http://ww
<option label="The Single UNIX Specification, Version 3" value="ht
</datalist>

```

...and the user had typed "www.w3", and the user agent had also found that the user had visited <http://www.w3.org/Consortium/#membership> and <http://www.w3.org/TR/XForms/> in the recent past, then the rendering might look like this:



The first four URIs in this sample consist of the four URIs in the author-specified list that match the text the user has entered, sorted lexically. Note how the UA is using the knowledge that the values are URIs to allow the user to omit the scheme part and perform intelligent matching on the domain name.

The last two URIs (and probably many more, given the scrollbar's indications of more values being available) are the matches from the user agent's session history data. This data is not made available to the page DOM. In this particular case, the UA has no titles to provide for those values.

This example demonstrates how to design a form that uses the autocomplete list feature while still degrading usefully in legacy user agents.

If the autocomplete list is merely an aid, and is not important to the content, then simply using a `<datalist>` element with children `<option>` elements is enough. To prevent the values from being rendered in legacy user agents, they should be placed inside the `value` attribute instead of inline.

```

<p>
<label>
  Enter a breed:
  <input type="text" name="breed" data="breeds">
  <datalist id="breeds">
    <option value="Abyssinian">
    <option value="Alpaca">
    <!-- ... -->
  </datalist>

```

```

    </label>
  </p>

```

However, if the values need to be shown in legacy UAs, then fallback content can be placed inside the `datalist` element, as follows:

```

<p>
  <label>
    Enter a breed:
    <input type="text" name="breed" data="breeds">
  </label>
  <datalist id="breeds">
    <label>
      or select one from the list:
      <select name="breed">
        <option value=""> (none selected)
        <option>Abyssinian
        <option>Alpaca
        <!-- ... -->
      </select>
    </label>
  </datalist>
</p>

```

The fallback content will only be shown in UAs that don't support `datalist`. The options, on the other hand, will be detected by all UAs, even though they are not direct children of the `datalist` element.

Note that if an `option` element used in a `datalist` is selected, it will be selected by default by legacy UAs (because it affects the `select`), but it will not have any effect on the `input` element in UAs that support `datalist`.

Here is another example, this time with a range control. This could be useful if there are values along the full range of the control that are especially important, such as preconfigured light levels or typical speed limits in a range control used as a speed control. The following markup fragment:

```

<input type="range" min="-100" max="100" value="0" step="10" name="";
<datalist id="powers">
  <option value="0">
  <option value="-30">
  <option value="30">
  <option value="+50">
</datalist>

```

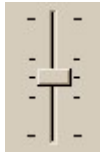
...with the following stylesheet applied:

```
input { height: 75px; width: 49px; background: #D5CCBB; color: black; }

```

...might render as:





Note how the UA determined the orientation of the control from the ratio of the stylesheet-specified height and width properties. The colours were similarly derived from the stylesheet. The tick marks, however, were derived from the markup. In particular, the `step` attribute has not affected the placement of tick marks, the UA deciding to only use the author-specified completion values and then adding longer tick marks at the extremes.

Note also how the invalid value `+50` was completely ignored.

***Note: This specification does not mandate a particular interface. The UA could have used a rotary control, a combo box, a voice-driven text box, or any other widget or interface while still being compliant with this specification.***

## 2.13. The `output` element

The `output` element acts very much like a `span` element, except that it is considered to be a form control for the purposes of the DOM. Its namespace (in XML) is the same as for the other form control elements, `http://www.w3.org/1999/xhtml`.

The `output` element may have any of the [common attributes](#), the `form` and `name` attributes, the `for` attribute (defined below), and the `onchange`, `onforminput` and `onformchange` attributes.

Its current value is given by its contents, which may be any inline content (like the `span` element).

The current value can be set and retrieved dynamically using the mutable `value` DOM attribute of type `DOMString`. This attribute is defined to be identical to the DOM3 Core `textContent` attribute. [\[DOM3CORE\]](#)

The *initial value* of the `output` control is stored in a mutable `defaultValue` DOM attribute of type `DOMString`. See [\[HTML4\]](#) section 17.2 for [the definition of the term "initial value"](#). (In brief, it is the value used when the form is reset.)

The `defaultValue` DOM attribute of an `output` control must initially be set to the empty string. If an `output` element is added to the document at parse time, its `defaultValue` DOM attribute must be set to the value of its `textContent` attribute

after all its children nodes were parsed. (If the value of `defaultValue` is queried before the entire element's contents have been parsed, or if the element was created dynamically (as opposed to being inserted into the DOM by the UA's parser), then `defaultValue` must return the empty string.)

The `output` element is never [successful](#) for form submission. Resetting a form *does* reset its `output` elements (using the `defaultValue` DOM attribute — note that if the element originally contained elements as children, they will be removed when the form is reset).

**Note: Unless the `value` attribute is directly manipulated or the form is reset, elements that are children of the `output` element when the document was parsed are not flattened away.**

The following example shows two input fields. Changing either field updates an `output` element containing the product of both fields.

```
<form>
  <p>
    <input name="a" type="number" step="any" value="0"> *
    <input name="b" type="number" step="any" value="0"> =
    <output name="result" onforminput="value = a.value * b.value">0</o
  </p>
</form>
```

This would work something like the following:

0  \* 0  = 0

**Note: The `forminput` event is defined in the section on new events.**

Authors may provide a list of space-separated IDs in a `for` attribute that represents the list of elements that control the value of the `output` element. User agents may use this list to suggest to users the relevant parts of the document with which the user should interact to change the value.

In the following example, the `output` element is used to indicate the relationship between the given value and the later prose. The number cannot be changed directly by the user, but the specified element describes the process through which the user could change the value.

```
...
<p>Your fax number is <output for="fax"><em>+1</em> 650 555 1234</o
...
<p id="fax">To change your fax number, you must send us a fax
from your new number with a signed request that your fax number
```

```
details be changed. We will then call you to confirm the  
change.</p>
```

Note the `em` element in the markup emphasising a part of the number.  
Markup like this is allowed inside `output` elements.

Whenever the `value` attribute changes (whether directly or because the DOM under the element was mutated), a `change` event is fired on the `output` element. The `onchange` attribute can therefore be used with this element, in the same way as for other form controls.

**Usage:** The `output` element should be used when the user will never directly manipulate the value, and when the value can be derived from other values (e.g. a total), or, when the value is a repetition of a value editable elsewhere (e.g. a fax number that the user can edit on the site's preferences page).

Contrast this with the `readonly` attribute, which should be used on controls that the user should not change, but which need to be submitted to the server (such as an ID number when editing a record), and the `disabled` attribute, which should be used on controls that the user cannot change and that are not to be submitted (controls that could be edited in some cases, for instance if the user had more privileges, but that are irrelevant at the current point in time).

## 2.14. Extensions to the `textarea` element

The `rows` and `cols` attributes of the `textarea` element are no longer required attributes. When unspecified, CSS-compliant browsers should lay the element out as specified by CSS, and non-CSS UAs may use UA-specific defaults, such as, for visual UAs, using the available width on the page and a height suitable for the device.

The `textarea` element may have a `wrap` attribute specified. This attribute controls the wrapping behaviour of submitted text.

### **soft**

This is the default value. The text is submitted without line breaks other than explicitly entered line breaks. (In other words, the submitted text is exactly as found in the DOM.)

### **hard**

The text is submitted with the explicit line breaks, and also with line breaks added to wrap the text at the width given by the `cols` attribute. (These additional line breaks can't be seen in the DOM.)

Authors should always specify a `cols` attribute when the `wrap` attribute is set to

hard. When `wrap="hard"` is specified without a `cols` attribute, user agents should use the display width when wrapping the text for submission. This will typically mean that different users submit text at different wrapping widths, defeating much of the purpose of client-side wrapping.

CSS UAs should *render* `textarea` elements as specified by the `'white-space'` property, although UAs may have rules in their UA stylesheet that set the default `'white-space'` property values based on the `wrap` element for `textarea` elements.

A newline in a `textarea` submission must be delimited by the two code points U+000D U+000A (CRLF). This includes the implied newlines that are added for submission when the `wrap` attribute has the value `hard`. This also affects [how newlines are handled for the `maxlength` attribute](#), which applies to `textarea` controls.

Authors may include an `accept` attribute on `textarea` elements to indicate the type of content expected. User agents may use this attribute to provide more appropriate editors, syntax highlighting, spelling checkers, etc. The value of the attribute must be a single text-based MIME type (for example, `text/plain`, `message/news`, `image/svg+xml`). The default, if the attribute is omitted or if the value is not recognised by the UA, shall be `text/plain`. [\[RFC2046\]](#)

## 2.15. Extensions to file upload controls

The `min` and `max` attributes apply to file upload controls (`input` elements of type `file`) and specify (using non-negative integers) how many files must be attached for the control to be valid. They default to 0 and 1 respectively (and so limit the default number of files to 1 optional file, as per most existing implementations in early 2004). The defaults are used when the attributes are omitted or have non-integer or negative integer values. The `rangeUnderflow` and `rangeOverflow` flags are used to indicate when fields do not have the specified number of files selected.

***Note: There is currently no way to specify that an unlimited number of files may be uploaded. Authors are encouraged to consider what the practical limit actually is for their server-side script. For example, if the server uses an unsigned 16 bit integer to track file uploads, a suitable `max` value would be 65536.***

The `required` attribute applies to file upload controls; it is only satisfied when at least one valid file (that is, one that will actually be uploaded if the form is submitted) is selected, irrespective of the `min` and `max` attributes.

Non-existent files and files the UA won't be able to upload (for whatever

reason), when specified in file upload controls, do not count towards the number of files selected for the purposes of [min](#) and [max](#), and are not submitted. It is recommended that user agents report problems of this nature to the user. There is no way for scripts to detect this situation because that would open the way for some privacy or security leaks.

The [maxlength](#) attribute applies to file upload controls. It specifies the maximum size, in bytes, of each file.

The [accept](#) attribute may be used to specify a comma-separated list of content types that a server processing the form will handle correctly. [\[RFC2046\]](#) This attribute was specified in HTML4. [\[HTML4\]](#) In this specification, this attribute is extended as follows:

- In addition to specific MIME types, MIME ranges may be used, either where the subtype is \*, or where the whole string is \*/\*. For example:

```
<input type="file" name="avatar"
      accept="image/*"/>
```

In this way, the [accept](#) attribute may be used to specify that the server is expecting (say) images or sound clips, without specifying the exact list of types.

- The values in the [accept](#) attribute must not have any MIME type parameters. The syntax of the attribute, therefore, is (in pseudo-BNF):

```
accept  := space* range [ space* "," space* range]* space*
range   := "*/*" | type "/"* | subtype "/" type
space   := U+0020 | U+000A
type    := defined in RFC2045 \[RFC2045\]
subtype := defined in RFC2045 \[RFC2045\]
```

- UAs should use the list of acceptable types in constructing a filter for a file picker, if one is provided to the user.
- If the UA wishes to let the user create the file prior to upload, it should use the [accept](#) attribute's MIME type list to determine which application to use.

One recent use for sound file upload has been the concept of *audio blogging*. This is similar to straight-forward Web logging, or diary writing, but instead of submitting textual entries, one submits sound bites.

The submission interface to such a system could be written as follows:

```
<form action="/weblog/submit" method="post" enctype="multipart
<label>
```

```

    Attach your audio-blog sound file:
    <input type="file" name="blog" accept="audio/*"/>
  </label>
  <input type="submit" value="Blog!"/>
</form>

```

A compliant UA could, upon encountering this form, provide a "Record" button instead of, or in addition to, the more usual "Browse" button. Selecting this button could then bring up a sound recording application.

This is expected to be most useful on small devices that do not have file systems and for which the only way of handling file upload is to generate the content on the fly.

- The `typeMismatch` flag is used to indicate that at least one of the selected files does not have a MIME type conforming to one of the MIME types or MIME ranges listed as acceptable. UAs may allow the user to override the MIME type to be one of the allowable types if the file is originally incorrectly labeled.
- If an `accept` attribute is set on a `form` element, it sets the default for any file upload controls in that form. If a file upload control has a valid `accept` attribute, its value is used and the relevant form element's `accept` attribute is ignored for that control. If a file upload control has an `accept` attribute but it is invalid, then the value `*/*` is assumed instead, and the form's `accept` attribute is ignored.

If the file upload process fails, UAs should report this failure to the user in a useful and accessible manner, as with any failed submission.

If a file selected for upload does not have an explicit file name (for example because the user agent allowed the user to create the submitted file on the fly) then the file name shall be the empty string for the purposes of submission.

## 2.16. Extensions to the `form` element

The `form` element's `action` attribute is no longer a required attribute. Authors may omit it. When the attribute is absent, UAs must act as if the `action` attribute was the empty string, which is a relative URI reference, and would thus point to the current document (or the specified base URI, if any).

In the following XML example, the form would submit to `http://search.example.com/`, whatever the URI of the current page.

```
<form xml:base="http://search.example.com/">
```

```

    <p><input type="submit"></p>
  </form>

```

To support incremental updates of forms, a new attribute is introduced on the `form` element: `replace`. This attribute takes two values:

### document

The default value. The entire document (as specified by the `target` attribute when the document uses frames or windows) is replaced by the return value.

### values

The body returned from the server is treated as a new data file for prefiling the form.

***Note: These names, and their exact semantics, differ from those of the equivalent attribute in XForms 1.0 (the `replace` attribute on the `submission` element). The equivalent of `document` in this specification is `all` in XForms, and the equivalent of `values` is `instance`. The equivalent of the XForms `none` value is `document` with the server returning an HTTP 204 No Content return code.***

The exact semantics are described in detail in [the section on submission](#), under [step eight](#).

## 2.17. Extensions to the submit buttons

Normally, activating a submit button (an `input` or `button` element with the `type` attribute set to `submit`, or an `input` element with the `type` attribute set to `image`) must submit the first form the control is associated with, using the form's submission details (`action`, `method`, `enctype`, and `replace` attributes).

In some cases, authors would like to be able to submit a form to different processors, using different submission methods, or not replacing the form but just updating the details with new data. For this reason, the following attributes may be used on submit buttons: `action`, `method`, `enctype`, `replace`, and `target`.

If a submit button is activated, then the submission must use the values as given by the button that caused the activation. For missing attributes, the values of the equivalent attributes on the relevant `form` element, if any, are used instead.

If the submit button is not associated with any form, then no form is submitted.

## 2.18. Handling unexpected elements and values

There are several elements that are defined as expecting particular elements as children. Using the DOM, or in the markup, it is sometimes possible for authors to violate these expectations and place elements in unexpected places.

Similarly, attributes are defined to accept values that conform to certain syntaxes, but it is possible for authors to violate these constraints.

Authors must not do this. User agent implementors may curse authors who violate these rules, and may persecute them to the full extent allowed by applicable international law.

Upon encountering incorrect constructs, UAs must proceed as follows:

### For parsing errors in HTML

This document does not specify exact parsing semantics for ambiguous cases that are not covered by SGML. UA implementors should divine appropriate behaviour by reverse engineering existing products and attempting to emulate their behaviour. (This does not apply to XHTML, since the XML specification specifies mandatory formal error handling rules.)

### For non-empty `form` elements in `head` elements in XHTML

Typically UAs are expected to hide all the contents of `head` elements. No other special behaviour is required to cope with this case; if the author overrides this hiding (e.g. through CSS) then the form must behave like any other form. (This does not apply to HTML, where a `form` in a `head` would, per SGML parsing rules, imply a `body` start tag.)

### For non-empty `input` elements

By default, the form control must be rendered, and the contents of the element must not be. Using CSS3 Generated Content [\[CSS3CONTENT\]](#) or XBL [\[XBL\]](#), however, it is possible for the author to override this behaviour.

### For `output` elements containing elements in the DOM

The `defaultValue` DOM attribute [must be initialized](#) from the DOM3 Core `textContent` attribute ([\[DOM3CORE\]](#)). Setting the element's `value` attribute must be identical to setting the DOM3 Core `textContent` attribute. While the element contains elements, they are rendered according to the CSS rules.

### For `textarea` elements containing tags in HTML

The tags should be parsed as character data, but entities and comments should be recognised and handled correctly. (This doesn't apply to



XHTML, since it would contravene XML parsing rules.)

### **For `textarea` elements containing elements in the DOM**

The `defaultValue` DOM attribute is identical to the `textContent` DOM attribute both for reading and writing, and is used to set the initial `value`.

The rendering is based on the `value` DOM attribute, not the contents of the element, unless CSS is used to override this somehow.

### **For `select` and `optgroup` elements containing nodes other than `option` and `optgroup` elements**

Only the `option` and `optgroup` elements take part in the `select` semantics. Unless otherwise forced to appear by a stylesheet, other child nodes are never visible.

### **For `option` elements containing nodes other than text nodes**

The value of the control, if not specified explicitly, must be initialized using the `textContent` DOM attribute's value.

How such markup should be rendered is undefined. Two possibilities are sensible: rendering the content normally, just as it would have been outside the form control; and rendering the initial value (the value stored in the DOM `defaultValue` attribute) only, with the rest of the content not displayed (unless forced to appear through some CSS).

### **For `option` and `optgroup` elements that are not inside `select` elements**

The elements should be rendered much the same as `span` elements.

### **For `accept` attributes that are invalid**

The given value is what must be used in the DOM and elsewhere, but it must be interpreted as `*/*` when doing comparisons with actual MIME types, e.g. to filter a file list or when validating the control prior to submission.

### **For `value` attributes that are invalid according to the `type` attribute**

The attribute must be ignored. It will appear in the DOM for the `defaultValue` attribute, but will not be used as the value of the control. The control will therefore initially have [no value selected](#).

### **For `value` attributes that are invalid according to the `min`, `max`, `step`, `maxlength`, etc, attributes**

The control must be set to that value. The form will not submit with that value, though. If the control cannot be set to that value (for example, a range control cannot represent values outside its range) then the value must be clamped to the nearest value that can be represented by the control. (This situation can also arise if these attributes are dynamically updated using the DOM.)

### For labels pointing (via `for`) to elements that are not form controls

The attribute must be ignored. It will appear in the DOM (including as the value of `htmlFor`) but the `control` DOM attribute must return null and activating the label must not send focus to the associated element.

### For labels with no `for` attributes and containing more than one form control

The `control` DOM attribute must return, and activating the label, if supported, must transfer focus to or activate, the first control in a depth-first search of the label's children.

### For labels containing other labels

As events bubble from the target node, user agents must make the default action of the event be the action that would be appropriate for the first `label` element that the event bubbles through. In other respects, nested labels are oblivious to each other. Note that in HTML, parsers typically treat a `<label>` opening tag as implying a closing tag for any already open label.

### For other attributes that contain invalid values

The attribute must be ignored. It will appear in the DOM, but not affect the form semantics. For example, if a `min` attribute on a `datetime` control is an integer instead of a date and time string, then the range has no minimum. If the `type` attribute is then changed to `number`, then the attribute would take effect.

### For other instances of elements that violate their content models

The user agent processing model is defined in terms of the DOM and is independent of whether content models are being violated or not. Behaviour is therefore fully defined even for invalid content.

Other invalid cases should be handled analogously.

## 3. The repetition model for repeating form controls

Occasionally forms contain repeating sections. For example, an order form could have one row per item, with product, quantity, and subtotal fields. The **repeating form controls model** defines how such a form can be described without resorting to scripting.

*Note: The entire model can be emulated purely using JavaScript*

**and the DOM. With such a library, this model could be used and down-level clients could be supported before user agents implemented it ubiquitously. Creating such a library is left as an exercise for the reader.**

### 3.1. Introduction for authors

*This subsection is not normative.*

Occasionally, a form may need a section to be repeated an arbitrary number of times. For example, an order form could have one row per item. Traditionally, this has been implemented either by using complex client-side scripts or by sending a request to the server for every new row.

Using the mechanisms described in this section, the problem is reduced to describing a template in the markup, and then specifying where and when that template should be repeated.

To explain this, we will step through an example. Here is a sample form with three rows:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Sample Order Form</title>
  </head>
  <body>
    <form>
      <table>
        <tr>
          <th>Product</th>
          <th>Quantity</th>
        </tr>
        <tr>
          <td><input type="text" name="row0.product" value=""></td>
          <td><input type="text" name="row0.quantity" value="1"></td>
        </tr>
        <tr>
          <td><input type="text" name="row1.product" value=""></td>
          <td><input type="text" name="row1.quantity" value="1"></td>
        </tr>
        <tr>
          <td><input type="text" name="row2.product" value=""></td>
          <td><input type="text" name="row2.quantity" value="1"></td>
        </tr>
      </table>
      <p><button type="submit">Submit</button></p>
    </form>
  </body>
```

```
</html>
```

The template for those rows could look something like:

```
<tr>
  <td><input type="text" name="row0.product" value=""></td>
  <td><input type="text" name="row0.quantity" value="1"></td>
</tr>
```

...except that then the names would all be the same — all rows would be "row0", so there would be no clear way of distinguishing which "quantity" went with which "product" except by the order in which they were submitted.

To get around this, the template is modified slightly:

```
<tr id="order">
  <td><input type="text" name="row[order].product" value=""></td>
  <td><input type="text" name="row[order].quantity" value="1"></td>
</tr>
```

The template now has a unique identifier ("order"), and that identifier is used to indicate where the row index should be substituted in. When a template is replicated, all the attributes containing the template's id between square bracket characters (`[id]`) have that ID (and the brackets) replaced by a unique index.

***Note: To prevent an attribute from being processed in this way, put a non-breaking zero-width space character (&#xFEFF;) at the start of the attribute value. When the template is cloned, that character will be removed, but any other text in the attribute will be left alone. This could be useful if you have no control over the rest of the contents in the attribute, e.g. if it is user configurable text.***

In order to distinguish this row from a normal row, something needs to be added to the template to mark it as being a template. This is done using a repeat attribute:

```
<tr id="order" repeat="template">
  <td><input type="text" name="row[order].product" value=""></td>
  <td><input type="text" name="row[order].quantity" value="1"></td>
</tr>
```

Now we replace the table with that markup:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Sample Order Form</title>
  </head>
  <body>
```

```

<form>
  <table>
    <tr>
      <th>Product</th>
      <th>Quantity</th>
    </tr>
    <tr id="order" repeat="template">
      <td><input type="text" name="row[order].product" value=""></td>
      <td><input type="text" name="row[order].quantity" value="1"></td>
    </tr>
  </table>
  <p><button type="submit">Submit</button></p>
</form>
</body>
</html>

```

This will make one row appear, because by default templates repeat themselves once. We can control how many times the template repeats itself using the repeat-start attribute:

```

...
<tr id="order" repeat="template" repeat-start="3">
  <td><input type="text" name="row[order].product" value=""></td>
  <td><input type="text" name="row[order].quantity" value="1"></td>
</tr>
</table>

```

This is now identical to the original example (three rows with empty fields will appear). It still isn't dynamic — there is no way for the user to add more rows.

This can be solved by adding an add button. The add button type adds a copy of a template when the user presses the button.

There are two ways to use add buttons. The first is by explicitly specifying which template should be replicated:

```

<p><button type="add" template="order">Add Row</button></p>

```

The template is specified using a template attribute on the `button type="add"` or `input type="add"` elements. In the template attribute, you put the ID of the template you want the button to affect.

When such a button is pressed, the template is replicated, and the resulting block is inserted just after the last block that is associated with the template. For example, there are three rows in the example above, so if the user pressed that button, the new block would be inserted just after the third one.

The second way is by including an add button inside the template, so that when the template is replicated, the button is replicated into the resulting block. When such a button is pressed, the template is replicated, and inserted immediately

after the block in which the button is found. For example, if there were add buttons in the rows of the example above, and someone pressed the button in the first row, a row would be inserted between the first row and the second row.

For this example we will only do it the first way:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Sample Order Form</title>
  </head>
  <body>
    <form>
      <table>
        <tr>
          <th>Product</th>
          <th>Quantity</th>
        </tr>
        <tr id="order" repeat="template" repeat-start="3">
          <td><input type="text" name="row[order].product" value=""></td>
          <td><input type="text" name="row[order].quantity" value="1"></td>
        </tr>
      </table>
      <p><button type="add" template="order">Add Row</button></p>
      <p><button type="submit">Submit</button></p>
    </form>
  </body>
</html>
```

Now the user can add more rows, but he cannot remove them. Removing rows is done via the remove button type. When a user presses such a button, the row in which the button is kept is removed from the document.

```
<button type="remove">Remove This Row</button>
```

This is added to the template so that it appears on every row:

```
<tr id="order" repeat="template" repeat-start="3">
  <td><input type="text" name="row[order].product" value=""></td>
  <td><input type="text" name="row[order].quantity" value="1"></td>
  <td><button type="remove">Remove This Row</button></td>
</tr>
```

The final result looks like this:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Sample Order Form</title>
  </head>
  <body>
    <form>
      <table>
```

```

<tr>
  <th>Product</th>
  <th>Quantity</th>
</tr>
<tr id="order" repeat="template" repeat-start="3">
  <td><input type="text" name="row[order].product" value=""></td>
  <td><input type="text" name="row[order].quantity" value="1"></td>
  <td><button type="remove">Remove This Row</button></td>
</tr>
</table>
<p><button type="add" template="order">Add Row</button></p>
<p><button type="submit">Submit</button></p>
</form>
</body>
</html>

```

If the user pressed "Add" once, removed the middle two rows, typed in some garbage in the two "product" text fields, and pressed "Submit", the user agent would submit the following name-value pairs:

```

row0.product=some
row0.quantity=1
row3.product=gabrage
row3.quantity=1

```

Further examples are given in the [examples section](#) below.

### 3.1.1. More features

The repetition model supports more than just the cases given above; for instance, there are [move-up](#) and [move-down](#) buttons that can be inserted inside templates much like the [remove](#) button but for moving rows up and down.

Repetition templates can also be nested. The concept of hierarchy is expected to be represented in the names, as it is in manually-created repeating forms; for example:

```

order0.name
order0.quantity
order0.comment0.text
order0.comment1.text
order1.name
order1.quantity
order1.comment0.text

```

That way the submission can remain compatible with the long-established `multipart/form-data`, yet not lose the structure of the data.

***Note: The naming schemes used above are arbitrary. Any naming scheme could be used, at the convenience of the author.***

### 3.1.2. Suggestions for authors

To include prefilled rows in the document, include copies of the row with `repeat` attributes having the value of the repetition index you want to use for that block.

For example, here is a prefilled version of the earlier table, with one row having index "1" with the text "Tom figurine" in the first field and "12" in the second, and with a second row having index "2" and values "Jerry figurine" and "5".

```
...
<table>
  <tr>
    <th>Product</th>
    <th>Quantity</th>
  </tr>
  <tr repeat="1">
    <td><input type="text" name="row1.product" value="Tom figurine"
    <td><input type="text" name="row1.quantity" value="12"></td>
    <td><button type="remove">Remove This Row</button></td>
  </tr>
  <tr repeat="2">
    <td><input type="text" name="row2.product" value="Jerry figurine"
    <td><input type="text" name="row2.quantity" value="5"></td>
    <td><button type="remove">Remove This Row</button></td>
  </tr>
  <tr id="order" repeat="template" repeat-start="1">
    <td><input type="text" name="row[order].product" value=""></td>
    <td><input type="text" name="row[order].quantity" value="1"></td>
    <td><button type="remove">Remove This Row</button></td>
  </tr>
</table>
...
```

Prefilled rows must go *before* the template.

Prefilled rows can contain any content; it need not match the template. In order to be considered a part of the repetition model, however, the row must have a `repeat` attribute with a numeric value. That value can be any integer. (For example, you could use "-1" as the value of all prefilled rows.)

In HTML4-compliant UAs that do not implement this specification, the template acts as an initial blank row, and the "add" and "remove" buttons cause the form to be submitted, allowing the server to simulate the insertion and removal of rows.

**Note:** The add and remove buttons act as submit buttons in compliant HTML4 UAs only if `button` elements are used. If `input` buttons are used, then legacy UAs will instead render the controls



*as text fields. It is thus recommended that authors use `button` elements.*

### 3.1.3. What the repetition model can't do

This specification does not address the ability to select a repetition block to move it up or down without using buttons directly associated with the current block.

## 3.2. Definitions

*Note: This section makes a number of references to namespaces. For authors who are only using HTML or XHTML, the definitions below ensure that no namespaces need appear in the document (except the namespace on the root element). Thus, such a reader can simply gloss over the parts that mention namespaces.*

Several new global attributes are introduced as part of the repetition model: `repeat`, `repeat-start`, `repeat-min`, `repeat-max`, and `repeat-template`. When placed on elements in the `http://www.w3.org/1999/xhtml` namespace, they must be namespace-free attributes, and when placed on other elements, they must be attributes in the `http://www.w3.org/1999/xhtml` namespace.

The most important one is the `repeat` attribute. The effect of this attribute depends on its value, which can be either the literal string `"template"`, or an integer.

### 3.2.1. Repetition templates

An element in the `http://www.w3.org/1999/xhtml` namespace with the `repeat` attribute in no namespace, or an element in any other namespace with the `repeat` attribute in the `http://www.w3.org/1999/xhtml` namespace, with the attribute's value equal to `template`, is a **repetition template**.

Repetition templates may occur anywhere. They are not specifically associated with any form.

Every template has an index associated with it. The initial value of a template's index is always 0. The index is used to ensure that when cloning templates, the new block has a unique ID. The template's index does not appear in the markup. (It does, however, appear in the DOM, as the `repetitionIndex` attribute.)

Unrecognized values must be ignored.

```

<div repeat="template"/> <!-- A template. -->
<div repeat="template +1 3"/> <!-- Not a template. -->
<div repeat=" template"/> <!-- Not a template (leading whitespace). -->

```

Authors may use "title" attributes on templates (if they are in the HTML namespace) to describe their purpose. Assistive technologies may then use these descriptions to help users. (Note, though, that such titles would get propagated to the repetition blocks too.)

### 3.2.2. Repetition blocks

An element in the <http://www.w3.org/1999/xhtml> namespace with the [repeat](#) attribute in no namespace, or an element in any other namespace with the [repeat](#) attribute in the <http://www.w3.org/1999/xhtml> namespace, with the attribute's value equal to an integer (an optional leading '-' character followed by one or more decimal digits), is a **repetition block**.

Repetition blocks may have a [repeat-template](#) attribute to specify a template that the block is to be associated with. If the document contains an element with an ID equal to the value of the [repeat-template](#) attribute, and that element is a [repetition template](#), then that is the template that the repetition block is associated with. (In the case of duplicate IDs, the behaviour should be the same as with `getElementById()`.) Otherwise, if the [repeat-template](#) attribute does not point to a repetition template, then the element is not associated with a template.

Repetition blocks without a [repeat-template](#) attribute are associated with their first following sibling that is a repetition template, if there is one.

Repetition blocks that don't have an associated template are called **orphan repetition blocks**. They can take part in the deletion and movement aspects of the repetition model, but not addition.

Every repetition block has an index associated with it. The index's initial value is the value of the [repeat](#) attribute.

```

<div>
  <div repeat="0"/> <!-- A simple repetition block, index 0. -->
  <div repeat="-5"/> <!-- Another, index -5 -->
  <div repeat="2"/> <!-- A simple repetition block, index 2. -->
  <div repeat="nothing"/> <!-- Just a normal element. -->
  <div repeat=" 3"/> <!-- Another normal element (leading whitespace). --
  <div repeat="template"/> <!-- The template for the last few elements. .
  <div repeat="1"/> <!-- Orphan repetition block, index 1. -->
</div>
<div repeat="0"/> <!-- Orphan repetition block, index 0. -->

```

### 3.3. New form controls

Several new button types are introduced to support the repetition model. These values are valid types for both the `button` element and the `input` element.

**add**

Adds a new repetition block.

**remove**

Removes the nearest ancestor repetition block.

**move-up**

Moves the nearest ancestor repetition block up one.

**move-down**

Moves the nearest ancestor repetition block down one.

These control types can never be [successful](#).

When these new types are used with `input` buttons, the `value` attribute shall, if present, provide the button caption (although this may be further overridden by the stylesheet). When the `value` attribute is absent, the buttons should be given locale-dependent default labels, in the same way as `submit` and `reset` buttons.

However, user agents may instead render buttons consistent with those performing equivalent functions in the user's operating environment. For this reason, authors who are nesting repetition blocks should position such buttons carefully to make clear which block a button applies to.

Invoking these buttons generates events (such as `click`), as specified by the DOM specifications. The default action for these events is to act as described below. However, if the event is canceled, then the default action will not occur.

In addition, to support the `add` type, a new attribute is introduced to the `button` and `input` elements: `template`.

**template**

Specifies the repetition template to use.

These new types and attributes are described in more detail in the next few sections.

### 3.4. The `repeat-min` and `repeat-max` attributes

The `repeat-min` attribute specifies the number of repetition blocks that the `remove` button type will ensure are present each time a block is removed. Its

value must be a positive integer (one or more digits 0-9 interpreted as a base ten number). If the attribute is omitted or if it has an invalid value then it is treated as if its value was zero.

The `repeat-max` attribute specifies the maximum number of repetition blocks that the `add` button type can cause to be present. Its value must be a positive integer (one or more digits 0-9 interpreted as a base ten number). If the attribute is omitted or if it has an invalid value then there is no limit.

These two attributes have no effect on the repetition model when present on elements that do not have a `repeat` attribute with the value set to `template`.

### 3.5. Event interface for repetition events

The repetition model includes several events. The following interface is used by these events.

```
/* Similar to the UIEvent interface */
interface RepetitionEvent : Event {
  readonly attribute RepetitionElement element;
  void initRepetitionEvent(in DOMString typeArg,
                           in boolean canBubbleArg,
                           in boolean cancelableArg,
                           in RepetitionElement elementArg);
  void initRepetitionEventNS(in DOMString namespaceURIArg,
                             in DOMString typeArg,
                             in boolean canBubbleArg,
                             in boolean cancelableArg,
                             in RepetitionElement elementArg);
};
```

The `initRepetitionEvent()` and `initRepetitionEventNS()` methods have the same behaviours as the `initEvent()` and `initEventNS()` events from [\[DOM3EVENTS\]](#).

The context information for repetition events must be given in the `element` attribute.

### 3.6. The repetition model

A [repetition template](#) should not be displayed. In CSS-aware user agents, this should be achieved by including the following rules, or their equivalent, in the UA's user agent stylesheet:

```
@namespace xh url(http://www.w3.org/1999/xhtml);
:not(xh|*) [xh|repeat="template"],
```

```
xh|* [|repeat="template"] { display: none; }
```

Any form controls inside a [repetition template](#) are associated with their forms' `templateElements` DOM attributes, and are *not* present in the forms' `elements` DOM attributes. Since controls in the `templateElements` attribute cannot be [successful](#), controls inside repetition templates can never be submitted. They also cannot be prefilled directly when the form is preseeded. However, see the section on [seeding a form with initial values](#) for details on how repetition blocks can be prefilled.

### 3.6.1. Addition

If an `add` button with a `template` attribute is activated, and its `template` attribute gives the ID of an element in the document that is a [repetition template](#) as defined above, then that template's replication behaviour is invoked.

(Specifically, in scripting-aware environments, the template's `addRepetitionBlock()` method is called with a null argument.) In the case of duplicate IDs, the behaviour should be the same as with `getElementById()`.

If an `add` button *without* a `template` attribute is activated, and it has an ancestor that is a [repetition block](#) that is not an orphan repetition block, then the [repetition template](#) associated with that repetition block has its template replication behaviour invoked with the respective repetition block as its argument.

(Specifically, in scripting-aware environments, the template's `addRepetitionBlock()` method is called with a reference to the DOM Element node that represents the repetition block.)

When a template's replication behaviour is invoked (specifically, when either its `addRepetitionBlock()` method is called or its `addRepetitionBlockByIndex()` method is called) the following is performed:

1. If the template has no parent node or its parent node is not an element, then the method must abort the steps and do nothing.
2. The template examines its preceding siblings, up to the start of the parent element. For each sibling that is a [repetition block](#) whose associated template is this template, if the repetition block's index is greater than or equal to the template's index, then the template's index is increased to the repetition block's index plus one. The total number of repetition blocks associated with this template that were found is used in the next step.
3. If the repetition template has a `repeat-max` attribute and that attribute's value is less than or equal to the number of repetition blocks associated with this template that were found in the previous step, the UA must stop at this step, returning a null value.

4. If this algorithm was invoked via the `addRepetitionBlockByIndex()` method, and the value of the method's index argument is greater than the template's index, then the template's index is set to the value of the method's index argument.
5. A clone of the template is made. The resulting element is the new repetition block element.
6. If this algorithm was invoked via the `addRepetitionBlockByIndex()` method, the new repetition block element's index is set to the method's index argument. Otherwise, the new repetition block element's index is set to the template's index.
7. If the new repetition block element is in the `http://www.w3.org/1999/xhtml` namespace, then the `repeat` attribute in no namespace on the cloned element has its value changed to the new block's index. Otherwise, the `repeat` attribute in the `http://www.w3.org/1999/xhtml` namespace has its value changed to the new block's index.
8. If the new repetition block element is in the `http://www.w3.org/1999/xhtml` namespace, then any `repeat-min`, `repeat-max`, or `repeat-start` attributes in no namespace are removed from the element. Otherwise, any `repeat-min`, `repeat-max`, or `repeat-start` attributes in the `http://www.w3.org/1999/xhtml` namespace are removed instead.
9. If the new repetition block has an ID attribute (that is, an attribute specifying an ID, regardless of the attribute's namespace or name), then that attribute's value is used as the template name in the following steps. Otherwise, the template has no name. (If there is more than one ID attribute, the "first" one in terms of [node order](#) is used. [\[DOM3CORE\]](#))
10. If the template has a name (see the previous step), and that name contains either an opening square bracket (U+005B, "[") a modifier letter half triangular colon (U+02D1, "⋮"), a closing square bracket (U+005D, "]") or a middle dot (U+00B7, "⋅"), then the template's name is ignored for the purposes of the next step.
11. If the template has a name and it is not being ignored (see the previous two steps), then, for every attribute on the new element, and for every attribute in every descendant of the new element: if the attribute starts with a zero-width non-breaking space character (U+FEFF) then that character is removed from the attribute; otherwise, any occurrences of a string consisting of an opening square bracket (U+005B, "[") or a modifier letter half triangular colon (U+02D1, "⋮"), the template's name, and a closing square bracket (U+005D, "]") or a middle dot (U+00B7, "⋅"), are replaced by the new repetition block's index. This is performed regardless of the

types, names, or namespaces of attributes, and is done to *all* descendants, even those inside nested forms, nested repetition templates, and so forth.

For example, if the template is called `order`, and the new repetition block's index has the value 2, and one of the attributes of one of the descendants of the new repetition block is marked up as `name="order.[order].comment.[comment[order]]"`, then the attribute's value is changed to `order.2.comment.[comment2]`. However, if the attribute was written as `name="&#xFEFF;order.[order]"`, then its value would have only been changed to `order[order]`, only removing the leading character.

The characters don't have to be paired, so for instance, `name="field[n·"` will cause the repetition block's index to be put in place of the `"[n·` string (assuming the template is called `"n"`).

***Note: The recommended characters to use for readability are the square brackets [ ], but the two dot-like characters are allowed as well because those two characters happen to be valid in XML IDs while the square brackets are not.***

12. If the template has a name (see the earlier steps): If the new repetition block element is in the `http://www.w3.org/1999/xhtml` namespace, then the `repeat-template` attribute in no namespace on the cloned element has its value set to the template's name. Otherwise, the `repeat-template` attribute in the `http://www.w3.org/1999/xhtml` namespace has its value set to the template's name. (This happens even if the name was ignored for the purposes of the previous step.)
13. The attribute from which the template's name was derived, if any, and even if it was ignored, is removed from the new repetition block element. (See the previous four steps.)
14. If the first argument to the method was null, then the template once again crawls through its previous siblings, this time stopping at the first node (possibly the template itself) whose previous sibling is a repetition block (regardless of what that block's template is) or the first node that has no previous sibling, whichever comes first. The new element is inserted into the parent of the template, immediately before that node. Mutation events are fired if appropriate.
15. Otherwise, the new element is inserted into the parent of the node that was passed to the method as the first argument, immediately *after* that node (before the node's following sibling, if any). Mutation events are fired

if appropriate.

16. The template's index is increased by one.
17. An `added` event in the `http://www.w3.org/2001/xml-events` namespace, which bubbles but is not cancelable and has no default action, is fired on the repetition template using the `RepetitionEvent` interface, with the repetition block's DOM node as the context information.
18. The return value is the newly cloned element.

In addition, user agents must automatically disable `add` buttons (irrespective of the value of the `disabled` DOM attribute) when the buttons are not in a repetition block that has an associated template and their `template` attribute is either not specified or does not have an ID that points to a repetition template, and, when the repetition template's `repeat-max` attribute is less than or equal to the number of repetition blocks that are associated with that template and that have the same parent. This automatic disabling does not affect the DOM `disabled` attribute. It is an intrinsic property of these buttons.

For an example, see the [example section](#) below.

### 3.6.2. Removal

If a `remove` button is activated, and the element has an ancestor that is a [repetition block](#) as defined above, then the nearest such ancestor's template deletion behaviour is invoked. (Specifically, in scripting-aware environments, the element's `removeRepetitionBlock()` method is invoked.)

When a repetition block's deletion behaviour is invoked (specifically, when its `removeRepetitionBlock()` method is called) the following is performed:

1. The node is removed from its parent, if it has one. Mutation events are fired if appropriate. (This occurs even if the repetition block is an orphan repetition block.)
2. If the repetition block is not an orphan, a `removed` event in the `http://www.w3.org/2001/xml-events` namespace, which bubbles but is not cancelable and has no default action, is fired on the element's repetition template, using the `RepetitionEvent` interface, with the repetition block's DOM node as the context information.
3. If the repetition block is not an orphan, then while the remaining number of repetition blocks associated with the original element's repetition template and with the same parent as the template is less than the template's `repeat-min` attribute and less than its `repeat-max` attribute, the template's



replication behaviour is invoked (specifically, its `addRepetitionBlock()` method is called).

In addition, user agents must automatically disable `remove` buttons (irrespective of the value of the `disabled` DOM attribute) when the buttons are not in a repetition block. This automatic disabling does not affect the DOM `disabled` attribute. It is an intrinsic property of these buttons.

For an example, see the [example section](#) below.

### 3.6.3. Movement of repetition blocks

The two remaining button types, `move-up` and `move-down`, are used to move repetition blocks up or down past its sibling repetition blocks.

If a `move-up` or `move-down` button is activated, and the element has an ancestor that is a [repetition block](#) as defined above, then the nearest such ancestor's template movement behaviour is invoked in the relevant direction. (Specifically, in scripting-aware environments, the element's `moveRepetitionBlock()` method is called; for `move-up` buttons the argument is -1 and for `move-down` buttons the argument is 1).

When a repetition block's movement behaviour is invoked (specifically, when its `moveRepetitionBlock()` method is called) the following is performed, where *distance* is an integer representing how far and in what direction to move the block (the argument to the method):

1. If *distance* is 0, or if the repetition block has no parent, nothing happens and the algorithm ends here.
2. Set *target*, a reference to a DOM Node, to the repetition block being moved.
3. If *distance* is negative: while *distance* is not zero and *target*'s `previousSibling` is defined and is not a [repetition template](#), set *target* to this `previousSibling` and, if it is a [repetition block](#), increase *distance* by one (make it less negative by one).
4. Otherwise, *distance* is positive: while *distance* is not zero and *target*'s `nextSibling` is defined and is not a [repetition template](#), set *target* to this `nextSibling` and, if it is a [repetition block](#), decrease *distance* by one. After the loop, set *target* to *target*'s `nextSibling` (which may be null).
5. Call the repetition block's parent node's `insertBefore()` method with the `newChild` argument being the repetition block and the `refChild` argument being *target* (which may be null by this point). Mutation events are fired if

appropriate.

6. A `moved` event in the `http://www.w3.org/2001/xml-events` namespace, which bubbles but is not cancelable and has no default action, is fired on the element's repetition template (if it has one), using the `RepetitionEvent` interface, with the repetition block's DOM node as the context information.

This occurs even if the repetition block is an orphan repetition block (although if it is, the event is not fired).

Moving repetition blocks does not change the index of the repetition blocks.

In addition, user agents must automatically disable `move-up` buttons (irrespective of the value of the `disabled` DOM attribute) when their repetition block could not be moved any higher according to the algorithm above, and when the buttons are not in a repetition block. Similarly, user agents must automatically disable `move-down` buttons when their repetition block could not be moved any lower according to the algorithm above, and when the buttons are not in a repetition block. This automatic disabling does not affect the DOM `disabled` attribute. It is an intrinsic property of these buttons.

#### 3.6.4. Initial repetition blocks

The `repeat-start` attribute on repetition templates is used to insert repetition blocks at load time without having to explicitly copy the repetition template markup in the source document or use scripting.

When present, the attribute must contain one or more digits 0-9 interpreted as a base ten integer. If the value is not in this format, or if the attribute is omitted, the value "1" is used instead.

Before `load` events are fired, but after the entire document has been parsed and after forms with `data` attributes are prefilled (if necessary), UAs must iterate through every node in the document, depth first, looking for templates so that their initial repetition blocks can be created. For each element that has a `repeat` attribute with the literal value `template`, the UA must invoke the template's replication behaviour as many times as the `repeat-start` attribute on the same element specifies (just once, if the attribute is missing or has an invalid value). Then, while the number of repetition blocks associated with the repetition template is less than the template's `repeat-min` attribute, the template's replication behaviour must be further invoked. (Invoking the template's replication behaviour means calling its `addRepetitionBlock()` method).

The above step must be skipped for any element in the document whose parent node is not an element. (In particular, nothing happens if `repeat-start` or `repeat-min` are set on the root element.)

UAs should not specifically wait for images and stylesheets to be loaded before creating initial repetition blocks as described above.

### 3.6.5. Notes for assistive technologies

The repetition templates and blocks are present in the DOM, and all modifications to these blocks (whether via the DOM or via the buttons described above) result in mutation events being fired on the document. Assistive technologies can therefore use normal DOM navigation and mutation event listeners to help present repetition sections to the user.

## 3.7. Examples

This section gives some more practical examples of repetition.

### 3.7.1. Repeated rows

The following example shows how to use repetition templates to dynamically add rows to a form in a table.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Form Repeat Demo</title>
  </head>
  <body>
    <form action="http://software.hixie.ch/utilities/cgi/test-tools/echo"
      <table>
        <thead>
          <tr>
            <th>Name</th>
            <th>Number of Cats</th>
            <th></th>
          </tr>
        </thead>
        <tbody>
          <tr repeat="0">
            <td><input type="text" name="name_0" value="John Smith"></td>
            <td><input type="text" name="count_0" value="2"></td>
            <td><button type="remove">Delete Row</button></td>
          </tr>
          <tr repeat="template" id="row">
            <td><input type="text" name="name_[row]" value=""></td>
            <td><input type="text" name="count_[row]" value="1"></td>
            <td><button type="remove">Delete Row</button></td>
          </tr>
        </tbody>
      </table>
      <p>
        <button type="add" template="row">Add Row</button>
      </p>
    </form>
  </body>
</html>
```

```

        <button type="submit">Submit</button>
    </p>
</form>
</body>
</html>

```

Initially, two rows would be visible, each with two text input fields, the first row having the values "John Smith" and "2", the second row having the values "" (a blank text field) and "1". The second row is the result of the (implied) `repeat-start` attribute adding a repetition block when the document was loaded.

If the "Add Row" button is pressed, a new row is added. The first such row would have the index 2 (since there are already two repetition blocks numbered 0 and 1) and so the controls would be named "name\_2" and "count\_2" respectively.

If the "Delete Row" button above is pressed, the row is removed.

### 3.7.2. Nested repeats

The previous example does not demonstrate nested repeat blocks, reordering repetition blocks, and inserting new repetition blocks in the middle of the existing sequence, all of which are possible using the facilities described above.

This example shows nested repeats.

```

<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Solar System</title>
  </head>
  <body>
    <form>
      <h1> Solar system </h1>
      <p> <label> System Name: <input name="name"/> </label> </p>
      <h2> Planets </h2>
      <ol>
        <li repeat="template" repeat-start="0" id="planets">
          <label> Name: <input name="planet[planets].name" required="required" />
          <h3> Moons </h3>
          <ul>
            <li repeat="template" repeat-start="0" id="planet[planets].moons">
              <input name="planet[planets].moon[planet[planets].moons]" />
              <button type="remove">Delete Moon</button>
            </li>
          </ul>
          <p><button type="add" template="planet[planets].moons">Add Moon</button>
          <p><button type="remove">Delete Planet</button></p>
        </li>
      </ol>
      <p><button type="add" template="planets">Add Planet</button></p>
      <p><button type="submit">Submit</button></p>
    </form>
  </body>
</html>

```

```
</form>
</body>
</html>
```

Note that to uniquely identify each nested repeat (which is required since the `add` buttons are dependent on IDs to specify which template should have a block added), the IDs of the nested templates are specified in terms of the ancestor template's ID, using the index substitution feature.

***Note: Since square brackets are not allowed in ID attributes in XML, the above example cannot validate to an XML DTD. It is still well-formed, however, and conformant to this specification. It is possible to make a version that validates according to an XML DTD by using U+02D1 MODIFIER LETTER HALF TRIANGULAR COLON and U+00B7 MIDDLE DOT characters instead of the U+005B OPENING SQUARE BRACKET and U+005D CLOSING SQUARE BRACKET characters in the value of the second "id" attribute, without any other changes.***

## 4. The forms event model

**Form events** are those that are specifically fired on `form` elements and form controls as part of the forms event model. The following events are considered form events:

- {"http://www.w3.org/2001/xml-events", "change"}
- {"http://www.w3.org/2001/xml-events", "formchange"}
- {"http://www.w3.org/2001/xml-events", "input"}
- {"http://www.w3.org/2001/xml-events", "forminput"}
- {"http://www.w3.org/2001/xml-events", "invalid"}
- {"http://www.w3.org/2001/xml-events", "submit"}
- {"http://www.w3.org/2001/xml-events", "reset"}
- {"http://www.w3.org/2001/xml-events", "received"}
- {"http://www.w3.org/2001/xml-events", "DOMControlValueChanged"}

Some of these events are described in [\[DOM3EVENTS\]](#) and [\[HTML4\]](#). This

section introduces the new events and new semantics for the existing events.

***Note: The repetition model events are not form events since they are not form-specific.***

#### 4.1. The `click` event and `input` controls

During the handling of a `click` event on an `input` element with a `type` attribute that has the value "`radio`" or "`checkbox`", user agents must change the value of the `checked` property of the elements affected before the event is dispatched in the document. The default action of the `click` event in this case is to fire a `DOMActivate` event, and the default action of the `DOMActivate` event is to fire the `change` event (assuming the activation did change the value). If the default action of any of these events is canceled, the value of the properties changed must be changed back to their original value.

This is a change from DOM2 HTML where [this behaviour was optional](#).  
[\[DOM2HTML\]](#)

#### 4.2. The `change` and `input` events

In [\[DOM3EVENTS\]](#) and [\[HTML4\]](#), the `change` event is fired on a form control element when the control loses the input focus and its value has been modified since gaining focus.

This specification changes these semantics to more accurately reflect the behaviour expected by Web authors. Specifically, for controls implemented with a non-editable stateful UI (e.g. `select` elements, checkboxes, or radio buttons as deployed in typical desktop Web browsers), the `change` event shall be fired when the selection is completed, even if the control does not lose focus. (For example, if a `date` control has a button which shows a calendar, then the `change` event would fire when the calendar is closed after the user selected a date.)

In addition, to address the need for even more immediate feedback mechanisms, this specification introduces the `input` event. This event is fired on a control whenever the value of the control changes due to input from the user, and is otherwise identical to the `change` event. (For example, it bubbles, is not cancelable, and has no context information.)

UAs may delay firing the `input` event if the data entry is rapid. Authors must not rely on this event firing once for each key press, mouse input, or similar.

When the `change` event is fired at the same time as the `input` event, the `input`

event must be fired first.

Change and input events must never be triggered by scripted changes to the control value. Thus, loops caused by change event handlers triggering changes are not usually possible.

Any element that accepts an `onchange` attribute to handle `change` events shall also accept an `oninput` attribute to handle `input` events, except the output element (for which it would make no sense).

### 4.3. Events to enable simpler dependency tracking

Sometimes form controls are inter-dependent. In these cases, it is more intuitive to specify the dependencies on the control whose value or attributes depend on another's, rather than specify which controls should be affected by a change on the element that changes. For this reason, two new events are introduced, `formchange` and `forminput`.

These events are in the same namespace as the other [form events](#), do not bubble, cannot be canceled, have no context information, and have no default action.

The default action of a `change` event on most elements is to fire a `formchange` event at each element in the form's elements, in document order, and finally at the form itself. Note that template controls are not affected. If authors need this event to affect template controls, they should hook into the form's `onformchange` event handler.

The exception is the output control. Since changing an output control is often the result of a `formchange` event, `change` events whose target is an output control do not cause `formchange` events to be dispatched.

The `input` event analogously invokes the `forminput` event as its default action (again, except if the target is an output control).

When a form is reset, a `formchange` event is fired on all the form controls of the form in the same way.

**Note:** If authors wish to use `formchange` events to set up the initial state of the form (for forms with complex dependencies), they can either write a script to call `dispatchFormChange()` on the form, or set the `data` attribute of the form to

```
data:application/xml,%3Cformdata%20xmlns%3D%22http%3A%2F%2Fn.whatwg.org%2Fformdata%22%20type%3D%22incremental%22%2F%3E,
```

*which is equivalent to seeding a form with no new values.*

## 4.4. Form validation

With the introduction of the various type-checking mechanisms, some way for scripting authors to hook into the type-checking process is required. This is provided by the `change` event and the new `invalid` event (in the <http://www.w3.org/2001/xml-events> namespace).

Authors should use the `change` and `input` events to specify [custom validation code](#). Authors who use custom validation code should keep the validity flags up to date at all times, so that the `:invalid` pseudo class correctly matches elements as they go in and out of valid states, and so that UAs can provide up to date hints to the user to help him fill in the form.

When [a form is submitted](#), user agents must act as if they used the following algorithm. First, each element in that form's `elements` list is added to a temporary list (note that the `elements` list is defined to be in document order). Then, each element in this list whose `willValidate` DOM attribute is true is checked for validity, and an `invalid` event is fired on each element that, when checked, is found to fail to comply with its constraints (i.e. each element whose `validity.valid` DOM attribute is false) and is still a member of the form after the event has been handled.

***Note: This definition implies a defined behaviour in the face of event handlers that mutate the document. For example, if one control's `oninvalid` attribute changes a later control's value from invalid to valid, the event is not fired on that later control, yet in the reverse case, if a later form control's `oninvalid` attribute changes an earlier control, then the earlier control is not checked again. Controls added to the form during the process will not have any events fired, even if their value is invalid. Controls invalid at the start of the process that are removed from the form before receiving their events simply don't receive the event. Controls that change from one invalid state to another invalid state before receiving their event receive an event that describes their state at the point in the process at which they were checked and had the event fired on them.***

The event can also be fired if the `checkValidity()` method [of a form](#) or [form control](#) is invoked via script.

The `oninvalid` attribute (on `input`, `textarea`, and `select` elements) can be used



to write handlers for this event.

This event bubbles and is cancelable.

The default action is UA-specific, but is expected to consist of focusing the element (possibly firing focus events if appropriate), and alerting the user (ideally using a non-modal mechanism such as a help balloon) that the entered value is unacceptable in the user's native language along with explanatory text saying *why* the value is currently invalid.

If the reason the control is invalid is an author defined [custom error](#) then the message that the author provided using the [setCustomValidity\(\)](#) method should be used.

If the event was fired during form submission or as a result of the form's [form.checkValidity\(\)](#) method being called, UAs would typically only focus the first form control found to be invalid, although UAs are encouraged to give the user an idea of what other fields are invalid. The event shall, however, still be dispatched to all invalid controls whose [willValidate](#) DOM attribute is true. If the event was fired as a result of a control's [checkValidity\(\)](#) method, then the default action is performed regardless of whether the script has checked other controls as well.

If the element causing trouble is not visible (for example, a field made invisible using CSS or a field of type [hidden](#)) then the UA may wish to indicate to the user that there may be an error with the page's script.

When a radio group has no checked radio button and more than one of the radio buttons is marked as [required](#), the UA should only tell the user that the radio group as a whole is missing a value, not complain about each radio button in turn, even though all of the radio buttons marked with the [required](#) attribute would have the `missingValue` flag set.

***Note: Authors are encouraged to either cancel all [invalid](#) events (if they wish to handle the error UI themselves) or to not cancel any (if they wish to leave the error UI to the UA). Canceling one [invalid](#) event and reporting the error via script does not prevent the UA from handling another [invalid](#) event, possibly confusing the user by having two separate errors reported simultaneously in different ways. To cancel all events, a single capturing listener could be placed on the root element node canceling the default action of all [invalid](#) events.***

The following example shows one way to use this event.

```
<form action="..." method="post">
```

```

<p>
  <label>
    Byte 1:
    <input name="byte" type="number" min="0" max="255" required="req
      oninvalid="failed(event)" />
  </label>
  <output name="error"/>
</p>
<script type="text/javascript"> <![CDATA[
  function failed(event) {
    // a control can fail for more than one reason; only report one
    form.error.value = 'The value is wrong for a reason I did not e
    if (event.target.validity.typeMismatch)
      form.error.value = 'You must enter a number.';
    else if (event.target.validity.stepMismatch)
      form.error.value = 'Fractional numbers are not allowed.';
    else if (event.target.validity.rangeUnderflow)
      form.error.value = 'The number must be zero or greater.';
    else if (event.target.validity.rangeOverflow)
      form.error.value = 'The number must be 255 or less.';
    else if (event.target.validity.valueMissing)
      form.error.value = 'You must enter a number.';
    event.preventDefault(); /* don't want the UA to do its own repo
  }
}]> </script>
</form>

```

## 4.5. Receiving the results of form submission

The **ReceivedEvent** interface is used in the form submission process to handle the results of form submission.

```

interface ReceivedEvent : Event {
  readonly attribute Document receivedDocument;
  void          initReceivedEvent(in DOMString typeArg,
                                   in boolean canBubbleArg,
                                   in boolean cancelableArg,
                                   in Document documentArg);
  void          initReceivedEventNS(in DOMString namespaceURIArg,
                                   in DOMString typeArg,
                                   in boolean canBubbleArg,
                                   in boolean cancelableArg,
                                   in Document documentArg);
};

```

The **initReceivedEvent()** and **initReceivedEventNS()** methods have the same behaviours as the **initEvent()** and **initEventNS()** events from [\[DOM3EVENTS\]](#).

The **receivedDocument** attribute (set from the *documentArg* argument) contains a

reference to the document that was the result of the form submission. If the result cannot be represented as a DOM document, then the attribute is null. The document is mutable.

## 4.6. The `DOMControlValueChanged` event

A `DOMControlValueChanged` event, which bubbles, has no default action, and uses the basic `Event` interface, is fired on a control whenever its value changes, for whatever reason (including a scripted change).

This event has no corresponding event handler attribute and is primarily intended for use by assistive technologies.

## 5. Form submission

Processors conforming to this specification must use a slightly different algorithm than the [\[HTML4\]](#) form submission algorithm (HTML4 section 17.13.3), as described in this section.

When the user agent submits a form, it must perform the following steps.

### 1. Step one: Dispatch the `submit` event

If the submission was not initiated using the `submit()` method then the `submit` event is submitted as described in [\[HTML4\]](#). If it is canceled, then the submission processing stops at this point. If it is not canceled, then its default action is to perform the rest of the submission procedure.

### 2. Step two: Check the validity of the form

If the form submission was initiated as a result of a `submit` event's default action, then the form is [checked for validity](#). If, after the form has had any relevant `invalid` events fired, any controls remain invalid, then the submission shall be aborted.

Otherwise, if the form submission was initiated via the `submit()` method, then instead of firing `invalid` events, a `SYNTAX_ERR` exception shall be raised (and submission is aborted) if any of the controls are invalid. [\[DOM3CORE\]](#)

Script authors who wish to validate the form before performing submission can use script such as:

```
if (form.checkValidity())  
    form.submit();
```

...which will cause the UA to report the errors to the user, exactly as if the user had clicked a submit button.

### 3. Step three: Identify all form controls

All the controls that apply to the form should be identified, in document order. These controls are all those whose `forms` DOM attribute have an entry that points at the form and that are not in the form's `templateElements` DOM attribute (this excludes certain controls as specified in the section describing the repetition model, but does include the image controls that are excluded in the definition of the `elements` DOM attribute).

### 4. Step four: Build a form data set

A **form data set** consists of a form control list and a repetition block list.

The form control list is a sequence of *control-name*, *control-index*, *current-value* triplets constructed from the controls identified in the previous step.

***Note: The control index here is unrelated to the repetition index mentioned earlier.***

It is constructed by iterating over the form controls listed in step three, taking note of the form control names as they are seen. With each control, if it is the first time that control's name has been seen, then the control is assigned a control index of 0. Otherwise, if the control name was associated with an earlier control, then the index assigned is exactly one more than the last control with that name. Even unsuccessful controls and controls with [no value selected](#) are so numbered (including `output` elements). However, only [successful controls](#) are added to the form data set.

Successful controls have exactly one value, except for `select` controls and file upload controls, which have zero or more values depending on how many items or files they have selected. A successful control with more than one value is added multiple times, one for each value (each time with the same form control name and form control index). A successful control with zero values is omitted from the form data set.

Image buttons, during this step, must be handled as if they were two controls, one with the control's name with `.x` appended, whose value is the x coordinate selected by the user, and the other with the control's

name with `.y` appended, whose value is the `y` coordinate selected by the user. If the control's name is the empty string (e.g. if the `name` attribute is omitted) then the names `x` and `y` must be used instead. The indices of these two virtual controls are handled separately and could, depending on the names of other controls, end up with different values.

For example, the following form:

```
<form>
  <p> <label> Name: <input type="text" name="username"/> </label>
  <p> Lottery numbers:
    <input name="number" type="number" min="1" max="49"/>
    <input name="number" type="number" min="1" max="49"/>
    <input name="number" type="number" min="1" max="49"/>
    <input name="number" type="number" min="1" max="49"/>
    <input name="number" type="number" min="1" max="49"/>
  </p>
  <p>
    <label>
      Games:
      <select name="type" multiple="multiple">
        <option value="Thunderbolt"> Thunderbolt </option>
        <option value="Lightning"> Lightning </option>
      </select>
    </label>
  </p>
  <p>
    <input type="submit" value="Send"/>
  </p>
</form>
```

...if filled in with the name "Erwin" and the numbers 20, 30 and 40 with the first and last number fields left blank, and all the values in the select list selected, would generate the following form data set:

1. username, 0, "Erwin"
2. number, 0, ""
3. number, 1, "20"
4. number, 2, "30"
5. number, 3, "40"
6. number, 4, ""
7. type, 0, "Thunderbolt"
8. type, 0, "Lightning"

The form data set also includes a list of which [repetition blocks](#) are involved in the submission.

For each control in the form data set, the control and the control's ancestors are examined, up to but not including the first node that is a common ancestor of the control and the form, or is the form itself. For each element so examined, if it is a [repetition block](#) that is not an orphan repetition block and whose template does have an ID, and that repetition block has not yet been added to the list of repetition blocks, it is added.

## 5. Step five: Encode the form data set

The form data set is then encoded according to the content type specified by the `method` and `enctype` attributes of the element that caused the form to be submitted. See the [submitting the encoded form data set](#) section for details on how the `action` and `enctype` attributes are to be treated. The possible values of `enctype` defined by this specification are:

`application/x-www-form-urlencoded`

Described [below](#).

`multipart/form-data`

Described in [\[HTML4\]](#) ([section 17.13.4](#)) and [\[RFC2388\]](#). Note that this submission method discards the control index and repetition block parts of the form data set.

`application/x-www-form+xml`

Described [below](#).

`text/plain`

Described [below](#).

### Attribute not specified

Described [below](#).

Other values may be defined by other specifications.

During this step, the form data set is [examined to ensure all the characters are representable](#) in the submission character encoding.

When Unicode encodings are used, text in the submission should be converted to [normalisation form C](#) (NFC) before encoding the data set. [\[UNICODE\]](#)

## 6. Step six: Submit the encoded form data set

The encoded data is sent to the processing agent designated by the

`action` attribute of the element that initiated the submission, using the protocol method specified by the `method` attribute of that same element. If either of the attributes is missing from that element, and that element is not a `form`, then the relevant attribute on the element's associated `form` element is used instead. (If it is associated with multiple forms, the first one in the order given in the `form` attribute is used.) The [submitting the encoded form data set](#) section describes this in more detail.

## 7. Step seven: Dispatch the `received` event.

This step must be skipped if the form has no `onreceived` attribute. If this step is not skipped, then it defeats any attempt at incremental rendering, as the entire return value from the server must be downloaded and parsed before the event is fired (unless the user agent instantiates the document lazily).

The `received` event is fired on the `form` element. This event does not bubble. The `onreceived` attribute can be used to handle this event.

The event uses the `ReceivedEvent` interface.

If it is canceled, then the submission processing stops at this point. If it is not canceled, then its default action is to perform the rest of the submission procedure (step eight). If the `document` attribute of the event was mutated, the mutated version is what is used in the next step.

## 8. Step eight: Handle the returned data

If the response is an HTTP 204 No Content response (or equivalent for other protocols), then the document is left in place, and new metadata (if any) is applied, as per the HTTP specification [\[RFC2616\]](#).

If the response is an HTTP 205 Reset Content response (or equivalent for other protocols), then the document in the frame or window targeted by the form submission is left in place and all form elements in it are reset to their initial values (by effectively invoking their `reset()` method, which also causes `reset` events to be dispatched). (This is based on the vague requirement given in the HTTP specification. [\[RFC2616\]](#)) Other specifications may require additional processing for 205 responses, but those do not affect conformance to this specification.

***Note: This does not attempt to define how UAs are to react to 205 responses in other contexts.***

Otherwise, how the UA handles a response depends on the `replace` attribute of the element that initiated the submission.

For `replace="document"` (the default), the response body replaces the document from which the submission initiated (or, if there is a `target` attribute, the document in the appropriate frame or window).

For example, if the `action` attribute denotes an HTTP resource, the `method` attribute is "POST", the `replace` attribute is `document` and the remote server replies with a 200 OK response, then the returned document should be displayed to the user as if the user had navigated to that document by following a link to it.

For `replace="values"`, the algorithm described in the section on [seeding a form with initial values](#) must be run with the given response body used instead of the document mentioned in the `data` attribute. (Any `target` attribute is ignored.)

If the submission process fails, UAs should report this failure to the user in a useful and accessible manner.

## 5.1. Successful form controls

The controls that are **successful** are those that are included in the submission (in the form data set) when their form is submitted.

All form controls are successful except:

- Controls with no associated form.
- Controls that are inside [repetition templates](#) (those that are in their forms' `templateElements` list).
- Controls that are inside `datalist` elements.
- Controls with no name, except if they are `image` controls.
- Disabled controls.
- Checkboxes that are not checked.
- Radio buttons that are not checked.
- Submit buttons (including image buttons) that did not initiate the current submission process.
- Buttons of type `button`, `reset`, `add`, `remove`, `move-up`, or `move-down`.
- Output controls.



- Controls do not have to have a value to be successful.

The different form data set encoding types each define how to find the character encoding to use to submit the data.

Sometimes, the form submission character encoding used is not able to represent all the characters present in the form submission.

If the form data set contains characters that are outside the submission character set, the user agent should inform the user that his submission will be changed. For example, if the user entered "Dürst était très utile ici" in a text field but the author specified a character encoding where the "ü", "é", and "è" characters could not be represented, then the UA could use a dialog of the form:

[illegible]

When the value that is the cause of encoding problems is not accessible to the user (e.g. a hidden form control, one that is hidden by CSS, a read-only control, the value of an option or radio element, etc) the UA may wish to simply indicate to the user that there might be an error with the page itself.

If the submission is not canceled, the user agent must replace each character that is not in the submission character set with one or more replacement characters.

For each such missing character, UAs must either transliterate the character to a UA-defined human-recognizable representation (for example, transliterating U+263A to the three-character string ":-)" in US-ASCII, or U+2126 to the byte 0xD9 in ISO-8859-7), or, for characters where a dedicated transliteration is not known to the UA, replace the character with either U+FFFD, "?", or some other single character representing the same semantic as U+FFFD.

Note that a string containing the codepoint's value itself (for example, the six-character string "U+263A" or the seven-character string "&#9786;") is not considered to be human readable and must not be used as a transliteration. (This is to discourage servers from attempting to mechanically convert such codepoints back into Unicode characters, as there is no way to distinguish such characters from identical literal strings entered by the user.)

### 5.2.1. The `_charset_` field

The `application/x-www-form-urlencoded` and `text/plain` encoding types for the form data set look for a `hidden` form field with the name `_charset_`, and if it is present in the form data set, they include it in the submission with the submission encoding. This allows authors to determine the encoding that was used in submissions that would otherwise be ambiguous.

## 5.3. `application/x-www-form-urlencoded`

This section defines the expected behaviour for step 5, "Step five: Encode the form data set", of the submission algorithm described above, for the form content type `application/x-www-form-urlencoded`. The rest of the form submission process progresses as described above.

This is the default content type. Forms submitted with this content type must be encoded as follows:

1. The submission character encoding is selected from the form's `accept-charset` attribute. UAs must use the encoding that most completely covers the characters found in the form data set of the encodings specified. If the attribute is not specified, then the client should use either the page's character encoding, or, if that cannot encode all the characters in the form data set, UTF-8. Character encodings that are not mostly supersets of US-ASCII must not be used (this includes UTF-16 and EBCDIC) even if specified in the `accept-charset` attribute.

How a UA establishes the page's character encoding is determined by the markup language specification (for example, [HTML4 section 5.2.2 \[HTML4\]](#)). It could be explicitly specified by the page, overridden by the user, or auto-detected by the UA.

Authors must not specify an encoding other than UTF-8 or US-ASCII in the `accept-charset` attribute when the method used is `get` and the `action` indicates an HTTP resource.

2. If the form contains an input control of type `hidden` with the name `_charset_`, it is forced to appear in the form data set, with the value equal

to the name of the submission character encoding used.

3. The values of file upload controls are the names (excluding path information) of the files selected by the user, *not* their contents. (The names [may be blank](#).)
4. Control names and values are escaped. Space characters are replaced by "+" (U+002B), and other non-alphanumeric characters are encoded in the submission character encoding and each resulting byte is replaced by "%HH", a percent sign (U+0025) and two uppercase hexadecimal digits representing the value of the byte.
5. The control names/values are listed in the order they appear in the form data set. The name is separated from the value by "=" (U+003D) and name/value pairs are separated from each other by "&" (U+0026).

Note that the control index and repetition block parts of the form data set are not used.

#### 5.4. `application/x-www-form+xml`: XML submission

This section defines the expected behaviour for step 5, "Step five: Encode the form data set", of the submission algorithm described above, for the form content type `application/x-www-form+xml`. The rest of the form submission process progresses as described above.

The message entity must be a well-formed XML 1.0 document, encoded as either UTF-8 or UTF-16 (at the choice of the UA), which has a root element named `formdata`, with no prefix, defining a default namespace

`http://n.whatwg.org/formdata`.

***Note: As described by the XML specification, UAs may include an XML declaration (although doing so would be redundant). If using UTF-16, UAs must include a BOM. If using UTF-8, UAs may include a BOM, but it is not required.***

Note that the form's `accept-charset` attribute is ignored for this encoding type.

First, for each repetition block in the form data set, an element `repeat` is inserted, with an attribute `template` equal to the ID of the template, and an attribute `index` equal to the index of the repetition block. The element is empty.

***Note: Servers are generally expected to ignore `repeat` elements; they are primarily included so that form data can be round-tripped***

### *using the `data` attribute on the form element.*

Then, for each [successful](#) control that is not a file upload control, in the order that the controls are to be found in the original document, an element `field` is inserted, with an attribute `name` having the name of the form control, an attribute `index` having the control index described above in the definition of the [form data set](#), and with the element content being the current value of the form control. Form controls with multiple values result in multiple `field` elements being inserted into the output, one for each value, all with the same index.

File controls are submitted using a `file` element instead of a `field` element. The `file` element has four attributes, `name`, `index`, `filename`, and `type`. The `name` attribute contains the name of the file control. The `index` attribute contains the control index in the control's entry in the form data set. The `filename` attribute is optional and may contain the name of the file, [if it has one](#). The `type` attribute is also optional and must either contain the MIME type of the file or be omitted if the client is unaware of the correct type. The type may contain MIME parameters if appropriate. [\[RFC2046\]](#) The contents of the file are base64 encoded and then included literally as content directly inside the `file` element. As base64 data is whitespace-clean, UAs may introduce whitespace into the `file` element to ensure the submitted data has reasonable line lengths. This is, however, completely optional. (It is primarily intended to make it possible to write readable examples of submission output.)

UAs may use either CDATA blocks, entities, or both in escaping the contents of attributes and elements, as appropriate. The resulting XML must be a well-formed XML instance. The only mention of namespaces in the submission document must be the declaration of the default namespace on the root element.

Whitespace may be inserted around elements that are children of the `formdata` element in order to make the submitted data easier to scan by eye. However, this is optional. Processors should not be affected by such whitespace, or whitespace inside `file` elements, when reading the submitted data back from the XML instance. (Whitespace inside `field` elements is significant, however.)

***Note: While this section restricts the exact features of XML that a UA may use, these restrictions do not apply to the files used when [seeding a form with initial values](#).***

***Note: For forward compatibility, it is suggested that scripts skip past unexpected nodes and their descendents when processing XML files representing form submissions.***

The following example illustrates `application/x-www-form+xml` encoding. Suppose we have the following form:

```
<form action="http://example.com/cgi/handle"
      enctype="application/x-www-form+xml"
      method="post">
  <p>
    <label> What is your name? <input type="text" name="submit-name"/>
    <label> What files are you sending? <input type="file" name="file" />
    <label> When were they written? <input type="date" name="stamp"/>
    <input type="submit" value="Send"/>
  </p>
</form>
```

If the user enters "Larry" in the text input, selects the text file "file1.txt", and picks an arbitrary date, the user agent might send back the following data:

```
Content-Type: application/x-www-form+xml

<formdata xmlns="http://n.whatwg.org/formdata">
  <field name="submit-name" index="0">Larry</field>
  <file name="files" index="0" filename="file1.txt" type="text/plain"
    Y29udGVudHMgb2YgZmlsZTEudHh0
  </file>
  <field name="stamp" index="0">1979-04-13</field>
</formdata>
```

If the user selects a second (image) file "file2.png", the user agent might construct the entity as follows:

```
Content-Type: application/x-www-form+xml

<formdata xmlns="http://n.whatwg.org/formdata">
  <field name="submit-name" index="0">Larry</field>
  <file name="files" index="0" filename="file1.txt" type="text/plain"
    Y29udGVudHMgb2YgZmlsZTEudHh0
  </file>
  <file name="files" index="0" filename="file2.png" type="image/png"
    iVBORw0KGgoAAAANSUHEUgAAAAEAAAABCAMAAAAoyzS7AAAABGdBTUEAAK
    /INwWK6QAAABl0RVh0U29mdHdhcmUAQWRvYmUgSWlhZ2ZVSZWFkeXhJZTwA
    AAAGUExURQD/AAAAAG8DfkMAAAAMSURBVHjaYmAACDAAAAIAAU9tWeEAAA
    AASUVORK5CYII=
  </file>
  <field name="stamp" index="0">1979-12-27</field>
</formdata>
```

Note how the content of the plain text attached file is base64-encoded, despite being a plain text file. This preserves the integrity of the file in cases where the MIME type is incorrect. It also means that files with malformed content, for example, a file encoded as UTF-8 with stray continuation bytes, will be transmitted faithfully instead of being re-encoded by the UA.

This example illustrates this encoding for the case with two form controls with the same name. Suppose we have the following form:

```
<form enctype="application/x-www-form+xml" method="post">
  <p>
    Enter your new password twice:
    <input type="password" name="password"/>
    <input type="password" name="password"/>
    <input type="submit" value="Send"/>
  </p>
</form>
```

If the user enters "perfect" and "prefect", the user agent might send back the following data:

```
Content-Type: application/x-www-form+xml

<formdata xmlns="http://n.whatwg.org/formdata">
  <field name="password" index="0">perfect</field>
  <field name="password" index="1">prefect</field>
</formdata>
```

Recall the [example for repetition blocks](#). If it was immediately submitted, the output would be an XML file equivalent to:

```
Content-Type: application/x-www-form+xml

<formdata xmlns="http://n.whatwg.org/formdata">
  <repeat template="row" index="0"/>
  <repeat template="row" index="1"/>
  <field name="name_0" index="0">John Smith</field>
  <field name="count_0" index="0">2</field>
  <field name="name_1" index="0"></field>
  <field name="count_1" index="0">1</field>
</formdata>
```

## 5.5. text/plain

This section defines the expected behaviour for step 5, "Step five: Encode the form data set", of the submission algorithm described above, for the form content type `text/plain`. The rest of the form submission process progresses as described above.

This content type is more human readable than the others but is not unambiguously parseable. Forms submitted with this content type must be encoded as follows:

1. The submission character encoding is selected from the form's `accept-charset` attribute. UAs must use the encoding that most completely covers the characters found in the form data set of the encodings specified. If the

attribute is not specified, then the client should use either the page's character encoding, or, if that cannot encode all the characters in the form data set, UTF-8.

How a UA establishes the page's character encoding is determined by the language specification (for example, [HTML4 section 5.2.2 \[HTML4\]](#)). It could be explicitly specified by the page, overridden by the user, or auto-detected by the UA.

2. If the form contains an input control of type `hidden` with the name `_charset_`, it is forced to appear in the form data set, with the value equal to the name of the submission character encoding used.
3. The values of file upload controls are the names (excluding path information) of the files selected by the user, *not* their contents. (The names [may be blank](#).)
4. The control names/values are listed in the order they appear in the form data set. The name is separated from the value by "=" (U+003D) and name/value pairs are separated from each other by a newline character.

Note that the control index and repetition block parts of the form data set are not used.

***Note: This algorithm does not directly parallel the algorithm for `application/x-www-form-urlencoded`. This is mostly due to backwards compatibility concerns (for both cases).***

## 5.6. Submitting the encoded form data set

The exact semantics of the `method` and `enctype` attributes depend on the protocol specified by the `action` attribute, in the manner described in this section.

The attributes considered are those of the element that initiated the submission — if the user started the submission then the attributes come from the submit button or image that the user activated; if script started the submission then the attributes of the form are used. If an attribute is missing from a submit button, then the equivalent attribute on the form is used instead.

In the following example:

```
<form action="test.php" method="post">
  <input type="submit">
  <input type="submit" method="get">
</form>
```

The first submit button would submit to the `test.php` script using the HTTP POST method, and the second would submit to the same script but using the HTTP GET method.

The HTTP specification defines various methods that can be used with HTTP URIs. Four of these may be used as values of the `method` attribute: `get`, `post`, `put`, and `delete`. In this specification, these method names are applied to other protocols as well. This section defines how they should be interpreted.

If the specified `method` is not one of `get`, `post`, `put`, or `delete` then it is treated as `get` in the tables below.

If the `enctype` attribute is not specified (or is set to the empty string), and the form consists of exactly one file upload control with exactly one file selected, then in the tables below, the "File upload" rows must be used. (This is primarily to allow the PUT method to actually be useful for uploading data other than form fields.) If the form contains something other than just one file upload control with exactly one file selected, or if the attribute *is* specified but has an unrecognised value, the `enctype` attribute is treated as if it was `application/x-www-form-urlencoded`.

User agents may implement whichever URI schemes are required for their particular application. This specification does not specify a required core set of protocols that must be implemented. For those that are implemented, UAs must use the algorithms given in the following sections when submitting data using those protocols.

What user agents should do when the designated resource is fetched depends on the value of the `replace` attribute. This is described in [step eight of the algorithm](#).

The value of the `enctype` attribute must be dispatched using a `case-insensitive` literal comparison. The attribute must therefore not have any MIME parameters. (For example, the value `multipart/form-data; charset=utf-8` must not be treated as matching `multipart/form-data`.)

**Note:** In the tables below, cells that specify that their case should be handled as another represent attribute combinations that are otherwise meaningless. In such cases, the encoded data set is created from the fallback encoding type, not the specified encoding type.

#### 5.6.1. For `http:` actions

HTTP is described by [\[RFC2616\]](#).



	get	post	put	delete
application/x-www-form-urlencoded	Use the encoded data set as the query value for a URI formed from the <code>action</code> attribute and fetch it via HTTP GET. If the URI given by the <code>action</code> attribute already contains a query value (i.e. the URI already contains a <code>?</code> character) then that query value is removed first.	Use the encoded data set as the entity body, with the <code>Content-Type</code> set appropriately, and submit it using the specified method.		Ignore the form data set and <code>access</code> <code>action</code> with the specified method.
multipart/form-data	Handle as if <code>enctype</code> was <code>application/x-www-form-urlencoded</code> .			
application/x-www-form+xml				
text/plain				
File upload		Use the file content as the entity body, with the <code>Content-Type</code> set to its MIME type, and submit it using the specified method.		

For a GET request, if the `action` attribute of a form resolved to `http://example.com/?q=test` and the encoded data set was `foo=bar` then the resulting URI would be `http://example.com/?foo=bar`.

### 5.6.2. For `ftp:` actions

The `ftp:` URI scheme is described by [\[RFC1738\]](#) and FTP itself is described by [\[RFC959\]](#).

Using the FTP protocol for form submission is of dubious value and is discouraged.

	get	post	put	delete
<b>application/x-www-form-urlencoded</b>	Ignore the form data set and	Handle as if <code>method</code>	Use the encoded data set as the content of a file	Ignore the form data set and delete the file

<b>multipart/form-data</b>	retrieve the file specified by	was put.	and upload it to the location specified by <code>action (STOR)</code> .	specified by <code>action (DELE)</code> . The response body has no content
<b>application/x-www-form+xml</b>	<code>action (RETR)</code> .		The response body has no content (equivalent to an HTTP 204 No Content response.)	(equivalent to an HTTP 204 No Content response.)
<b>text/plain</b>				
<b>File upload</b>			Upload the selected file to the location specified by the <code>action</code> attribute ( <code>STOR</code> ). The response body has no content (equivalent to an HTTP 204 No Content response.)	

Using these semantics, a poor man's FTP upload form could be written like so:

```
<form method="put" xml:base="ftp://ftp.example.com/incoming/">
  <p>
    <label>
      Path:
      <input type="text" pattern="^[^/]*"
        onchange="if (validity == 0) form.action = encodeURI(
    </label>
    <input type="file" name="file"/>
    <input type="submit" value="Upload file"/>
  </p>
</form>
```

### 5.6.3. For `data:` actions

The `data:` URI scheme is described by [\[RFC2397\]](#).

	<b>get</b>	<b>post</b>	<b>put</b>	<b>delete</b>
<b>application/x-www-form-urlencoded</b>	Ignore the form data set and access the <code>action</code> URI.	If the <code>action</code> contains the string "%%%%", URI escape all non-alphanumeric	Ignore <code>action</code> ; form a new base64-encoded	Handle as if <code>method</code> was <code>post</code> .

<b>multipart/form-data</b>	Handle as if <code>enctype</code> was <code>application/x-www-form-urlencoded</code> .	characters in the encoded form data set, URI escape the result again, and substitute the result for the first occurrence of the string "%%%%" in the <code>action</code> .	data: URI from the entity body, using the appropriate MIME type, and access it.
<b>application/x-www-form+xml</b>			
<b>text/plain</b>		Otherwise, just URI escape the encoded form data set once and substitute it for the first occurrence of the string "%%" in the <code>action</code> (if any). Then, access the resulting URI.	
<b>File upload</b>			Ignore <code>action</code> ; form a new base64-encoded data: URI from the selected file's contents, using the file's MIME type, and access it.

If the following form is submitted by pressing the button:

```
<form action="data:,Data%20was:%20'%%%' " method="post">
  <p><input type="submit" name="x" value="It's a test">
</form>
```

...it would result in the following URI being used:

```
data:,Data%20was:%20'x%253DIt%2527s%252Ba%252Btest'
```

...but if the `action` was changed to just `data:,Data%20was:%20'%%'` (that is, just "%%" instead of "%%%%" ) then the URI used would be:

```
data:,Data%20was:%20'x%3DIt's%2Ba%2Btest'
```

Note that "%%" is invalid in a URI, so authors should exercise caution when using the `post` method with `data:` URIs.

**Usage:** Submitting to `data:` URIs is mainly useful when debugging (to find out exactly what is being submitted).

#### 5.6.4. For file: actions

The file: URI scheme is described by [\[RFC1738\]](#).

For security reasons, untrusted content should never be allowed to submit or fetch files specified by file URIs.

The semantics described in this subsection are recommended, but UAs may implement alternative semantics if desired, as consistent behaviour for submission to file: URIs is not required for interoperability on the World Wide Web.

	get	post	put	delete
<b>application/x-www-form-urlencoded</b>	Ignore the form data set and retrieve the file specified by <code>action</code> .	If the specified file is executable, launch the specified file in an environment that complies with the CGI Specification [RFC3875], using the encoded data set as the standard input and the resulting standard output as an HTTP response entity (see details below). If the specified file is not executable, handle as if <code>method</code> was <code>get</code> .	Use the encoded data set as the content of a file and store it in the location specified by <code>action</code> . The response body has no content (equivalent to an HTTP 204 No Content response.)	Ignore the form data set and delete the file specified by <code>action</code> . The response body has no content (equivalent to an HTTP 204 No Content response.)
<b>multipart/form-data</b>				
<b>application/x-www-form+xml</b>				
<b>text/plain</b>				
<b>File upload</b>		Same as for other types except the encoded form data set is the contents of the specified file.	Store the selected file at the location specified by the <code>action</code> URI.	

The standard output mentioned above for "post" requests should be handled as an HTTP/1.1 response entity, with a header section and an entity body section. If the header section contains a header with the name "Status", its value is appended to the string `HTTP/1.1` and treated as the response status line. Otherwise, the response status line is assumed to be `HTTP/1.1 200 OK`. An explicit response status line if present, is treated as a header, not a status line.

|| If the standard output of a program executed in this context is:

```
Content-Type: text/plain
Status: 301 Permanent Redirect
Location: http://www.example.org/
```

See <http://www.example.org/>

...then the UA should handle this exactly as if it had received the following response over an HTTP connection:

```
HTTP/1.1 301 Permanent Redirect
Content-Type: text/plain
Location: http://www.example.org/
```

See <http://www.example.org/>

If a program in this context returned the following standard output:

```
'submit.pl' is not recognized as an internal or external command,
operable program or batch file.
```

...then the UA should handle this as the following HTTP response entity:

```
HTTP/1.1 200 OK
'submit.pl' is not recognized as an internal or external command,
operable program or batch file.
```

...which is equivalent to an empty document (since there is no response body).

**Usage:** Submitting to file: URIs is mainly useful when developing. The definition for POST to file: enables authors to develop CGI scripts without a Web server. It also allows content that uses simple CGI scripts to be shipped on media without Web servers (such as CDROMs).

Submitting to file: with PUT and DELETE is useful primarily with the XML submission format and preseeding, as it allows data to be stored locally between uses of the application.

#### 5.6.5. For `mailto:` actions

The `mailto:` URI scheme is described by [\[RFC2368\]](#).

UAs should not send e-mails without the explicit consent of the user.

All submissions made using `mailto:` result in the equivalent of an HTTP 204 No Content response. Thus the `replace` attribute is effectively ignored when `action` is a `mailto` URI.

	get	post	put	delete
--	-----	------	-----	--------

<b>application/x-www-form-urlencoded</b>	Use the encoded data set as the <code>headers</code> part (see <a href="#">RFC2368</a> ) of a <code>mailto:</code> URI formed from the <code>action</code> attribute and process that URI.	Use the encoded data set as the default message body, with the <code>Content-Type</code> set appropriately, for a message based on the specified <code>action</code> attribute.	Handle as if method was <code>post</code> .
<b>multipart/form-data</b>	Handle as if <code>enctype</code> was <code>application/x-www-form-urlencoded</code> .		
<b>application/x-www-form+xml</b>			
<b>text/plain</b>			
<b>File upload</b>		Attach the selected file to a message based on the specified <code>action</code> attribute.	

5.6.6. For `smsto:` and `sms:` actions

The `smsto:` and `sms:` URI schemes are not yet specified.

UAs should not send SMSes without the explicit consent of the user.

All submissions made using the `smsto:` and `sms:` URI schemes result in the equivalent of an HTTP 204 No Content response. Thus the `replace` attribute is effectively ignored when `enctype` is an SMS URI.

	get	post	put	delete
<b>application/x-www-form-urlencoded</b>	<i>Behaviour is undefined, pending the release of an <code>smsto:</code> or <code>sms:</code> specification.</i>	Use the encoded data set as the default message body for a message based on the specified <code>action</code> attribute.	Handle as if method was <code>post</code> .	
<b>multipart/form-data</b>	Handle as if <code>enctype</code> was <code>application/x-www-form-urlencoded</code> .	Handle as if <code>enctype</code> was <code>application/x-www-form-urlencoded</code> .		
<b>application/x-www-form+xml</b>				
<b>text/plain</b>		Use the encoded data set as the default message body for a		

	message based on the specified <code>action</code> attribute.
File upload	Handle as if <code>enctype</code> was <code>application/x-www-form-urlencoded</code> .

5.6.7. For `javascript:` actions

The `javascript:` URI scheme is [described](#) by [\[CSJSR\]](#). ECMAScript is defined in [\[ECMA262\]](#).

If the response body of a submission to a `javascript:` action is the ECMAScript `void` type, then it is treated as if it was an HTTP 204 No Content response.

	get	post	put	delete
application/x-www-form-urlencoded	Ignore the form data set and access the URI given by the <code>action</code> attribute in the current context. The response body is the return value of the script.	Encode the form data set by putting each name/value pair into a newly created object using the names as attributes of that object and the values as the values of those attributes. Execute the URI given by the <code>action</code> attribute in the context of the document after having added the aforementioned object to the start of the scope chain. Duplicate names should cause the property to become an array, with each value represented in the array. The response body is the return value of the script.	Handle as if method was <code>post</code> .	
multipart/form-data				
application/x-www-form+xml				
text/plain				
File upload				

**Usage:** Submitting to `javascript:` URIs is useful for applications that are entirely client-side, but still form-driven.

6. Fetching data from external resources

There are two scenarios where authors may wish data to be fetched from an external file to fill forms. In the first, a `select`'s options are replaced by options

from an external file. In the second, a form's values are prefilled from an external data source.

In both cases, the prefilling may either be full, in which case the previous contents are removed first, or incremental, in which case the fetched data is in addition to the data already in the form.

Implementations may limit which hosts, ports, and schemes can be accessed using these methods. For example, HTTP-based content should not be able to preseed a form based on content from the local file system. Similarly, cross-domain scripting restrictions are fully expected to apply.

## 6.1. Filling `select` elements

If a `select` element or a `datalist` element being parsed has a `data` attribute, then as soon as the element and all its children have been parsed and added to the document, the prefilling process described here should start.

If a `select` element or a `datalist` element has a `data` attribute, it must be a URI or IRI that points to a well-formed XML file whose root element is a `select` element in the `http://www.w3.org/1999/xhtml` namespace. The MIME type must be an XML MIME type [\[RFC3023\]](#), preferably `application/xml`. It should not be `application/xhtml+xml` since the root element is not `html`.

UAs must process this file if it has an XML MIME type [\[RFC3023\]](#), if it is a well-formed XML file, and if the root element is the right root element in the right namespace. If any of these conditions are not met, UAs must act as if the attribute was not specified, although they may report the error to the user. UAs are expected to correctly handle namespaces, so the file may use prefixes, etc.

If the UA processes the file, it must use the following algorithm to fill the form.

1. Unless the root element of the file has a `type` attribute with the exact literal string `incremental`, the children of the `select` or `datalist` element in the original document must all be removed from the document.
2. The entire contents of the `select` element in the referenced document are imported into the original document and appended as children of the `select` or `datalist` element. (Even if importing into a `text/html` document, the newly imported nodes will still be namespaced.)
3. All nodes outside the `select` (such as stylesheet processing instructions, whitespace text nodes, and `DOCTYPEs`) are ignored, as are attributes (other than `type`) on the `select` element.



The prefilling processes for `select` and `datalist` elements started during document load must all be completed before the document's `load` event can fire.

If a `select` or `datalist` element has its `data` attribute manipulated via the DOM, then the prefilling process must start as soon as any executing scripts have run to completion. If the attribute is set multiple times during one execution of a script, only the last request must take effect. If the process is started while an outstanding prefilling request is still being attended to, the requests must all be serviced in the order they were started.

The following script has only one possible valid outcome:

```
var select = document.createElementNS('http://www.w3.org/1999/xhtml')
select.data = 'data:application/xml,%3Cselect%20xmlns%3D%22http%3A%2F%2Fwww.w3.org%2F1999%2Fxml%2F%3E%3Coption%20value%3D%22a%22%3Ea%3C%2Foption%3E%3Coption%20value%3D%22b%22%3Eb%3C%2Foption%3E%3Coption%20value%3D%22c%22%3Ec%3C%2Foption%3E%3C%2Fselect%3E'
// at this point, select.length == 0 is guaranteed
var option = document.createElementNS('http://www.w3.org/1999/xhtml')
option.appendChild(document.createTextNode('a'))
select.appendChild(option)
// at this point, select.length == 1 is guaranteed
document.documentElement.appendChild(select);
```

...namely, the insertion at the end of the document of a select widget which, in due course, will have three options, namely "a", "b" and "c".

## 6.2. Seeding a form with initial values

Before `load` events are fired, but after the entire document has been parsed and after `select` elements have been [filled from external data sources](#) (if necessary), forms with `data` attributes are prefilled.

***Note: In particular, UAs should not specifically wait for images and stylesheets to be loaded before preseeding forms.***

If a `form` has a `data` attribute, it must be a URI or IRI that points to a well-formed XML file whose root element is a `formdata` element in the `http://n.whatwg.org/formdata` namespace. The MIME type must be an XML MIME type [\[RFC3023\]](#), preferably `application/xml`.

UAs must process this file if these conditions are met. If any of these conditions are not met, UAs must act as if the attribute was not specified, although they may report the error to the user. UAs are expected to correctly handle namespaces, so the file may use prefixes, etc.

If the UA processes the file, it must use the following algorithm to fill the form.

1. Unless the root element has a `type` attribute with the exact literal string `incremental`, the form must be reset to its initial values as specified in the markup.
2. Child text nodes, CDATA blocks, comments, and PIs of the root element of the specified file must be ignored.
3. `repeat` elements in the `http://n.whatwg.org/formdata` namespace that are children of the root element, have a non-empty `template` attribute and an `index` attribute that contains only one or more digits in the range 0-9 with an optional leading minus sign (U+002D, "-"), have no other non-namespaced attributes (ignoring `xmlns` attributes), and have no content, must be processed as follows:

The `template` attribute should contain the ID of an element in the document. If the `template` attribute specifies an element that is not a [repetition template](#), then the `repeat` element is ignored.

If the `template` attribute specifies a [repetition template](#) and that template already has a [repetition block](#) with the index specified by the `index` attribute, then the element is ignored.

Otherwise, the specified template's `addRepetitionBlockByIndex()` method is called, with a null first argument and the index specified by the `repeat` element's `index` attribute as the second.

4. `field` elements in the `http://n.whatwg.org/formdata` namespace that are children of the root element, have a non-empty `name` attribute, either an `index` attribute that contains only one or more digits in the range 0-9 or no `index` attribute at all, have no other non-namespaced attributes (ignoring `xmlns` attributes), and have either nothing or only text and CDATA nodes as children, must be used to initialize fields, as follows.

First, the form control that the field references must be identified. This is done by walking the list of form controls associated with the form until one is found that has a name exactly equal to the name given in the `field` element's `name` attribute, skipping as many such matches as is specified in the `index` attribute, or, *if the `index` attribute was omitted*, skipping over any `type="radio"` and `type="checkbox"` controls that have the exact name given but have a value that is not exactly the same as the contents of the `field` element.

For `image` controls, instead of using the name given by the `name` attribute, the field's name is checked against two names, the first being the value of the `name` attribute with the string `.x` appended to it, and the second being the same but with `.y` appended instead. If an `image` control's name is the

empty string (e.g. if its `name` attribute is omitted) then the names `x` and `y` must be used instead. Thus `image` controls are handled as if they were two controls.

If the identified form control is a file upload control, a push button control, or an image control, then the `field` element is now skipped.

Next, if the identified form control is not a multiple-valued control (a multiple-valued control is one that can generate more than one value on submission, such as a `<select multiple="multiple">`), or if it is a multiple-valued control but it is the first time the control has been identified by a `field` element in this data file that was not ignored, then it is set to the given value (the contents of the `field` element), removing any previous values (even if these values were the result of processing previous `field` elements in the same data file). Otherwise, this is a subsequent value for a multiple-valued control, and the given value (the contents of the `field` element) should be *added* to the list of values that the element has selected.

If the element cannot be given the value specified, the `field` element is ignored and the control's value is left unchanged. For example, if a checkbox has its value attribute set to `green` and the `field` element specifies that its value should be set to `blue`, it won't be changed from its current value. (The only values that would have an effect in this example are `""`, which would uncheck the checkbox, and `"green"`, which would check the checkbox.) Another example would be a `datetime` control where the specified value is outside the range allowed by the `min` and `max` attributes. The format must match the allowed formats for that type for the value to be set.

If the element is a multiple-valued control and the control already has the given value selected, but it can be given the value again, then that occurs. For example, in the following case:

```
<select name="select" multiple="multiple">
  <option>test</option>
  <option>test</option>
  <option>test</option>
</select>
```

...if the data file contained two instances of:

```
<field name="select" index="0">test</field>
```

...then the first two `option` elements would end up selected, and the last would not. This would be the case irrespective of which `option` elements had their `selected` attribute set in the markup.

**Note:** The *option* elements are never directly matched by *field* elements; it is the *select* element in this case that is matched (twice). This is why the two *field* elements select subsequent values in the control.

If the element is a multiple-valued control and the control already has the given value selected and it *cannot* be given the value again, then the field is ignored.

If the element is a radio button, then setting it to its value resets all the other radio buttons in the group to their unchecked state.

5. A formchange event is then fired on all the form controls of the form.

All other elements in the file must be ignored. The algorithm must be processed in the order given above, meaning repeat elements are handled before the *field* elements, regardless of the order in which the elements are given. (Note that this implies that this process cannot be performed incrementally.)

**Note:** Note that file upload controls cannot be repopulated. However, an output control can be populated. This can be used, for example, for localizing a form by including the structure in one file and the strings in another. (The semantics of this practice are somewhat dubious, however. It is only mentioned because XForms advocates claim this as a feature.)

The form prefilling processes started during document load must all be completed before the document's `load` event can fire.

Setting the `data` attribute dynamically does not cause the UA to refill the form. The semantics of the `data` attribute are only relevant during initial document load, with the form filling kicked off just as the document has finished being parsed. Thus, as far as scripted changes to the DOM are concerned, UAs must only prefill forms with `data` attributes that are added to the document by script before the UA has finished parsing the document. The DOM [can be used to refill a form](#) after the document has finishing loading using other methods, however.

## 7. Extensions to the HTML Level 2 DOM interfaces

Unless otherwise specified, these interfaces have the same semantics as

defined in [\[DOM2HTML\]](#).

The interfaces found within this section are mandatory for UAs that implement this specification and support scripting. A DOM application may use the `hasFeature(feature, version)` method of the `DOMImplementation` interface with parameter values "WebForms" and "2.0" (respectively) to determine whether or not this module is supported by the implementation. User agents may return true for such a call (full compliance is not required to return true). Please refer to additional information about conformance in the DOM Level 3 Core specification. [\[DOM3CORE\]](#)

```
interface HTMLFormElement : HTMLElement {
    readonly attribute HTMLCollection elements;
    readonly attribute long length;
    attribute DOMString name;
    attribute DOMString acceptCharset;
    attribute DOMString action;
    attribute DOMString enctype;
    attribute DOMString method;
    attribute DOMString target;

    void submit();
    void reset();

    // new in this specification:
    attribute DOMString accept;
    attribute DOMString replace;
    attribute DOMString data;
    readonly attribute HTMLCollection templateElements;
    bool checkValidity();
    void resetFromData(in Document data);

    void dispatchFormInput();
    void dispatchFormChange();
};

interface HTMLSelectElement : HTMLElement {
    readonly attribute DOMString type;
    attribute long selectedIndex;
    attribute DOMString value;
    attribute unsigned long length;
    // raises(DOMException) on sett

    readonly attribute HTMLFormElement form;
    readonly attribute HTMLOptionsCollection options;
    attribute boolean disabled;
    attribute boolean multiple;
    attribute DOMString name;
    attribute long size;
    attribute long tabIndex;

    void add(in HTMLElement element,
             in HTMLElement before)
```

```

        raises(DOMException);

void          remove(in long index);
void          blur();
void          focus();

// new in this specification:
readonly attribute NodeList      forms;
        attribute DOMString      accessKey;
        attribute boolean        autofocus;
        attribute DOMString      data;
readonly attribute HTMLCollection selectedOptions;
readonly attribute HTMLCollection labels;

readonly attribute boolean        willValidate;
readonly attribute ValidityState validity;
readonly attribute DOMString      validationMessage;
bool          checkValidity();
void          setCustomValidity(in DOMString error);
void          dispatchChange();
void          dispatchFormChange();
};

interface HTMLDataListElement : HTMLElement {
    readonly attribute HTMLOptionsCollection options;
        attribute DOMString      data;
};

interface HTMLOptGroupElement : HTMLElement {
        attribute boolean        disabled;
        attribute DOMString      label;
};

interface HTMLOptionElement : HTMLElement {
    readonly attribute HTMLFormElement form;
        attribute boolean        defaultSelected;
    readonly attribute DOMString      text;
    readonly attribute long           index;
        attribute boolean        disabled;
        attribute DOMString      label;
        attribute boolean        selected;
        attribute DOMString      value;

    // new in this specification:
    readonly attribute NodeList      forms;
};

interface HTMLInputElement : HTMLElement {
        attribute DOMString      defaultValue;
        attribute boolean        defaultChecked;
    readonly attribute HTMLFormElement form;
        attribute DOMString      accept;
        attribute DOMString      accessKey;
        attribute DOMString      align;
        attribute DOMString      alt;
};
```

```

        attribute boolean          checked;
        attribute boolean          disabled;
        attribute long             maxLength;
        attribute DOMString        name;
        attribute boolean          readOnly;
        attribute unsigned long    size;
        attribute DOMString        src;
        attribute long             tabIndex;
        attribute DOMString        type;
        attribute DOMString        useMap;
        attribute DOMString        value;

void      blur();
void      focus();
void      select();
void      click();

// new in this specification:
readonly attribute NodeList       forms;
        attribute DOMString       min;
        attribute DOMString       max;
        attribute DOMString       step;
        attribute DOMString       pattern;
        attribute boolean         required;
        attribute boolean         autocomplete;
        attribute boolean         autofocus;
        attribute DOMString       inputmode;
        attribute DOMString       action;
        attribute DOMString       enctype;
        attribute DOMString       method;
        attribute DOMString       target;
        attribute DOMString       replace;
readonly attribute HTMLElement    list;
readonly attribute HTMLOptionElement selectedOption;
readonly attribute RepetitionElement htmlTemplate;
readonly attribute HTMLCollection labels;

        attribute DOMTimeStamp    valueAsDate;
        attribute float           valueAsNumber;

void stepUp(in int n);
void stepDown(in int n);

readonly attribute boolean         willValidate;
readonly attribute ValidityState   validity;
readonly attribute DOMString       validationMessage;
bool      checkValidity();
void      setCustomValidity(in DOMString error);
void      dispatchChange();
void      dispatchFormChange();
};

interface HTMLTextAreaElement : HTMLElement {
        attribute DOMString       defaultValue;
        readonly attribute HTMLFormElement form;

```

```

        attribute DOMString      accessKey;
        attribute long           cols;
        attribute boolean        disabled;
        attribute DOMString      name;
        attribute boolean        readOnly;
        attribute long           rows;
        attribute long           tabIndex;
    readonly attribute DOMString  type;
        attribute DOMString      value;

    void      blur();
    void      focus();
    void      select();

    // new in this specification:
    readonly attribute NodeList    forms;
        attribute DOMString      wrap;
        attribute DOMString      pattern;
        attribute boolean        required;
        attribute boolean        autofocus;
        attribute DOMString      inputmode;
        attribute long           maxLength;
        attribute DOMString      accept;
    readonly attribute HTMLCollection labels;

    readonly attribute boolean    willValidate;
    readonly attribute ValidityState validity;
    readonly attribute DOMString  validationMessage;
    bool      checkValidity();
    void      setCustomValidity(in DOMString error);
    void      dispatchChange();
    void      dispatchFormChange();
};

interface HTMLButtonElement : HTMLElement {
    readonly attribute HTMLFormElement form;
        attribute DOMString      accessKey;
        attribute boolean        disabled;
        attribute DOMString      name;
        attribute long           tabIndex;
        attribute DOMString      value;

    // modified in this specification
        attribute DOMString      type;

    // new in this specification:
    readonly attribute NodeList    forms;
        attribute DOMString      action;
        attribute DOMString      enctype;
        attribute DOMString      method;
        attribute DOMString      target;
        attribute DOMString      replace;
        attribute boolean        autofocus;
    readonly attribute HTMLCollection labels;
    readonly attribute RepetitionElement htmlTemplate;

```



```

void                blur();
void                focus();

readonly attribute boolean                willValidate; // always false
readonly attribute ValidityState        validity; // all members always se
readonly attribute DOMString              validationMessage; // always the e
bool                checkValidity(); // returns true
void                setCustomValidity(in DOMString error); // raises N
void                dispatchChange();
void                dispatchFormChange();
};

interface HTMLLabelElement : HTMLElement {
    readonly attribute HTMLFormElement form;
        attribute DOMString            accessKey;
        attribute DOMString            htmlFor;

    // new in this specification:
    readonly attribute NodeList          forms;
    readonly attribute Element           control;
};

interface HTMLFieldSetElement : HTMLElement {
    readonly attribute HTMLFormElement form;

    // new in this specification
    readonly attribute NodeList          forms;
    readonly attribute HTMLCollection   elements;
        attribute boolean                disabled;
};

interface HTMLLegendElement : HTMLElement {
    readonly attribute HTMLFormElement form;
        attribute DOMString            accessKey;
        attribute DOMString            align;

    // new in this specification:
    readonly attribute NodeList          forms;

    readonly attribute boolean          willValidate; // always false
    readonly attribute ValidityState    validity; // all members always se
    readonly attribute DOMString        validationMessage; // always the e
    bool                checkValidity(); // returns true
    void                setCustomValidity(in DOMString error); // raises N
};

// new in this specification
interface HTMLOutputElement : HTMLElement {
        attribute DOMString            defaultValue;
    readonly attribute HTMLFormElement form;
    readonly attribute NodeList          forms;
        attribute DOMString            name;
        attribute DOMString            value;
};

```

```

    readonly attribute boolean          willValidate; // always false
    readonly attribute ValidityState  validity; // all members always se
    readonly attribute DOMString        validationMessage; // always the e
    bool                                checkValidity(); // returns true
    void                                setCustomValidity(in DOMString error); // raises N
};

// new in this specification
interface RepetitionElement {
    const          unsigned short  REPETITION_NONE = 0;
    const          unsigned short  REPETITION_TEMPLATE = 1;
    const          unsigned short  REPETITION_BLOCK = 2;

    attribute unsigned short  repetitionType;
    attribute long            repetitionIndex;
    readonly attribute Element repetitionTemplate;
    readonly attribute HTMLCollection repetitionBlocks;
    attribute unsigned long    repeatStart;
    attribute unsigned long    repeatMin;
    attribute unsigned long    repeatMax;

    Element          addRepetitionBlock(in Node refNode);
    Element          addRepetitionBlockByIndex(in Node refNode, in long
    void              moveRepetitionBlock(in long distance);
    void              removeRepetitionBlock();
};

// new in this specification
interface ValidityState {
    readonly attribute boolean  typeMismatch;
    readonly attribute boolean  stepMismatch;
    readonly attribute boolean  rangeUnderflow;
    readonly attribute boolean  rangeOverflow;
    readonly attribute boolean  tooLong;
    readonly attribute boolean  patternMismatch;
    readonly attribute boolean  valueMissing;
    readonly attribute boolean  customError;
    readonly attribute boolean  valid; // !(typeMismatch || rangeU
};

```

## 7.1. Additions specific to the HTMLFormElement interface

The new **accept** attribute reflects the `form` element's accept attribute and its addition here merely addresses an oversight in DOM2.

The **elements** array must contain all the `input`, `output`, `select`, `textarea`, `button`, and `fieldset` elements that are [associated with the form](#) except those that have a repetition template as an ancestor.

There is one exception. The **elements** array is defined to not include image controls (`input` elements of type `image`). This is for backwards compatibility with

DOM Level 0. This excludes image buttons from several features of this specification, such as `onformchange` processing and validation.

The `templateElements` attribute contains the list of form controls associated with this form that form part of repetition templates. It is defined in more detail in the section on the [repetition model](#). (Image controls *are* part of this array, when appropriate.)

The controls in the `elements` and `templateElements` lists must be in document order.

The `form.checkValidity()` method shall make a list of all the elements in the form's `elements` list whose interfaces have a `checkValidity()` method defined and a `willValidate` attribute defined, and whose `willValidate` attribute has the true value, then shall invoke the `checkValidity()` method on all the elements in that list. It shall return the logical-and of all the return values (i.e. it returns false if any of the form controls have `willValidate` set to true but are invalid). See the section on [form validation](#) for details regarding the resulting events.

The `reset()` method resets the form, then fires a `formchange` event on all the form controls of the form.

The `resetFromData()` method takes one argument, a `Document` to use for resetting the form. If this argument is null, the method does nothing. Otherwise, the algorithm described in the section on [seeding a form with initial values](#) must be run with the given document instead of the document mentioned in the `data` attribute.

The `dispatchFormChange()` and `dispatchFormInput()` methods cause `formchange` and `forminput` events (respectively) to be fired to all the elements in the `elements` array, much like what happens for the default action of `change` and `input` events.

In the ECMAScript binding, objects that implement the `HTMLFormElement` interface must reflect their `elements` according to the following semantics:

If a name is used by more than one control, the form object has a property of that name that references a `NodeList` interface that lists the controls of that name.

If a name is used by exactly one control, the form object has a property of that name that references that control.

## 7.2. Additions specific to the `HTMLSelectElement` interface

The `selectedOptions` attribute provides a readonly list of the descendent `HTMLOptionElement` nodes that currently have their `selected` attribute set to a true value (a subset of the controls listed in the `options` attribute). The list is returned live, so changing the options selected (by the user or by script) will change the list. The order of the list should be consistent with the order of the `options` list.

The `size` DOM attribute must reflect the content attribute of the same name. When the content attribute is absent the DOM attribute must return 0.

Setting the `value` or `selectedIndex` DOM attributes to values that do not correspond to any of the actual values of the `select` element must unselect any selected options. For single-select controls, this must then [cause the first non-disabled option to be selected](#).

### 7.3. The `HTMLDataListElement` interface

The `options` element returns the same as `getElementsByTagName()` would return if called on the same element, with `option` as the argument (or, in XHTML, if the namespaced version was called with the same tag name but with the XHTML namespace).

### 7.4. Changes to the `HTMLOptionElement` interface

The `index` DOM attribute is redefined to be the index of the element in the `options` list of its nearest ancestor implementing the `HTMLSelectElement` or `HTMLDataListElement` interface. If there is no such ancestor, or if the element is not in that list, the `index` attribute must return -1.

***Note: In addition to adding `datalist` to the definition, this also corrects an ambiguity in DOM2 HTML's definition of the attribute: it was undefined if the parent was not a `select` element.***

### 7.5. Additions specific to the `HTMLFieldsetElement` interface

The new `elements` attribute lists the form controls under the `fieldset`, regardless of which form they belong to. The array must contain all the `input`, `output`, `select`, `textarea` and `button` controls that have the `fieldset` element as an ancestor, in document order.

### 7.6. The `HTMLOutputElement` interface

This interface is added for the new `output` element. Its attributes work analogously to those on other controls. The semantics of the `value` and `defaultValue` DOM attributes are described in the section describing the `output` element.

## 7.7. Validation APIs

The `willValidate` attribute of a form control element must return true if all of the following conditions are met:

- The control is [associated](#) with a form (or several forms).
- The control does not have a [repetition template](#) as an ancestor.
- The control does not have a `datalist` element as an ancestor.
- The control has a name.
- The control is not [disabled](#).
- The control is not a button of type `button`, `reset`, `add`, `remove`, `move-up`, or `move-down`.
- The control is not an `output` element.

It must return false if any of these conditions are not met.

The `validity` attribute must return an object implementing the `validityState` interface that represents the state of that form control. Note that this object is live; the same object must be returned every time the attribute is retrieved. The interface has several flags, each of which must be set on the object when its respective condition (as defined below) is matched.

### `typeMismatch`

The data entered does not match the type of the control. For example, if the UA allows uninterpreted arbitrary text entry for `month` fields, and the user has entered `SEP02`, then this flag would be set. This code is also used when the selected file in a file upload control does not have an appropriate MIME type. If the control is empty, this flag must not be set.

### `rangeUnderflow`

The numeric, date, or time value of a field with a `min` attribute is lower than the minimum, or a file upload control has fewer files selected than the minimum. If the control is empty or if the `typeMismatch` flag is set, this flag must not be set.

**rangeOverflow**

The numeric, date, or time value of a field with a max attribute is higher than the maximum, or a file upload control has more files selected than the maximum. If the control is empty or if the typeMismatch flag is set, this flag must not be set.

**stepMismatch**

The value is not one of the values allowed by the step attribute, and the UA will not be rounding the value for submission. Empty values and values that caused the typeMismatch flag to be set must not cause this flag to be set.

**tooLong**

The value of a field with a maxlength attribute is longer than the attribute allows.

**patternMismatch**

The value of the field with a pattern attribute doesn't match the pattern. If the control is empty, this flag must not be set.

**valueMissing**

The field has the required attribute set but has [no value selected](#).

**customError**

The field was marked invalid from script. See the definition of the [setCustomValidity\(\)](#) [method](#).

**valid**

Set if all the other flags are not set, and not set if any of the other flags are set.

When the definitions above refer to elements that have an attribute set on them, they do not refer to elements on which that attribute is defined not to apply. For example, the valueMissing flag cannot be set on an `<input type="button">` element, even if that element has the required attribute set, since required doesn't apply to buttons.

For example, the following control:

```
<input type="text" name="test" pattern="[a-z]+" maxlength="3" value="
```

...would be flagged with both tooLong and patternMismatch when initially inserted into the document.

Assistive technologies should use the willValidate and validity DOM attributes to determine which controls are in need of user attention when submission fails due to a validity problem.

The `checkValidity()` method, present on several of the form control interfaces, causes an `invalid` event to be fired on that control, unless the `validity.valid` flag of the control is set. It returns true if the `validity.valid` flag of the control is true, otherwise it returns false. **Recall that this is automatically done during form submission.**

The `setCustomValidity()` method sets and resets the `customError` flag on the `validity` attribute. If the method's argument is `null` or the empty string, then the flag is reset (indicating that the control is valid), otherwise the given value is recorded and the flag is set (the control is not valid). Even form controls that are empty and not required can be marked invalid like this, and would abort form submission if so marked. Setting this flag is persistent, in that the flag remains set (and the string that was passed is remembered) until specifically unset using the same method. For instance, resetting the form, changing the control value, or moving the element around the document do not affect the value of this flag. The string passed to this method should be used by the UA when the UA needs to report the control as having a custom error to the user.

The `validationMessage` attribute shall be the empty string if the control is valid, and shall contain a suitably localised message that the UA would be giving the user if the form was submitted, if the control is invalid. If the only error with the form control is a custom error, then the string returned should be, or contain, the string that was passed to the `setCustomValidity()` method.

On the `HTMLInputElement`, `HTMLButtonElement`, and `HTMLFieldsetElement` interfaces, the `willValidate` attribute must return false, the `validity` attribute must return an object with all its attributes set to false except the `valid` flag which must be true, the `validationMessage` attribute must return the empty string, the `checkValidity()` method must return true, and the `setCustomValidity()` method must raise a `NOT_SUPPORTED_ERR` DOM exception.

The `HTMLInputElement`, `HTMLButtonElement`, and `HTMLFieldsetElement` interfaces implement these attributes and methods so that when iterating over the form's `elements` array, authors do not have to ensure that each form element is not an `output` element or a `button` before using them.

## 7.8. New DOM attributes for new content attributes

The new `data`, `pattern`, `required`, `autocomplete`, `autofocus`, `inputmode`, `min`, `max`, `step`, `wrap`, `disabled`, `accept`, `action`, `enctype`, `method`, `replace`, and `target` DOM attributes must simply reflect the current value of their relevant content attribute. If the content attribute is not specified or has an invalid value, then the DOM attribute reflects the default value.

The `disabled` attribute of form controls is not affected by the `disabled` attributes of ancestors. (Thus the `disabled` attribute of a form control can be `false` even though the control is disabled by a parent element.)

The `type` attribute on the `HTMLButtonElement` interface is changed from read-only to read-write. That interface is also given the `focus()` and `blur()` methods.

The `forms` attribute on most of the control interfaces is a live read-only `NodeList`, which shall contain the forms that the control is associated with, as determined by, and in the order given by, the `form` content attribute, or, if there is no `form` attribute, the element's ancestors.

The `form` attribute on most of the control interfaces is read-only, and shall return the first entry in the `forms` list, unless that list is empty, in which case it returns null.

The `forms` and `form` DOM attributes are live, so that adding a form with an ID that was previously listed in a control's `form` content attribute, for example, will cause that control to be associated with that form, and that form to appear in the DOM attributes.

For `HTMLOptionElement`s, `HTMLLegendElement`s, and `HTMLLabelElement`s, the `forms` and `form` DOM attributes shall reflect the attributes of their respective `HTMLSelectElement`, `HTMLFieldSetElement`, or (for labels) their associated form control, if they have one, and shall return null and empty (respectively) if they don't.

## 7.9. Additions specific to the `HTMLInputElement` interface

The `list` attribute is read-only, and reflects the current state of the `list` content attribute. That is, it returns the same as `getElementById()` would if passed the value of the `list` attribute, *if* that element is either an (X)HTML `select` or `datalist` element.

The `input` element's `selectedOption` DOM attribute is non-null only if the `datalist` DOM attribute is non-null and the `input` element's `value` DOM attribute is equal to the value of one of the `list`'s `option` descendants, in which case it points to that `option` element.

The `valueAsDate` attribute returns the `value` DOM attribute, converted to `DOMTimeStamp`. The control's value must be interpreted as appropriate for the given `type`, then converted to a `DOMTimeStamp` as follows:

### `datetime`

Simply express the given date and time.



**date**

Express the date as 00:00 UTC on that date (the first second of that date in UTC).

**time**

Express the time as 1970-01-01 at the given time, assuming the time zone is UTC.

**week**

Express the date as 00:00 UTC on the Monday of the given week (the first second of that ISO week, in UTC).

**month**

Express the date as 00:00 UTC on the first day of the given month (the first second of that month, in UTC).

**All other types (including datetime-local)**

Return NaN or an equivalent value representing an invalid date in the given language binding.

The `valueAsNumber` attribute returns the `value` DOM attribute, converted to `float`. The control's value must be converted as follows:

**datetime, date, month, week, time**

Express the date given by the `valueAsDate` attribute in terms of milliseconds since 1970-01-01 00:00 UTC.

**datetime-local**

Interpret the value as described for the type, then express the given date and time in terms of milliseconds since 1970-01-01 00:00.

**All other types**

Convert the value using algorithms equivalent to those specified in ECMA262 sections 9.3.1 ("ToNumber Applied to the String Type") and 8.5 ("The Number type"). [\[ECMA262\]](#)

Two new methods, `stepUp()` and `stepDown()`, enable authors to write code that increases and decreases (respectively) the value of the control in integral increments of the given `step` value. The argument specifies the number of steps by which to increase or decrease the value. If the control's value is invalid or empty, if the `step` attribute has the value `any`, or if the control is not a date-related, time-related, or numeric type, then the methods shall raise an `INVALID_STATE_ERR` exception. If a particular invocation of the method would invalidate the value (for example, if `stepUp(1)` is called when the control's value is less than `step` from `max`), then the method shall raise an `INVALID_MODIFICATION_ERR` exception. If the argument passed is zero, then the

method shall raise an `INDEX_SIZE_ERR` exception.

Setting a control's `value` DOM attribute dynamically in such a way that it makes the control invalid because the value doesn't conform to the syntax required by the type given in the `type` attribute must set the control to the value that the control would have had if it had never been given an explicit value.

Setting a control's `value` DOM attribute dynamically in such a way that it makes the control invalid because the value doesn't conform to the attributes setting constraints on the value, however, must cause the control to be set to the new value, and the control becoming invalid. (For example setting the value of a text field to a string that is longer than the `maxlength` attribute specifies would set the control to that long value, but then prevent submission.) If the control cannot be set to that value (for example, a range control cannot represent values outside its range) then the value must be clamped to the nearest value that can be represented by the control.

## 7.10. The `defaultValue` DOM attribute

For `HTMLInputElement` objects, the `defaultValue` DOM attribute must always mirror the `value` content attribute, both on getting and setting. Setting the `value` content attribute or the `defaultValue` DOM attribute must reset the `value` DOM attribute to the new default value *unless* the `value` DOM attribute for the element has been explicitly set by script or via user edits.

For `HTMLTextAreaElement` objects, the `defaultValue` DOM attribute must always mirror the `textContent` DOM attribute, both on getting and setting (so setting `defaultValue` changes the child nodes of the `textarea` element, and changing the child nodes of the element changes the `defaultValue` DOM attribute). Changing the child nodes of the `textarea` element, the `textContent` DOM attribute, or the `defaultValue` DOM attribute must not affect the `value` DOM attribute, even if the user has not changed the value.

## 7.11. Labels

Form controls all have a `labels` DOM attribute that lists all the `label` elements that refer to the control (either through the `for` attribute or via containership), in document order.

Similarly, `HTMLLabelElements` have a `control` DOM attribute that points to the associated element node, if any.

A label must be listed in the `labels` list of the control to which its `control` attribute points, and no other.

**Note:** Assistive technologies may use the labels attribute to determine what label to read out when a control is focused. An assistive technology could also wish to determine if the element is in a fieldset group. To do so, it should walk up the element's parentNode chain to find the fieldset ancestors.

## 7.12. Firing change events

The `dispatchChange()` and `dispatchFormChange()` methods must cause change and formchange events to be fired on the element. They are intended primarily to be used from `oninput` and `onforminput` handlers to avoid code duplication:

```
<input oninput="dispatchChange()" onchange="some long algorithm">
```

The change event fired by the dispatchChange() method must have the same default action as when that event is fired by the UA.

**Note:** The dispatchFormChange() method on form controls does not fire the event on all the controls like the dispatchFormChange() method on the HTMLFormElement interface.

## 7.13. Repetition interfaces

The `RepetitionElement` interface must be implemented by all elements.

If the element is a repetition template, its `repetitionType` DOM attribute must return `REPETITION_TEMPLATE`. Otherwise, if the element is a repetition block, it must return `REPETITION_BLOCK`. Otherwise, it is a normal element, and that attribute should return `REPETITION_NONE`.

Setting repetitionType modifies the repeat attribute. The repeat attribute's namespace depends on the element node's namespace; if the element is in the `http://www.w3.org/1999/xhtml` namespace then the attribute has no namespace, otherwise the attribute is in that namespace. If repetitionType is set to `REPETITION_NONE`, the attribute is removed. If it is set to `REPETITION_TEMPLATE`, the attribute is set to `"template"`. If repetitionType is set to `REPETITION_BLOCK`, the repeat content attribute is set to the value of the repetitionIndex DOM attribute.

The `repetitionIndex` attribute must return the current value of the index of the repetition template or block. If the element is a repetition block, setting this attribute must update the repeat attribute appropriately (and changing the

attribute directly must affect the value of the `repetitionIndex` attribute). Otherwise, if the element is a repetition template, setting this attribute changes the template's index but does not affect any other aspect of the DOM. If the element is a normal element, it must always return zero, and setting the attribute must have no effect.

The `repetitionTemplate` attribute is null unless the element is a repetition block, in which case it points to the block's template. If the block is an orphan repetition block then it returns null.

The `repetitionBlocks` attribute is null unless the element is a repetition template, in which case it points to a list of elements (an `HTMLCollection`, although the name of that interface is a misnomer since there is nothing HTML-specific about it). The list consists of all the repetition blocks that have this element as their template. The list is live.

The `repeatStart`, `repeatMin`, and `repeatMax` DOM attribute must reflect the values of the `repeat-start`, `repeat-min`, and `repeat-max` attributes respectively, those attributes being in no namespace if the element is in the `http://www.w3.org/1999/xhtml` namespace, and the `http://www.w3.org/1999/xhtml` namespace if the element is in another namespace. If those attributes are absent or do not have valid values, then the DOM attributes return their default values (1, 0, and the largest value that an `unsigned long` can hold, respectively). Setting the DOM attributes sets the relevant content attributes, unless the DOM attributes are set to their default values, in which case it removes the content attributes.

The `addRepetitionBlock()`, `addRepetitionBlockByIndex()`, `moveRepetitionBlock()` and `removeRepetitionBlock()` methods are defined in the section on [the repetition model](#).

The `htmlTemplate` DOM attribute on the `HTMLInputElement` and `HTMLButtonElement` interfaces represents the repetition template that the `template` content attribute refers to. If the content attribute points to a non-existent element or an element that is not a repetition template, the DOM attribute returns null. This DOM attribute is readonly in this version of this specification.

## 8. Presentation and rendering

### 8.1. Styling form controls

The CSS working group is expected to develop a language designed, amongst

other things, for the advanced styling of form controls. In the meantime, technologies such as [\[HTC\]](#) and [\[XBL\]](#) can be used as guides for what is expected.

UAs, in the absence of such advanced styling information, may render form controls described in this draft as they wish. It is recommended that form controls remain faithful to the look and feel of the system's global user interface.

***Note: CSS 2.1 explicitly does not define how CSS applies to form controls. [\[CSS21\]](#)***

## 8.2. Relation to CSS selectors

The W3C Selectors and CSS3 UI specifications define a number of pseudo-classes for form controls. [\[SELECTORS\]](#) [\[CSS3UI\]](#) Their relationship to the form controls described in this specification is described here.

### **:enabled**

Matches form control elements or `fieldset`s that do not have the `disabled` attribute set and that do not have any ancestor `fieldset` elements with their `disabled` attribute set, and that are not implicitly disabled (e.g. `move-up` buttons can be implicitly disabled in certain cases).

### **:disabled**

Matches form control elements or `fieldset`s that have their `disabled` attribute set or that have any ancestor `fieldset` elements with *their* `disabled` attribute set, or that are implicitly disabled (e.g. `move-up` buttons can be implicitly disabled in certain cases).

### **:checked**

Matches radio and checkbox form control elements that are `checked`.

### **:indeterminate**

Matches no HTML form control elements.

### **:default**

Matches [the button](#) (if any) that will be selected if the user presses the enter key (or some equivalent behaviour on less typical systems), as well as `input` elements of type `checkbox` and `radio` whose `defaultChecked` DOM attribute is true, and `option` elements whose `defaultSelected` DOM attribute is true.

### **:valid**

Matches form control elements whose `validity` objects have the `valid`

flag set.

**:invalid**

Matches form control elements whose validity objects have the valid flag cleared (not set).

**:in-range**

Matches numeric, date-related, or time-related form control elements which have a min or max attribute set and whose validity objects have none of the typeMismatch, rangeUnderflow, and rangeOverflow flags set.

**:out-of-range**

Matches numeric, date-related, or time-related form control elements which have a min or max attribute set and whose validity objects have one or both of the rangeUnderflow and rangeOverflow flags set.

**:required**

Matches form control elements that have the required attribute set.

**:optional**

Matches form control elements that do not have the required attribute set.

**:read-only**

Matches form control elements that have the readonly attribute set, and to which the readonly attribute applies (thus radio buttons will never match this, regardless of the value of the attribute), as well as elements defined by this specification that are not form controls (namely `form`, `label`, `datalist`, `option`, `optgroup`, and `fieldset` elements).

**:read-write**

Matches form control elements that do not have the readonly attribute set (including password fields, although technically they should be called "writeonly"), or to which the attribute doesn't apply (such as radio buttons). A disabled field can still match this pseudo-class; the states are orthogonal.

When the definitions above refer to elements that have an attribute set on them, they do not refer to elements on which that attribute is defined not to apply. For example, the `:read-only` attribute cannot apply to an `<input type="radio">` element, even if that element has the readonly attribute set, since readonly doesn't apply to radio buttons.

***Note: These pseudo-classes might also apply to other elements. This specification only gives the relationship between these pseudo-classes and the elements defined by this specification.***

*For example, the `:read-only` pseudo-class applies to most non-form elements in HTML, as well as those given above.*

*Note: The definition of the `:in-range` and pseudo-classes given here for HTML form controls does not quite match the generic definitions of the pseudo-classes given in CSS3 UI. This is intentional as the definitions given in CSS3 UI do not map well to the HTML semantics.*

### 8.3. Interaction of the form processing model with CSS

There is no interaction between the form processing model and CSS. For instance, making a control `display: none` does not have the slightest effect on whether the control is submitted or not. (Indeed, `input` elements of type `hidden` typically have `display: none` applies to them in the UA default stylesheet.)

### 8.4. Interaction of the form processing model with SMIL

There is no interaction between the form processing model and SMIL. Specifically, none of the attributes defined in this draft are animatable.

## A. XHTML module definition

*This section is non-normative.*

The Web Forms 2.0 Module provides all of the forms features found in HTML 4.0, plus the extensions described above. The form control elements and attributes of that the Web Forms 2.0 Module supports are:

Elements	Attributes	Minimal Content Model
form	<a href="#">Common</a> , accept ( <a href="#">ContentTypes</a> ), accept-charset ( <a href="#">Charsets</a> ), action ( <a href="#">URI</a> ), data ( <a href="#">URI</a> ), enctype ( <a href="#">ContentType</a> ), method ("get"*   "post"   "put"   "delete"), replace ("document"*   "values")	<a href="#">Flow</a> *
input	<a href="#">Common</a> , accept ( <a href="#">ContentTypes</a> ), accesskey ( <a href="#">Character</a> ), action ( <a href="#">URI</a> ), alt ( <a href="#">Text</a> ), autocomplete ("on"*   "off"), autofocus ("autofocus"), checked	EMPTY

Elements	Attributes	Minimal Content Model
	("checked"), disabled ("disabled"), enctype ( <a href="#">ContentType</a> ), form ( <a href="#">IDREF</a> ), inputmode ( <a href="#">CDATA</a> ), list ( <a href="#">IDREF</a> ), maxlength ( <a href="#">Number</a> ), method ("get"   "post"   "put"   "delete"), min ( <a href="#">CDATA</a> ), max ( <a href="#">CDATA</a> ), name ( <a href="#">CDATA</a> ), pattern ( <a href="#">CDATA</a> ), step ( <a href="#">CDATA</a> ), readonly ("readonly"), replace ("document"   "values") required ("required"), size ( <a href="#">Number</a> ), src ( <a href="#">URI</a> ), tabindex ( <a href="#">Number</a> ), template ( <a href="#">IDREF</a> ), type ("text"*   "password"   "checkbox"   "radio"   "button"   "submit"   "reset"   "add"   "remove"   "move-up"   "move-down"   "file"   "hidden"   "image"   "datetime"   "datetime-local"   "date"   "month"   "week"   "time"   "number"   "range"   "email"   "url"), value ( <a href="#">CDATA</a> )	
select	<a href="#">Common</a> , accesskey ( <a href="#">Character</a> ), autofocus ("autofocus"), data ( <a href="#">URI</a> ), disabled ("disabled"), form ( <a href="#">IDREF</a> ), multiple ("multiple"), name ( <a href="#">CDATA</a> ), size ( <a href="#">Number</a> ), tabindex ( <a href="#">Number</a> )	(optgroup   option)*
datalist	<a href="#">Common</a> , data ( <a href="#">URI</a> )	(option   Flow)*
optgroup	<a href="#">Common</a> , disabled ("disabled"), label* ( <a href="#">Text</a> )	(optgroup   option)*
option	<a href="#">Common</a> , disabled ("disabled"), label ( <a href="#">Text</a> ), selected ("selected"), value ( <a href="#">CDATA</a> )	PCDATA
textarea	<a href="#">Common</a> , accept ( <a href="#">ContentType</a> ), accesskey ( <a href="#">Character</a> ), autofocus ("autofocus"), cols ( <a href="#">Number</a> ), disabled ("disabled"), form ( <a href="#">IDREF</a> ), inputmode ( <a href="#">CDATA</a> ), maxlength ( <a href="#">Number</a> ), name ( <a href="#">CDATA</a> ), readonly ("readonly"), required ("required"), rows ( <a href="#">Number</a> ), tabindex ( <a href="#">Number</a> ), wrap ("soft"*   "hard")	PCDATA
output	<a href="#">Common</a> , for ( <a href="#">IDREFS</a> ), form ( <a href="#">IDREF</a> ), name ( <a href="#">CDATA</a> )	(PCDATA   Inline)*
button	<a href="#">Common</a> , accesskey ( <a href="#">Character</a> ), action ( <a href="#">URI</a> ), autofocus ("autofocus"), disabled ("disabled"), enctype ( <a href="#">ContentType</a> ), form ( <a href="#">IDREF</a> ), method ("get"   "post"   "put"   "delete"), name ( <a href="#">CDATA</a> ), replace ("document"   "values") tabindex ( <a href="#">Number</a> ), template ( <a href="#">IDREF</a> ), type ("button"   "submit"*   "reset"   "add"   "remove"   "move-up"   "move-down"), value ( <a href="#">CDATA</a> )	(PCDATA   Heading   List   Block - Form   Inline - Formctrl)*



Elements	Attributes	Minimal Content Model
fieldset	<a href="#">Common</a> , disabled ("disabled"), form ( <a href="#">IDREF</a> )	(PCDATA   legend   Flow)*
legend	<a href="#">Common</a> , accesskey ( <a href="#">Character</a> )	(PCDATA   Inline)*
label	<a href="#">Common</a> , accesskey ( <a href="#">Character</a> ), for ( <a href="#">IDREF</a> )	(PCDATA   Inline - label)*

This module defines two content sets:

### Form

form | fieldset | datalist

### Formctrl

input | select | textarea | output | button | label | datalist

When this module is used, it adds the `Form` content set to the `Block` content set and it adds the `Formctrl` content set to the `Inline` content set as these are defined in the Text Module.

All XHTML elements (all elements in the `http://www.w3.org/1999/xhtml` namespace) may have the `repeat`, `repeat-start`, `repeat-min`, `repeat-max`, and `repeat-template` attributes specified. Similarly, the global attributes `repeat`, `repeat-start`, `repeat-min`, `repeat-max`, and `repeat-template` in the `http://www.w3.org/1999/xhtml` namespace may be specified on any non-XHTML element. The `repeat` attribute must either have the value `"template"` or be an integer (an optional `'.'` character followed by one or more decimal digits). The `repeat-template` attribute must be an IDREF. The other attributes must have a non-negative integer value (one or more digits 0-9).

The `form` element may be placed inside XHTML `head` elements when it is empty.

The `oninput` attribute is added to all the elements that have an `onchange` attribute in the XHTML Intrinsic Events module. The `onformchange` and `onforminput` attributes are added to all form control elements (including `output`) and the `form` element. The `oninvalid` attribute is added to all form controls except `output` and `button` elements. The `onchange` attribute (but not `oninput`) is added to the `output` element.

When frames and multiple windows are also allowed, the `target` attribute is

added to the `form`, `input` and `button` elements.

The Web Forms 2.0 Module is a superset of the Forms and Basic Forms modules. These modules may not be used together in a single document type. Note that the content models in this module differ from those of the XHTML1 Forms module in some subtle ways (for example, the `select` element may be empty).

## B. Attribute summary

The `input` element takes a large number of attributes that do not always apply. The following table summarizes which attributes apply to which input types.

type	<u>text</u>	<u>password</u>	<u>checkbox</u> <u>radio</u>	<u>button</u>	<u>submit</u>	<u>reset</u>	<u>add</u>	<u>remove</u> <u>move-</u> <u>up</u> <u>move-</u> <u>down</u>	<u>file</u>	<u>list</u>
<u>accept</u>	-	-	-	-	-	-	-	-	Yes	-
<u>accesskey</u>	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	-
<u>action</u>	-	-	-	-	Yes	-	-	-	-	-
<u>alt</u>	-	-	-	-	-	-	-	-	-	-
<u>autocomplete</u>	Yes	Yes	-	-	-	-	-	-	-	-
<u>autofocus</u>	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	-
<u>checked</u>	-	-	Yes	-	-	-	-	-	-	-
<u>disabled</u>	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	-
<u>enctype</u>	-	-	-	-	Yes	-	-	-	-	-
<u>form</u>	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	-
<u>inputmode</u>	Yes	Yes	-	-	-	-	-	-	-	-
<u>list</u>	Yes	-	-	-	-	-	-	-	-	-
<u>maxlength</u>	Yes	Yes	-	-	-	-	-	-	Yes	-
<u>method</u>	-	-	-	-	Yes	-	-	-	-	-
<u>min</u>	-	-	-	-	-	-	-	-	Yes	-
<u>max</u>	-	-	-	-	-	-	-	-	Yes	-
<u>name</u>	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	-

type	<u>text</u>	<u>password</u>	<u>checkbox</u> <u>radio</u>	button	<u>submit</u>	reset	<u>add</u>	<u>remove</u> <u>move-</u> <u>up</u> <u>move-</u> <u>down</u>	<u>file</u>	<u>list</u>
<u>pattern</u>	Yes	Yes	-	-	-	-	-	-	-	-
<u>step</u>	-	-	-	-	-	-	-	-	-	-
<u>readonly</u>	Yes	Yes	-	-	-	-	-	-	-	-
<u>replace</u>	-	-	-	-	Yes	-	-	-	-	-
<u>required</u>	Yes	Yes	Yes	-	-	-	-	-	-	Yes
size	Yes	Yes	-	-	-	-	-	-	-	-
src	-	-	-	-	-	-	-	-	-	-
tabindex	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
target	-	-	-	-	Yes	-	-	-	-	-
<u>template</u>	-	-	-	-	-	-	Yes	-	-	-
value	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	-	-

## C. Deprecated features

The following features are deprecated by this specification:

- [The `size` attribute on the `input` element.](#)

Documents must not use deprecated features. User agents should support deprecated features.

## D. Requirements for declaring interoperability

The [WHAT working group charter](#) requires that for this specification to be considered interoperably implemented, the tests for each *feature* must be passed by two implementations. The features of this specification are, for the purpose of testing interoperability:

- Each subsection of the [Extensions to form control elements](#) section.
- The [The repetition model for repeating form controls](#) section.

- Each subsection of the [The forms event model](#) section.
- The [Form submission](#) section.
- Each subsection of the [Fetching data from external resources](#) section.
- Each subsection of the [Extensions to the HTML Level 2 DOM interfaces](#) section.

## References

All references are normative unless marked "Informative".

### [CHARMOD]

[Character Model for the World Wide Web 1.0](#), M. Dürst, F. Yergeau, R. Ishida, M. Wolf, T. Texin. W3C, February 2005. The latest version of the Character Model specification is available at <http://www.w3.org/TR/charmod/>

### [CSJSR]

[Client-Side JavaScript Reference](#) (1.3). Netscape Communications Corporation, May 1999. The Client-Side JavaScript Reference (1.3) is available at <http://devedge.netscape.com/library/manuals/2000/javascript/1.3/reference/index.html>

### [CSS21]

[CSS 2.1 Specification](#), B. Bos, T. Çelik, I. Hickson, H. Lie. W3C, September 2003. The latest version of the CSS 2.1 specification is available at <http://www.w3.org/TR/CSS21>

### [CSS3CONTENT]

[CSS3 Generated and Replaced Content Module](#), I. Hickson. W3C, May 2003. The latest version of the CSS3 Generated and Replaced Content module is available at <http://www.w3.org/TR/css3-content>

### [CSS3UI]

[CSS3 Basic User Interface Module](#), T. Çelik. W3C, May 2004. The latest version of the CSS3 UI module is available at <http://www.w3.org/TR/css3-ui>

### [DOM2HTML]

[Document Object Model \(DOM\) Level 2 HTML Specification](#), J. Stenback, P. Le Hégarret, A. Le Hors. W3C, January 2003. The latest version of the DOM Level 2 HTML specification is available at <http://www.w3.org>

[/TR/DOM-Level-2-HTML/](#)

**[DOM3CORE]**

[Document Object Model \(DOM\) Level 3 Core Specification](#), A. Le Hors, P. Le Hégarret, L. Wood, G. Nicol, J. Robie, M. Champion, S. Byrne. W3C, November 2003. The latest version of the DOM Level 3 Core specification is available at <http://www.w3.org/TR/DOM-Level-3-Core/>

**[DOM3EVENTS]**

[Document Object Model \(DOM\) Level 3 Events Specification](#), P. Le Hégarret, T. Pixley. W3C, November 2003. (Note: Despite its non-normative status on the W3C Recommendation track, this specification should be considered normative for the purposes of conformance.) The latest version of the DOM Level 3 Events specification is available at <http://www.w3.org/TR/DOM-Level-3-Events/>

**[ECMA262]**

[ECMAScript Language Specification](#), Third Edition. ECMA, December 1999. This version of the ECMAScript Language is available at <http://www.ecma-international.org/publications/standards/Ecma-262.htm>

**[HTC]**

(Informative) [HTML Components](#), Chris Wilson. Microsoft, September 1998. The HTML Components submission is available at <http://www.w3.org/TR/1998/NOTE-HTMLComponents-19981023>

**[HTML4]**

[HTML 4.01 Specification](#), D. Raggett, A. Le Hors, I. Jacobs. W3C, December 1999. The latest version of the HTML4 specification is available at <http://www.w3.org/TR/html4>

**[ISO8601]**

[ISO8601:2004 Data elements and interchange formats -- Information interchange -- Representation of dates and times](#). ISO, December 2004. ISO 8601 is available for purchase at <http://www.iso.ch/>

**[RFC959]**

[File Transfer Protocol \(FTP\)](#), J. Postel, J. Reynolds. IETF, October 1985. RFC 959 is available at <http://www.ietf.org/rfc/rfc959>

**[RFC1738]**

[Uniform Resource Locators \(URL\)](#), T. Berners-Lee, L. Masinter, M. McCahill. IETF, December 1994. RFC 1738 is available at <http://www.ietf.org/rfc/rfc1738>

**[RFC1866]**

(Informative) [Hypertext Markup Language - 2.0](#), T. Berners-Lee, D. Connolly. IETF, November 1995. RFC 1866 is available at <http://www.ietf.org/rfc/rfc1866>

**[RFC2045]**

[Multipurpose Internet Mail Extensions \(MIME\) Part One: Format of Internet Message Bodies](#), N. Freed, N. Borenstein. IETF, November 1996. RFC 2045 is available at <http://www.ietf.org/rfc/rfc2045>

**[RFC2046]**

[Multipurpose Internet Mail Extensions \(MIME\) Part Two: Media Types](#), N. Freed, N. Borenstein. IETF, November 1996. RFC 2046 is available at <http://www.ietf.org/rfc/rfc2046>

**[RFC2119]**

[Key words for use in RFCs to Indicate Requirement Levels](#), S. Bradner. IETF, March 1997. RFC 2119 is available at <http://www.ietf.org/rfc/rfc2119>

**[RFC2368]**

[The mailto URL scheme](#), P. Hoffman, L. Masinter, J. Zawinski. IETF, July 1998. RFC 2368 is available at <http://www.ietf.org/rfc/rfc2368>

**[RFC2397]**

[The "data" URL scheme](#), L. Masinter. IETF, August 1998. RFC 2397 is available at <http://www.ietf.org/rfc/rfc2397>

**[RFC2388]**

[Returning Values from Forms: multipart/form-data](#), L. Masinter. IETF, August 1998. RFC 2388 is available at <http://www.ietf.org/rfc/rfc2388>

**[RFC3986]**

[Uniform Resource Identifier \(URI\): Generic Syntax](#), T. Berners-Lee, R. Fielding, L. Masinter. IETF, January 2005. RFC 3986 is available at <http://www.ietf.org/rfc/rfc3986>

**[RFC3987]**

[Internationalized Resource Identifiers \(IRIs\)](#), M. Duerst, M. Suignard. IETF, January 2005. RFC 3987 is available at <http://www.ietf.org/rfc/rfc3987>

**[RFC2616]**

[Hypertext Transfer Protocol -- HTTP/1.1](#), R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee. IETF, June 1999. RFC 2616 is available at <http://www.ietf.org/rfc/rfc2616>

**[RFC2822]**

[Internet Message Format](#), P. Resnick. IETF, April 2001. RFC 2822 is available at <http://www.ietf.org/rfc/rfc2822>

**[RFC3023]**

[\*XML Media Types\*](#), M. Murata, S. St.Laurent, D. Kohn. IETF, January 2001. RFC 3023 is available at <http://www.ietf.org/rfc/rfc3023>

**[RFC3106]**

[\*ECML v1.1: Field Specifications for E-Commerce\*](#), D. Eastlake, T Goldstein. IETF, April 2001. RFC 3106 is available at <http://www.ietf.org/rfc/rfc3106>

**[RFC3490]**

[\*Internationalizing Domain Names in Applications \(IDNA\)\*](#), P. Faltstrom, P. Hoffman, A. Costello. IETF, March 2003. RFC 3490 is available at <http://www.ietf.org/rfc/rfc3490>

**[RFC3875]**

[\*The Common Gateway Interface \(CGI\) Version 1.1\*](#), D. Robinson, K. Coar. IETF, October 2004. RFC 3875 is available at <http://www.ietf.org/rfc/rfc3875>

**[SELECTORS]**

[\*Selectors\*](#), D. Glazman, T. Çelik, I. Hickson, P. Linss, J. Williams. W3C, November 2001. The latest version of the Selectors specification is available at <http://www.w3.org/TR/css3-selectors>

**[UNICODE]**

[\*The Unicode Standard, Version 4.1\*](#), The Unicode Consortium. Boston, MA, Addison-Wesley, March 2005. ISBN 0-321-18578-1, as amended by [Unicode 4.0.1](#) and [Unicode 4.1.0](#). The latest version of the Unicode specification is available at <http://www.unicode.org/versions/>

**[XBL]**

(Informative) [\*XML Binding Language\*](#), David Hyatt. Mozilla, February 2001. The XBL submission is available at <http://www.w3.org/TR/2001/NOTE-xbl-20010223/>

**[XForms]**

[\*XForms 1.0\*](#), M. Dubinko, L. Klotz, R. Merrick, T. Raman. W3C, October 2003. The latest version of the XForms specification is available at <http://www.w3.org/TR/xforms>

**[XHTML1]**

[\*XHTML™ 1.1 - Module-based XHTML\*](#), M. Altheim, S. McCarron. W3C, May 2001. The latest version of the XHTML 1.1 specification is available at <http://www.w3.org/TR/xhtml11>

**[XML]**

[\*Extensible Markup Language \(XML\) 1.0 \(Third Edition\)\*](#), T Bray, J Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau. W3C, February 2004. The latest version of the XML specification is available at <http://www.w3.org/TR/REC-xml/>

## Acknowledgements

This work is the direct result of discussion on the WHATWG public mailing list.

Thanks to Alan Plum, Allan Clements, Alexander J. Vincent, Andrew Clover, Andrew Smith, Andy Heydon, Anne van Kesteren, Anthony Boyd, Ave Wrigley, Bert Bos, Bill McCoy, Björn Hörmann, Boris Zbarsky, Brad Fults, Brendan Eich, Brian Korver, Brian Ryner, Brian Wilson, C. Williams, Chris Morris, Christian Biesinger, Christian Schmidt, Christopher Aillon, Craig Cockburn, Csaba Gabor, Daniel Bratell, Daniel Brooks, Dave Hodder, David Baron, David E. Cleary, David Hyatt, David Matja, Dean Edwards, dolphinling, Doron Rosenberg, Edmund Lai, Edward Welbourne, Eira Monstad, Eric Rescorla, fantasai, Gytis Jakutonis, Håkon Wium Lie, Hallvord Reiar Michaelsen Steen, Henri Sivonen, Ian Bicking, James Graham, Jason Kersey, Jason Lustig, Jens Lindström, Jim Ley, Joe Gregorio, John Keiser, Johnny Stenback, Jon Ferraiolo, Jonas Sicking, Jonny Axelsson, Jorunn Danielsen Newth, Jukka K. Korpela, Justin Sinclair, Lachlan Hunt, Laurens Holst, Maciej Stachowiak, Mark Birbeck, Mark Nottingham, Mark Schenk, Mark Wilton-Jones, Martijn Wargers, Martin Honnen, Martin Kutschker, Matt Wright, Matthew Mastracci, Matthew Thomas, Mattias Walda, Max Romantschuk, Menno van Slooten, Micah Dubinko, Michael A. Nachbaur, Michael Daskalov, Michael Enright, Mike Shaver, Mikko Rantalainen, Neil Rashbrook, Olav Junker Kjær, Olli Pettay, Paul Norman, Peter Stark, Peter-Paul Koch, Rene Stach, Rich Doughty, Rigo Wenning, Sander van Lambalgen, Sebastian Schnitzenbaumer, Shanti Rao, Sigbjørn Vik, Simon Montagu, Simon Pieters, Stuart Ballard, Subramanian Peruvemba, Susan Borgrink, Tantek Çelik, Ted Mielczarek, Tom Pike, voracity, Will Levine, and Wladimir Palant for their comments, both large and small. Thanks to the W3C QA Working Group for providing useful guidelines for specification authors. Thanks also to the Slashdot, Mozillazine, and My Opera communities for some ideas, and to the #mozilla crew, the #opera crew, and the #mrt crew for their ideas and support.

The editor would like to convey very special thanks to Malcolm Rowe and Matthew Raymond for their help fielding comments in the WHATWG mailing list.