Document: execCommand() method

- Syntax
  - Parameters
  - Return value
- Examples
  - Using insertText
    - HTML
    - JavaScript
    - Result
  - Using paste
    - HTML
    - JavaScript
    - Result
- Specifications
- Browser compatibility
- See also

> **NOTE:** Although the `execCommand()` method is deprecated, there are still some valid use cases that do not yet have viable alternatives. For example, unlike direct DOM manipulation, modifications performed by `execCommand()` preserve the undo buffer (edit history). For these use cases, you can still use this method, but test to ensure cross-browser compatibility, such as by using document.queryCommandSupported().

The **`execCommand`** method implements multiple different commands. Some of them provide access to the clipboard, while others are for editing form inputs, `contenteditable` elements or entire documents (when switched to design mode).

To access the clipboard, the newer Clipboard API is recommended over `execCommand()`.

Most commands affect the document's selection. For example, some commands (bold, italics, etc.) format the currently selected text, while others delete the selection, insert new elements (replacing the selection) or affect an entire line (indenting). Only the currently active editable element can be modified, but some commands (e.g., `copy`) can work without an editable element.

> **NOTE:** Modifications performed by `execCommand()` may or may not trigger beforeinput and input events, depending on the browser and configuration. If triggered, the handlers for the events will run before `execCommand()` returns. Authors need to be careful about such recursive calls, especially if they call `execCommand()` in response to these events. From Firefox 82, nested `execCommand()` calls will always fail, see bug 1634262.

Syntax

```
execCommand(commandName, showDefaultUI, valueArgument)
```

Parameters

- `commandName`
  - : A string specifying the name of the command to execute. The following commands are specified:
    - `backColor`
      - : Changes the document background color. In `styleWithCss` mode, it affects the background color of the containing block instead. This requires a <color> value string to be passed in as a value argument.
    - `bold`
      - : Toggles bold on/off for the selection or at the insertion point.
    - `contentReadOnly`
      - : Makes the content document either read-only or editable. This requires a boolean true/false as the value argument.
    - `copy`
      - : Copies the current selection to the clipboard. Conditions of having this behavior enabled vary from one browser to another, and have evolved over time. Check the compatibility table to determine if you can use it in your case.
    - `createLink`

    - : Creates a hyperlink from the selection, but only if there is a selection. Requires a URI string as a value argument for the hyperlink's `href`. The URI must contain at least a single character, which may be whitespace.
- `cut`
    - : Removes the current selection and copies it to the clipboard. When this behavior is enabled varies between browsers, and its conditions have evolved over time. Check the compatibility table for usage details.
- `decreaseFontSize`
    - : Adds a small tag around the selection or at the insertion point.
- `defaultParagraphSeparator`
    - : Changes the paragraph separator used when new paragraphs are created in editable text regions.
- `delete`
    - : Deletes the current selection.
- `enableAbsolutePositionEditor`
    - : Enables or disables the grabber that allows absolutely-positioned elements to be moved around. The grabber is disabled by default since Firefox 64 (Firefox bug 1490641).
- `enableInlineTableEditing`
    - : Enables or disables the table row/column insertion and deletion controls. The controls are disabled by default since Firefox 64 (Firefox bug 1490641).
- `enableObjectResizing`
    - : Enables or disables the resize handles on images, tables, and absolutely-positioned elements and other resizable objects. The handles are disabled by default since Firefox 64 (Firefox bug 1490641).
- `fontName`
    - : Changes the font name for the selection or at the insertion point. This requires a font name string (like `"Arial"`) as a value argument.
- `fontSize`
    - : Changes the font size for the selection or at the insertion point. This requires an integer from `1` - `7` as a value argument.
- `foreColor`
    - : Changes a font color for the selection or at the insertion point. This requires a hexadecimal color value string as a value argument.
- `formatBlock`
    - : Adds an HTML block-level element around the line containing the current selection, replacing the block element containing the line if one exists (in Firefox, blockquote is the exception — it will wrap any containing block element). Requires a tag-name string as a value argument. Virtually all block-level elements can be used. (Legacy Edge only supports heading tags `H1` – `H6`, `ADDRESS`, and `PRE`, which must be wrapped in angle brackets, such as `"<H1>"`.)

- `forwardDelete`
  - : Deletes the character ahead of the cursor's position, identical to hitting the Delete key on a Windows keyboard.
- `heading`
  - : Adds a heading element around a selection or insertion point line. Requires the tag-name string as a value argument (i.e., `"H1"`, `"H6"`). (Not supported by Safari.)
- `hiliteColor`
  - : Changes the background color for the selection or at the insertion point. Requires a color value string as a value argument. `useCSS` must be `true` for this to function.
- `increaseFontSize`
  - : Adds a big tag around the selection or at the insertion point.
- `indent`
  - : Indents the line containing the selection or insertion point. In Firefox, if the selection spans multiple lines at different levels of indentation, only the least indented lines in the selection will be indented.
- `insertBrOnReturn`
  - : Controls whether the Enter key inserts a br element, or splits the current block element into two.
- `insertHorizontalRule`
  - : Inserts a hr element at the insertion point, or replaces the selection with it.
- `insertHTML`
  - : Inserts a TrustedHTML instance or string of HTML markup at the insertion point (deletes selection). This requires valid HTML markup.

    > **WARNING:** The input is parsed as HTML and written into the DOM. APIs like this are known as injection sinks, and are potentially a vector for cross-site scripting (XSS) attacks, if the input originally came from an attacker.
    > You can mitigate this risk by always assigning TrustedHTML objects instead of strings and enforcing trusted types. See the Trusted Types API for more information.

- `insertImage`
  - : Inserts an image at the insertion point (deletes selection). Requires a URL string for the image's `src` as a value argument. The requirements for this string are the same as `createLink`.
- `insertLineBreak`

- : Deletes the selection, and replaces it with a line break element.
- `insertOrderedList`
  - : Creates a numbered ordered list for the selection or at the insertion point.
- `insertUnorderedList`
  - : Creates a bulleted unordered list for the selection or at the insertion point.
- `insertParagraph`
  - : Inserts a paragraph around the selection or the current line.
- `insertText`
  - : Inserts the given plain text at the insertion point (deletes selection).
- `italic`
  - : Toggles italics on/off for the selection or at the insertion point.
- `justifyCenter`
  - : Centers the selection or insertion point.
- `justifyFull`
  - : Justifies the selection or insertion point.
- `justifyLeft`
  - : Justifies the selection or insertion point to the left.
- `justifyRight`
  - : Right-justifies the selection or the insertion point.
- `outdent`
  - : Outdents the line containing the selection or insertion point.
- `paste`
  - : Pastes the clipboard contents at the insertion point (replaces current selection).
    This feature is specified as disabled for *web content*, but has been implemented via the Clipboard API on some browsers. On these browsers the feature requires transient activation, and acknowledgement of a popup UI when pasting cross-origin content. See the Browser compatibility table for more information.
- `redo`
  - : Redoes the previous undo command.
- `removeFormat`
  - : Removes all formatting from the current selection.
- `selectAll`
  - : Selects all of the content of the editable region.
- `strikeThrough`
  - : Toggles strikethrough on/off for the selection or at the insertion point.
- `subscript`
  - : Toggles subscript on/off for the selection or at the insertion point.
- `superscript`

    - : Toggles superscript on/off for the selection or at the insertion point.
  - `underline`
    - : Toggles underline on/off for the selection or at the insertion point.
  - `undo`
    - : Undoes the last executed command.
  - `unlink`
    - : Removes the anchor element from a selected hyperlink.
  - `useCSS`
    - : Toggles the use of HTML tags or CSS for the generated markup. Requires a boolean true/false as a value argument.

      > **NOTE:** This argument is logically backwards (i.e., use `false` to use CSS, `true` to use HTML). This has been deprecated in favor of `styleWithCSS`.

  - `styleWithCSS`
    - : Replaces the `useCSS` command. `true` modifies/generates `style` attributes in markup, false generates presentational elements.
  - `AutoUrlDetect`
    - : Changes the browser auto-link behavior.
- `showDefaultUI`
  - : A boolean value indicating whether the default user interface should be shown. This is not implemented in Mozilla.
- `valueArgument`
  - : For commands which require an input argument, is a string providing that information. For example, `insertImage` requires the URL of the image to insert. Specify `null` if no argument is needed.

Return value

A boolean value that is `false` if the command is unsupported or disabled.

> **NOTE:** `document.execCommand()` only returns `true` if it is invoked as part of a user interaction. You can't use it to verify browser support before calling a command.

Examples

Using insertText

This example shows two very basic HTML editors, one using a textarea element and one using a pre element with the `contenteditable` attribute set.

Clicking the "Bold" or "Italic" buttons inserts the appropriate tags in the element, using `insertText` to preserve the edit history, so the user can undo the action.

HTML

```
<h2>textarea</h2>

<div class="actions" data-for="textarea">
  <button data-el="b">Bold</button>
  <button data-el="i">Italic</button>
</div>

<textarea class="editarea">Some text.</textarea>

<h2>contenteditable</h2>

<div class="actions" data-for="pre">
  <button data-el="b">Bold</button>
  <button data-el="i">Italic</button>
</div>

<pre contenteditable="true" class="editarea">Some text.</pre>
```

JavaScript

```
// Prepare action buttons
const buttonContainers = document.querySelectorAll(".actions");

for (const buttonContainer of buttonContainers) {
  const buttons = buttonContainer.querySelectorAll("button");
  const pasteTarget = buttonContainer.getAttribute("data-for");

  for (const button of buttons) {
    const elementName = button.getAttribute("data-el");
```

```
    button.addEventListener("click", () =>
      insertText(`<${elementName}></${elementName}>`, pasteTarget),
    );
  }
}

// Inserts text at cursor, or replaces selected text
function insertText(newText, selector) {
  const textarea = document.querySelector(selector);
  textarea.focus();

  let pasted = true;
  try {
    if (!document.execCommand("insertText", false, newText)) {
      pasted = false;
    }
  } catch (e) {
    console.error("error caught:", e);
    pasted = false;
  }

  if (!pasted) {
    console.error("paste unsuccessful, execCommand not supported");
  }
}
```

Result

Using paste

This example has a textarea element, and a button element that you can use to paste content into it.

HTML

```
<button id="paste">Paste</button>
<hr />
<textarea id="text_box">Some text.</textarea>
```

JavaScript

```
const pasteButton = document.querySelector("#paste");
const textBox = document.querySelector("#text_box");

pasteButton.addEventListener("click", () => {
  textBox.focus();

  let pasted = document.execCommand("paste", false);
  if (!pasted) {
    textBox.textContent = "paste unsuccessful, execCommand not supported";
  }
});
```

Result

On browsers that implement this feature using the Clipboard API you should be able to copy same-origin content, such as text from the text area, and then paste it to replace any selected content. When you try to paste cross-origin content, such as text copied from any other page or location, you will first need to select the "Paste" UI that is displayed.

Specifications

This feature is not part of any current specification. It is no longer on track to become a standard.

There is an unofficial W3C execCommand spec draft.

Browser compatibility

See also

- Clipboard API
- MDN example: execCommands supported in your browser
- HTMLElement.contentEditable
- document.designMode
- document.queryCommandEnabled()

- document.queryCommandState()
- document.queryCommandSupported()

□□□ #3
□ □□□ □□□ 2026-10-31 03:27:53 UTC
□ cocobook-sync □□□ 2026-11-04 17:33:43 UTC