14th January 2020

Selection and Range

In this chapter we'll cover selection in the document, as well as selection in form fields, such as `<input>` .

JavaScript can get the existing selection, select/deselect both as a whole or partially, remove the selected part from the document, wrap it into a tag, and so on.

You can get ready to use recipes at the end, in "Summary" section. But you'll get much more if you read the whole chapter. The underlying `Range` and `Selection` objects are easy to grasp, and then you'll need no recipes to make them do what you want.

Range

The basic concept of selection is Range: basically, a pair of "boundary points": range start and range end.

Each point represented as a parent DOM node with the relative offset from its start. If the parent node is an element node, then the offset is a child number, for a text node it's the position in the text. Examples to follow.

Let's select something.

First, we can create a range (the constructor has no parameters):

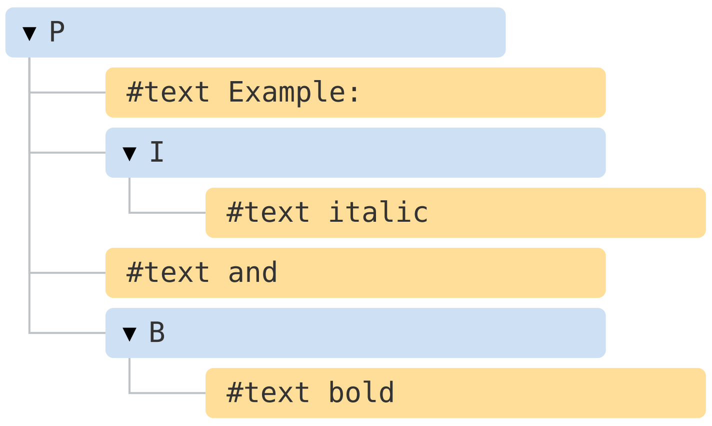
```
let range = new Range();
```

Then we can set the selection boundaries using `range.setStart(node, offset)` and `range.setEnd(node, offset)` .

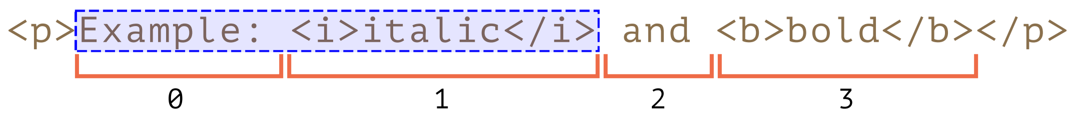
For example, consider this fragment of HTML:

```
<p id="p">Example: <i>italic</i> and <b>bold</b></p>
```

Here's its DOM structure, note that here text nodes are important for us:



Let’s select `"Example: <i>italic</i>"`. That’s two first children of `<p>` (counting text nodes):



```html
<p id="p">Example: <i>italic</i> and <b>bold</b></p>

<script>
  let range = new Range();

  range.setStart(p, 0);
  range.setEnd(p, 2);

  // toString of a range returns its content as text (without tags)
  alert(range); // Example: italic

  // apply this range for document selection (explained later)
  document.getSelection().addRange(range);
</script>
```

- `range.setStart(p, 0)` – sets the start at the 0th child of `<p>` (that’s the text node `"Example: "`).
- `range.setEnd(p, 2)` – spans the range up to (but not including) 2nd child of `<p>` (that’s the text node `" and "`, but as the end is not included, so the last selected node is `<i>`).

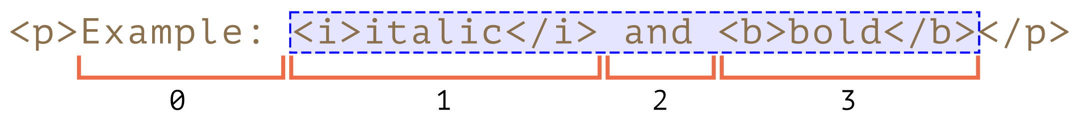
Here’s a more flexible test stand where you try more variants:

```html
<p id="p">Example: <i>italic</i> and <b>bold</b></p>

From <input id="start" type="number" value=1> – To <input id="end" type="numb
<button id="button">Click to select</button>
<script>
  button.onclick = () => {
    let range = new Range();

    range.setStart(p, start.value);
    range.setEnd(p, end.value);

    // apply the selection, explained later
    document.getSelection().removeAllRanges();
    document.getSelection().addRange(range);
  };
</script>
```

Example: *italic* and **bold**

From 1 – To 4 Click to select

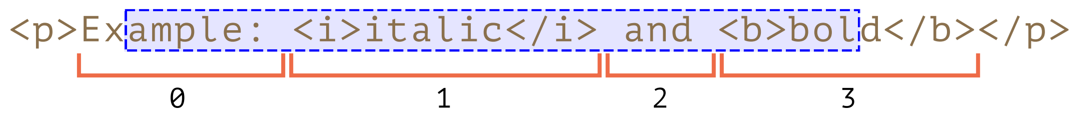
E.g. selecting from `1` to `4` gives range `<i>italic</i> and <b>bold</b>` .



We don't have to use the same node in `setStart` and `setEnd` . A range may span across many unrelated nodes. It's only important that the end is after the start.

Selecting parts of text nodes

Let's select the text partially, like this:



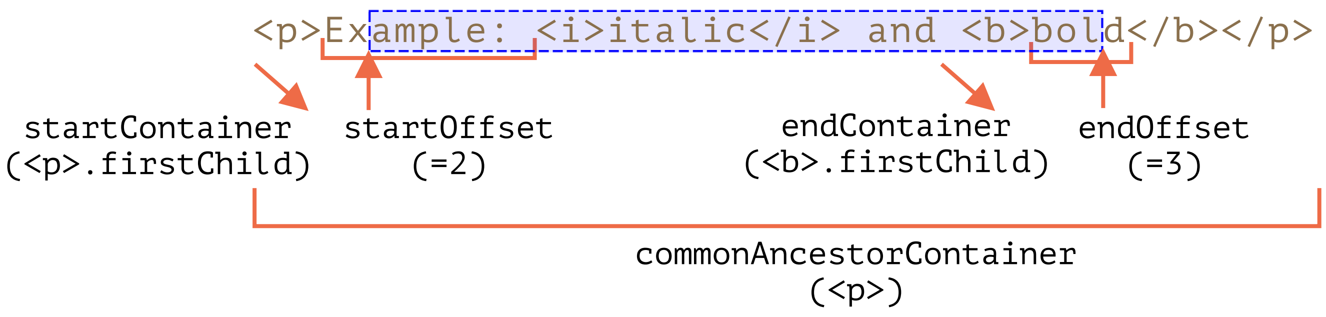
That's also possible, we just need to set the start and the end as a relative offset in text nodes.

We need to create a range, that:

- starts from position 2 in `<p>` first child (taking all but two first letters of "Ex**ample:** ")
- ends at the position 3 in `<b>` first child (taking first three letters of "**bol**d", but no more):

```
<p id="p">Example: <i>italic</i> and <b>bold</b></p>

<script>
  let range = new Range();

  range.setStart(p.firstChild, 2);
  range.setEnd(p.querySelector('b').firstChild, 3);

  alert(range); // ample: italic and bol

  // use this range for selection (explained later)
  window.getSelection().addRange(range);
</script>
```

The range object has following properties:



<p>Example: <i>italic</i> and <b>bold</b></p>
startContainer (<p>.firstChild)
startOffset (=2)
endContainer (<b>.firstChild)
endOffset (=3)
commonAncestorContainer (<p>)

- `startContainer` , `startOffset` – node and offset of the start,
  - in the example above: first text node inside `<p>` and `2` .
- `endContainer` , `endOffset` – node and offset of the end,
  - in the example above: first text node inside `<b>` and `3` .
- `collapsed` – boolean, `true` if the range starts and ends on the same point (so there's no content inside the range),
  - in the example above: `false`
- `commonAncestorContainer` – the nearest common ancestor of all nodes within the range,
  - in the example above: `<p>`

Range methods

There are many convenience methods to manipulate ranges.

Set range start:

- `setStart(node, offset)` set start at: position `offset` in `node`
- `setStartBefore(node)` set start at: right before `node`
- `setStartAfter(node)` set start at: right after `node`

Set range end (similar methods):

- `setEnd(node, offset)` set end at: position `offset` in `node`
- `setEndBefore(node)` set end at: right before `node`
- `setEndAfter(node)` set end at: right after `node`

As it was demonstrated, `node` can be both a text or element node: for text nodes `offset` skips that many of characters, while for element nodes that many child nodes.

Others:

- `selectNode(node)` set range to select the whole `node`
- `selectNodeContents(node)` set range to select the whole `node` contents
- `collapse(toStart)` if `toStart=true` set end=start, otherwise set start=end, thus collapsing the range

- `cloneRange()` creates a new range with the same start/end

To manipulate the content within the range:

- `deleteContents()` – remove range content from the document
- `extractContents()` – remove range content from the document and return as DocumentFragment
- `cloneContents()` – clone range content and return as DocumentFragment
- `insertNode(node)` – insert `node` into the document at the beginning of the range
- `surroundContents(node)` – wrap `node` around range content. For this to work, the range must contain both opening and closing tags for all elements inside it: no partial ranges like `<i>abc` .

With these methods we can do basically anything with selected nodes.

Here's the test stand to see them in action:

```
Click buttons to run methods on the selection, "resetExample" to reset it.

<p id="p">Example: <i>italic</i> and <b>bold</b></p>

<p id="result"></p>
<script>
  let range = new Range();

  // Each demonstrated method is represented here:
  let methods = {
    deleteContents() {
      range.deleteContents()
    },
    extractContents() {
      let content = range.extractContents();
      result.innerHTML = "";
      result.append("extracted: ", content);
    },
    cloneContents() {
      let content = range.cloneContents();
      result.innerHTML = "";
      result.append("cloned: ", content);
    },
    insertNode() {
      let newNode = document.createElement('u');
      newNode.innerHTML = "NEW NODE";
      range.insertNode(newNode);
    },
    surroundContents() {
      let newNode = document.createElement('u');
      try {
        range.surroundContents(newNode);
      } catch(e) { alert(e) }
    },
    resetExample() {
      p.innerHTML = `Example: <i>italic</i> and <b>bold</b>`;
      result.innerHTML = "";

      range.setStart(p.firstChild, 2);
      range.setEnd(p.querySelector('b').firstChild, 3);

      window.getSelection().removeAllRanges();
      window.getSelection().addRange(range);
    }
  };

  for(let method in methods) {
    document.write(`<div><button onclick="methods.${method}()">${method}</but
  }

  methods.resetExample();
</script>
```

Click buttons to run methods on the selection, "resetExample" to reset it.

Example: *italic* and **bold**

deleteContents
extractContents
cloneContents
insertNode
surroundContents
resetExample

There also exist methods to compare ranges, but these are rarely used. When you need them, please refer to the spec or MDN manual.

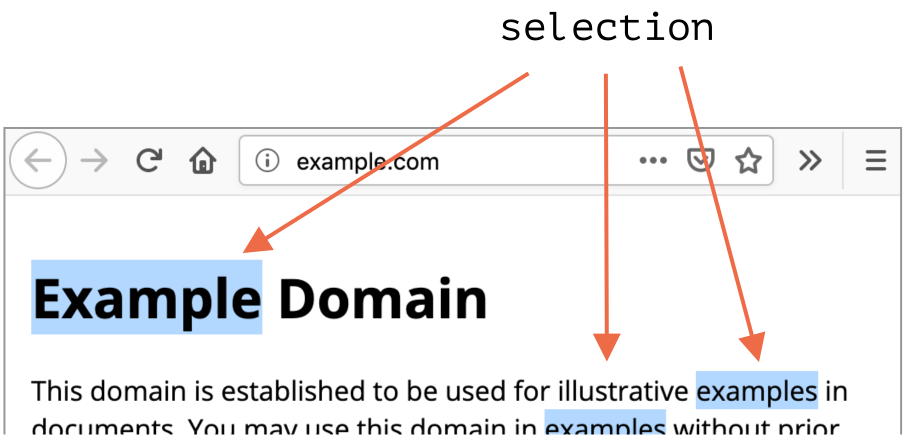
Selection

`Range` is a generic object for managing selection ranges. We may create such objects, pass them around – they do not visually select anything on their own.

The document selection is represented by `Selection` object, that can be obtained as `window.getSelection()` or `document.getSelection()` .

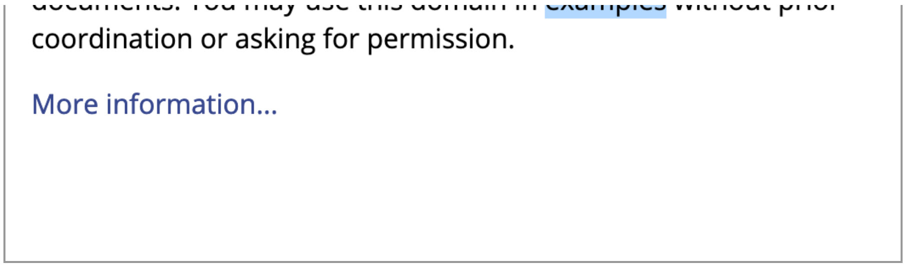
A selection may include zero or more ranges. At least, the Selection API specification says so. In practice though, only Firefox allows to select multiple ranges in the document by using `Ctrl+click` (`Cmd+click` for Mac).

Here’s a screenshot of a selection with 3 ranges, made in Firefox:



Example Domain

This domain is established to be used for illustrative examples in documents. You may use this domain in examples without prior



documents. You may use this domain in examples without prior coordination or asking for permission.

More information...

Other browsers support at maximum 1 range. As we'll see, some of `Selection` methods imply that there may be many ranges, but again, in all browsers except Firefox, there's at maximum 1.

Selection properties

Similar to a range, a selection has a start, called "anchor", and the end, called "focus".

The main selection properties are:

- `anchorNode` – the node where the selection starts,
- `anchorOffset` – the offset in `anchorNode` where the selection starts,
- `focusNode` – the node where the selection ends,
- `focusOffset` – the offset in `focusNode` where the selection ends,
- `isCollapsed` – `true` if selection selects nothing (empty range), or doesn't exist.
- `rangeCount` – count of ranges in the selection, maximum `1` in all browsers except Firefox.

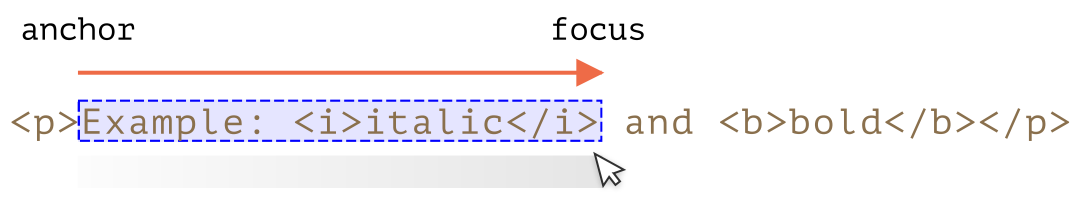
Selection end may be in the document before start

There are many ways to select the content, depending on the user agent: mouse, hotkeys, taps on a mobile etc.

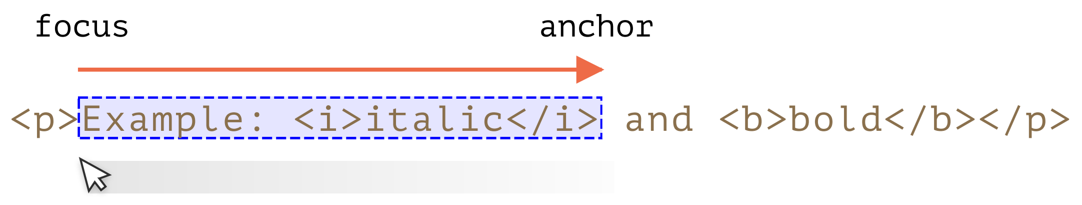
Some of them, such as a mouse, allow the same selection can be created in two directions: “left-to-right” and “right-to-left”.

If the start (anchor) of the selection goes in the document before the end (focus), this selection is said to have “forward” direction.

E.g. if the user starts selecting with mouse and goes from “Example” to “italic”:



Otherwise, if they go from the end of “italic” to “Example”, the selection is directed “backward”, its focus will be before the anchor:



That’s different from `Range` objects that are always directed forward: the range start can’t be after its end.

Selection events

There are events on to keep track of selection:

- `elem.onselectstart` – when a selection starts on `elem`, e.g. the user starts moving mouse with pressed button.
  - Preventing the default action makes the selection not start.
- `document.onselectionchange` – whenever a selection changes.
  - Please note: this handler can be set only on `document`.

Selection tracking demo

Here’s a small demo that shows selection boundaries dynamically as it changes:

```
<p id="p">Select me: <i>italic</i> and <b>bold</b></p>

From <input id="from" disabled> – To <input id="to" disabled>
<script>
```

```
  document.onselectionchange = function() {
    let {anchorNode, anchorOffset, focusNode, focusOffset} = document.getSe

    from.value = `${anchorNode && anchorNode.data}:${anchorOffset}`;
    to.value = `${focusNode && focusNode.data}:${focusOffset}`;
  };
</script>
```

Selection getting demo

To get the whole selection:

- As text: just call `document.getSelection().toString()` .
- As DOM nodes: get the underlying ranges and call their `cloneContents()` method (only first range if we don't support Firefox multiselection).

And here's the demo of getting the selection both as text and as DOM nodes:

```
<p id="p">Select me: <i>italic</i> and <b>bold</b></p>

Cloned: <span id="cloned"></span>
<br>
As text: <span id="astext"></span>

<script>
  document.onselectionchange = function() {
    let selection = document.getSelection();

    cloned.innerHTML = astext.innerHTML = "";

    // Clone DOM nodes from ranges (we support multiselect here)
    for (let i = 0; i < selection.rangeCount; i++) {
      cloned.append(selection.getRangeAt(i).cloneContents());
    }

    // Get as text
    astext.innerHTML += selection;
  };
</script>
```

Selection methods

Selection methods to add/remove ranges:

- `getRangeAt(i)` – get i-th range, starting from `0` . In all browsers except firefox, only `0` is used.
- `addRange(range)` – add `range` to selection. All browsers except Firefox ignore the call, if the selection already has an associated range.
- `removeRange(range)` – remove `range` from the selection.
- `removeAllRanges()` – remove all ranges.
- `empty()` – alias to `removeAllRanges` .

Also, there are convenience methods to manipulate the selection range directly, without `Range` :

- `collapse(node, offset)` – replace selected range with a new one that starts and ends at the given `node`, at position `offset`.
- `setPosition(node, offset)` – alias to `collapse`.
- `collapseToStart()` – collapse (replace with an empty range) to selection start,
- `collapseToEnd()` – collapse to selection end,
- `extend(node, offset)` – move focus of the selection to the given `node`, position `offset`,
- `setBaseAndExtent(anchorNode, anchorOffset, focusNode, focusOffset)` – replace selection range with the given start `anchorNode/anchorOffset` and end `focusNode/focusOffset`. All content in-between them is selected.
- `selectAllChildren(node)` – select all children of the `node`.
- `deleteFromDocument()` – remove selected content from the document.
- `containsNode(node, allowPartialContainment = false)` – checks whether the selection contains `node` (partially if the second argument is `true`)

So, for many tasks we can call `Selection` methods, no need to access the underlying `Range` object.

For example, selecting the whole contents of the paragraph `<p>`:

```
<p id="p">Select me: <i>italic</i> and <b>bold</b></p>

<script>
  // select from 0th child of <p> to the last child
  document.getSelection().setBaseAndExtent(p, 0, p, p.childNodes.length);
</script>
```

The same thing using ranges:

```
<p id="p">Select me: <i>italic</i> and <b>bold</b></p>

<script>
  let range = new Range();
  range.selectNodeContents(p); // or selectNode(p) to select the <p> tag too

  document.getSelection().removeAllRanges(); // clear existing selection if a
  document.getSelection().addRange(range);
</script>
```

To select, remove the existing selection first

If the selection already exists, empty it first with `removeAllRanges()`. And then add ranges. Otherwise, all browsers except Firefox ignore new ranges.

The exception is some selection methods, that replace the existing selection, like `setBaseAndExtent`.

Selection in form controls

Form elements, such as `input` and `textarea` provide special API for selection, without `Selection` or `Range` objects. As an input value is a pure text, not HTML, there's no need for such objects, everything's much simpler.

Properties:

- `input.selectionStart` – position of selection start (writeable),
- `input.selectionEnd` – position of selection end (writeable),
- `input.selectionDirection` – selection direction, one of: “forward”, “backward” or “none” (if e.g. selected with a double mouse click),

Events:

- `input.onselect` – triggers when something is selected.

Methods:

- `input.select()` – selects everything in the text control (can be `textarea` instead of `input`),
- `input.setSelectionRange(start, end, [direction])` – change the selection to span from position `start` till `end`, in the given direction (optional).
- `input.setRangeText(replacement, [start], [end], [selectionMode])` – replace a range of text with the new text.

  Optional arguments `start` and `end`, if provided, set the range start and end, otherwise user selection is used.

  The last argument, `selectionMode`, determines how the selection will be set after the text has been replaced. The possible values are:

  - `"select"` – the newly inserted text will be selected.
  - `"start"` – the selection range collapses just before the inserted text (the cursor will be immediately before it).
  - `"end"` – the selection range collapses just after the inserted text (the cursor will be right after it).
  - `"preserve"` – attempts to preserve the selection. This is the default.

Now let’s see these methods in action.

Example: tracking selection

For example, this code uses `onselect` event to track selection:

```html
<textarea id="area" style="width:80%;height:60px">
Selecting in this text updates values below.
</textarea>
<br>
From <input id="from" disabled> – To <input id="to" disabled>

<script>
  area.onselect = function() {
    from.value = area.selectionStart;
    to.value = area.selectionEnd;
  };
</script>
```

```
Selecting in this text updates values below.
```

From ________ – To ________

Please note:

- `onselect` triggers when something is selected, but not when the selection is removed.
- `document.onselectionchange` event should not trigger for selections inside a form control, according to the spec, as it's not related to `document` selection and ranges. Some browsers generate it, but we shouldn't rely on it.

Example: moving cursor

We can change `selectionStart` and `selectionEnd`, that sets the selection.

An important edge case is when `selectionStart` and `selectionEnd` equal each other. Then it's exactly the cursor position. Or, to rephrase, when nothing is selected, the selection is collapsed at the cursor position.

So, by setting `selectionStart` and `selectionEnd` to the same value, we move the cursor.

For example:

```
<textarea id="area" style="width:80%;height:60px">
Focus on me, the cursor will be at position 10.
</textarea>

<script>
  area.onfocus = () => {
    // zero delay setTimeout to run after browser "focus" action finishes
    setTimeout(() => {
      // we can set any selection
      // if start=end, the cursor it exactly at that place
      area.selectionStart = area.selectionEnd = 10;
    });
  };
</script>
```

```
Focus on me, the cursor will be at position 10.
```

Example: modifying selection

To modify the content of the selection, we can use `input.setRangeText()` method. Of course, we can read `selectionStart/End` and, with the knowledge of the selection, change the corresponding substring of `value`, but `setRangeText` is more powerful and often more convenient.

That's a somewhat complex method. In its simplest one-argument form it replaces the user selected range and removes the selection.

For example, here the user selection will be wrapped by `*...*` :

```
<input id="input" style="width:200px" value="Select here and click the button
<button id="button">Wrap selection in stars *...*</button>

<script>
button.onclick = () => {
  if (input.selectionStart == input.selectionEnd) {
    return; // nothing is selected
  }

  let selected = input.value.slice(input.selectionStart, input.selectionEnd);
  input.setRangeText(`*${selected}*`);
};
</script>
```

Select here and click the button [Wrap selection in stars *...*]

With more arguments, we can set range `start` and `end` .

In this example we find `"THIS"` in the input text, replace it and keep the replacement selected:

```
<input id="input" style="width:200px" value="Replace THIS in text">
<button id="button">Replace THIS</button>

<script>
button.onclick = () => {
  let pos = input.value.indexOf("THIS");
  if (pos >= 0) {
    input.setRangeText("*THIS*", pos, pos + 4, "select");
    input.focus(); // focus to make selection visible
  }
};
</script>
```

Replace THIS in text [Replace THIS]

Example: insert at cursor

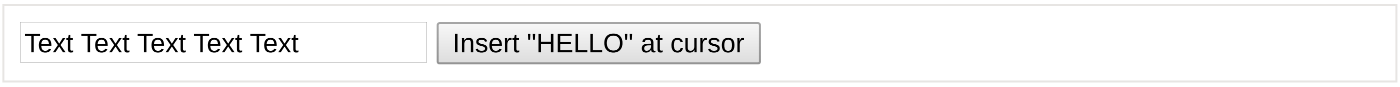
If nothing is selected, or we use equal `start` and `end` in `setRangeText` , then the new text is just inserted, nothing is removed.

We can also insert something "at the cursor" using `setRangeText` .

Here's a button that inserts `"HELLO"` at the cursor position and puts the cursor immediately after it. If the selection is not empty, then it gets replaced (we can detect it by comparing `selectionStart!=selectionEnd` and do something else instead):

```
<input id="input" style="width:200px" value="Text Text Text Text Text">
<button id="button">Insert "HELLO" at cursor</button>

<script>
```

```
  button.onclick = () => {
    input.setRangeText("HELLO", input.selectionStart, input.selectionEnd, "en
    input.focus();
  };
</script>
```


Text Text Text Text Text

Making unselectable

To make something unselectable, there are three ways:

1. Use CSS property `user-select: none`.

```
<style>
#elem {
  user-select: none;
}
</style>
<div>Selectable <div id="elem">Unselectable</div> Selectable</div>
```

   This doesn't allow the selection to start at `elem`. But the user may start the selection elsewhere and include `elem` into it.

   Then `elem` will become a part of `document.getSelection()`, so the selection actually happens, but its content is usually ignored in copy-paste.

2. Prevent default action in `onselectstart` or `mousedown` events.

```
<div>Selectable <div id="elem">Unselectable</div> Selectable</div>

<script>
  elem.onselectstart = () => false;
</script>
```

   This prevents starting the selection on `elem`, but the visitor may start it at another element, then extend to `elem`.

   That's convenient when there's another event handler on the same action that triggers the select (e.g. `mousedown`). So we disable the selection to avoid conflict, still allowing `elem` contents to be copied.

3. We can also clear the selection post-factum after it happens with `document.getSelection().empty()`. That's rarely used, as this causes unwanted blinking as the selection appears-disappears.

References

- DOM spec: Range
- Selection API
- HTML spec: APIs for the text control selections

Summary

We covered two different APIs for selections:

1. For document: `Selection` and `Range` objects.
2. For `input`, `textarea`: additional methods and properties.

The second API is very simple, as it works with text.

The most used recipes are probably:

1. Getting the selection:

```
let selection = document.getSelection();

let cloned = /* element to clone the selected nodes to */;

// then apply Range methods to selection.getRangeAt(0)
// or, like here, to all ranges to support multi-select
for (let i = 0; i < selection.rangeCount; i++) {
  cloned.append(selection.getRangeAt(i).cloneContents());
}
```

2. Setting the selection:

```
let selection = document.getSelection();

// directly:
selection.setBaseAndExtent(...from...to...);

// or we can create a range and:
selection.removeAllRanges();
selection.addRange(range);
```

And finally, about the cursor. The cursor position in editable elements, like `<textarea>` is always at the start or the end of the selection. We can use it to get cursor position or to move the cursor by setting `elem.selectionStart` and `elem.selectionEnd`.

Previous lesson Next lesson

Share Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.

- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)