



UNIVERSITÀ DEGLI STUDI DI CAGLIARI

CORSO DI LAUREA IN INFORMATICA

Silvia Maria Massa – silviam.massa@unica.it

Fondamenti di Programmazione Web

Client (2)





Un piccolo ripasso

Document Object Model

- Il **DOM** è una rappresentazione ad oggetti (Javascript) della struttura ad albero della pagina HTML
- Questa rappresentazione permette di accedere e aggiornare dinamicamente il contenuto, la struttura e lo stile dei documenti attraverso il linguaggio javascript.
- Javascript ha un oggetto speciale, **document**, che rappresenta il documento correntemente visualizzato e ci permette di accedere al DOM.

Pagina esempio

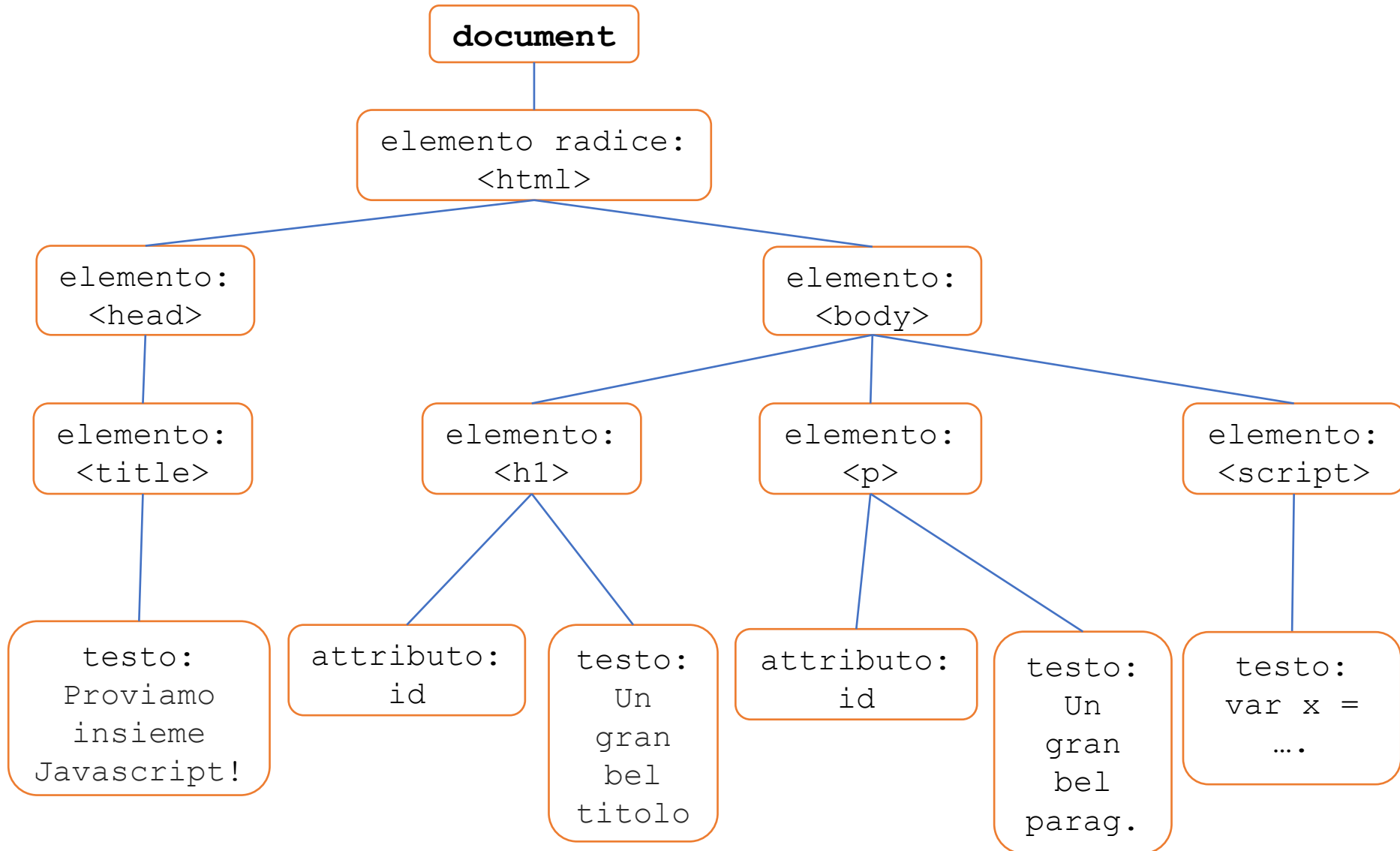
```
<!DOCTYPE html>
<html>
<head>
  <title> Proviamo insieme Javascript! </title>
</head>
<body>

  <h1 id="titolo"> Un gran bel titolo </h1>
  <p id="paragrafo"> Un gran bel paragrafo </p>

  <script>
    var x = document.getElementById("paragrafo").innerText;
  </script>

</body>
</html>
```

Struttura ad albero



Ricerca nel document – manipolazione manuale

- Per identificatore unico

```
function modificaTitolo() {  
    // ricerchiamo l'elemento con id titolo (l'id è univoco)  
    var titolo = document.getElementById("titolo");  
    if(titolo != undefined) {  
        // abbiamo trovato l'elemento  
        // cerchiamo l'elemento testuale  
        for(var i in titolo.childNodes) {  
            var child = titolo.childNodes[i];  
            if(child.nodeType == Node.TEXT_NODE) {  
                // abbiamo trovato il nodo di testo  
                // modifichiamo il valore  
                child.nodeValue = "Io arrivo da Javascript"  
            }  
        }  
    }  
}
```

Ricerca nel document – manipolazione manuale

- Per nome del tag

```
function modificaParagrafo(){
    // attenzione questo restituisce TUTTI i paragrafi
    // nel documento (il tag NON è univoco)
    var paragrafi = document.getElementsByTagName("p") ;
    for(var i in paragrafi){
        var paragrafo = paragrafi[i];
        // cerchiamo l'elemento testuale
        for(var j in paragrafo.childNodes){
            var child = paragrafo.childNodes[j];
            if(child.nodeType == Node.TEXT_NODE){
                // abbiamo trovato il nodo di testo
                // modifichiamo il valore
                child.nodeValue = "Anche io arrivo da Javascript"
            }
        }
    }
}
```

Aggiunta/eliminazione nodo – manipolazione manuale

```
function aggiungiBottone() {  
    // creiamo un nodo di tipo button  
    var button = document.createElement("button");  
  
    // creiamo un nodo testuale  
    var txt = document.createTextNode("Clicca qui");  
  
    // aggiungiamo il nodo testuale al bottone  
    button.appendChild(txt);  
  
    // ora aggiungiamo il bottone al body  
    document.getElementsByTagName("body")[0].appendChild(button);  
}
```

- In modo molto simile si può eliminare un nodo con la **removeNode()**

Modifica del CSS - manipolazione manuale

- Ogni nodo di tipo elemento ha una proprietà **style**
 - questa rappresenta gli stili inline definiti direttamente nell'attributo **style** dell'elemento
 - `<p style="color: red; font-size: 16px;">Testo rosso</p>`
 - Esempio: modifica del colore del testo di un elemento con `id="titolo"`
`document.getElementById('titolo').style.color = 'green';`
- Oltre a modificare direttamente gli stili, è possibile assegnare o modificare le classi CSS di un elemento tramite la proprietà **className**
 - questa rappresenta esattamente il contenuto dell'attributo **class** nell'HTML
 - `<p class="evidenziato">Questo paragrafo ha una classe</p>`
 - Esempio: aggiunta di una classe CSS chiamata `redColor`
`document.getElementById('titolo').className += ' redColor';`
 - attenzione agli spazi quando aggiungete la classe (ricordatevi che un elemento può avere più classi separate da spazi)



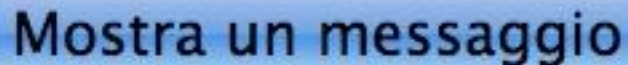
Eventi

Gli eventi del documento

- Abbiamo visto come modificare il DOM, ma come facciamo a sapere **quando** dobbiamo effettuare le modifiche?
- Ci aspettiamo, per esempio, di modificare qualcosa nel documento se l'utente clicca su un bottone o passa sopra un elemento con il mouse
- Questi sono **eventi** che sono gestibili tramite delle funzioni Javascript.

Observer pattern

- Abbiamo un "**soggetto**" che tiene traccia di una lista di osservatori.
- Questi "**osservatori**" sono interessati a essere notificati quando si verifica un determinato evento (es. clic di un pulsante).
- Ogni osservatore implementa un metodo **handler**, ovvero del codice JavaScript da eseguire quando l'evento si verifica.



Mostra un messaggio

Hai cliccato un bottone!

```
document.getElementById("par").innerText  
    = "Hai cliccato un bottone!"
```

Observer pattern

- Il soggetto espone dei metodi per:
 - registrare nuovi osservatori,
 - rimuovere osservatori esistenti,
 - notificare tutti gli osservatori registrati tramite il metodo **notify()**.
- Quando l'evento si verifica (es. un utente clicca un bottone), il soggetto invoca automaticamente il metodo **update()** (o simile) di ogni osservatore registrato.
- Soggetto e osservatori sono completamente disaccoppiati:
 - il soggetto non conosce i dettagli interni degli osservatori,
 - gli osservatori non conoscono come funziona internamente il soggetto.

Come agganciare gli handler

- Si può associare un **handler** a un determinato evento specificando una funzione JavaScript che se ne occupi.
- Questa funzione può essere collegata a un elemento HTML in due modi:
 - utilizzando un attributo evento, come onclick, direttamente nel codice HTML,
 - tramite JavaScript, usando proprietà o metodi, come .onclick
- Quando l'evento si verifica, l'interprete JavaScript del browser esegue automaticamente la funzione handler.
- Ogni volta che si verifica un evento, alla funzione viene passato un **event object** che contiene informazioni dettagliate sull'evento stesso.
- Ad esempio:
 - per eventi del mouse, l'oggetto include la posizione sullo schermo, il pulsante premuto, ecc.;
 - per eventi della tastiera, include il tasto premuto, eventuali tasti modificatori (Shift, Alt, Ctrl), ecc.

Agganciare un handler: esempio

- Es: event handler che conta i click su un elemento e li mostra all'interno di un elemento HTML

```
function addClick(event) {  
    //clickCount è una variabile globale  
    clickCount++;  
    event.currentTarget.innerHTML = 'Click: ' + clickCount;  
}
```

- Modi di assegnare un handler di eventi a un elemento HTML:

1. aggiunta direttamente nell'HTML

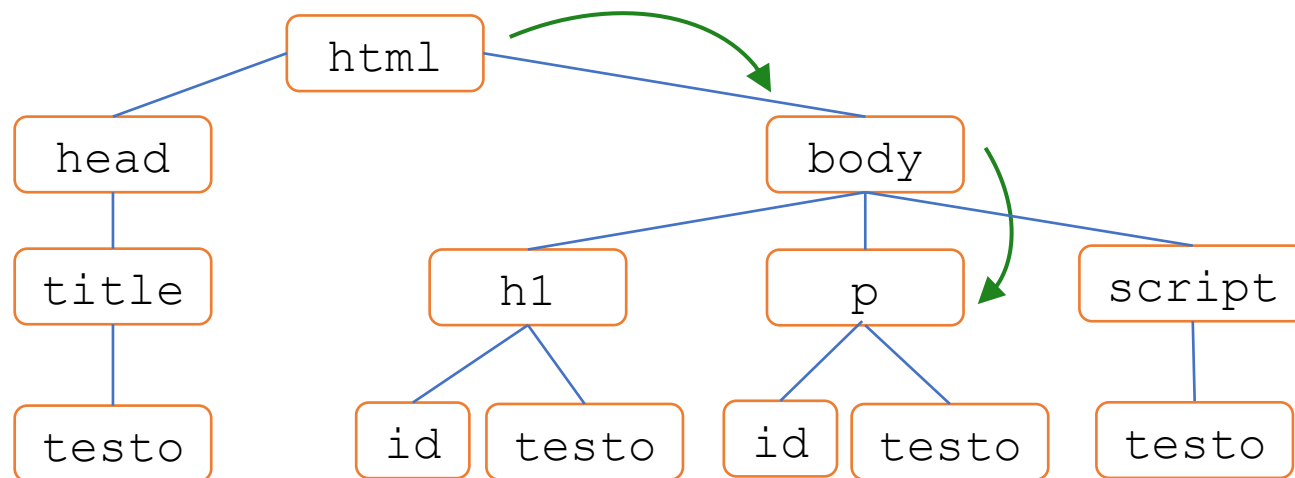
```
<p id="paragrafo" onclick="addClick(event)">
```

2. aggiunta tramite Javascript

```
document.getElementById('paragrafo').onclick =  
addClick;
```

Event tunnelling (click)

- Quando si clicca su un elemento nella pagina web, il browser segue un flusso gerarchico di propagazione dell'evento, attraverso la rappresentazione ad albero della pagina web, fino a raggiungere l'elemento più interno che ha innescato l'evento.

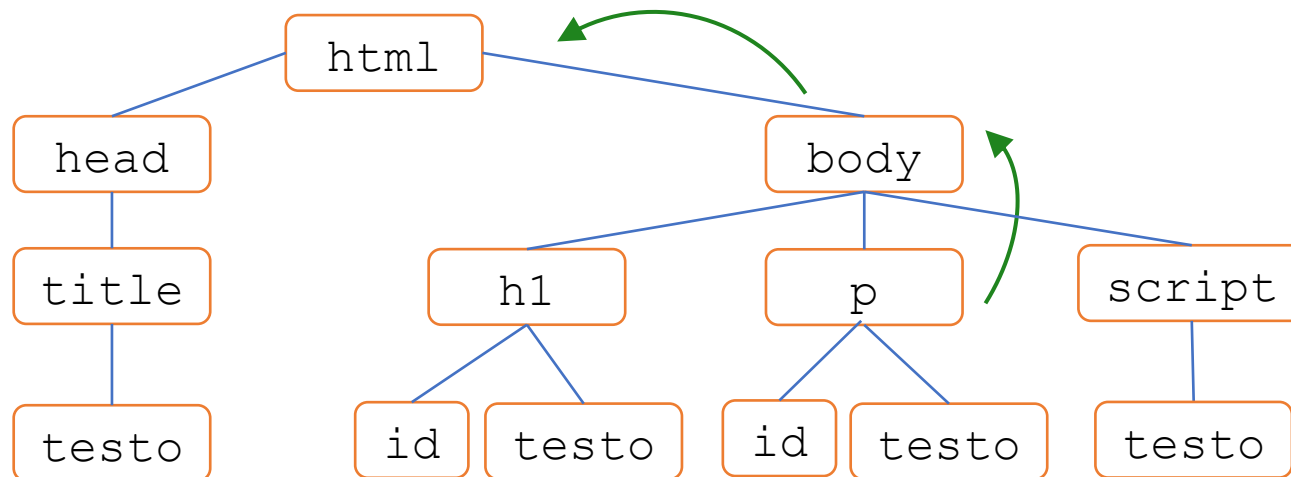


Questo è un paragrafo



Event bubbling (click)

- Dopo di che l'evento "risale" la gerarchia fino al nodo più esterno cercando qualcuno che lo gestisca.
- Di default viene richiamato ogni handler che si trova associati a quel tipo di evento.
- Il bubbling è attivo a meno che non venga interrotto utilizzando **`event.stopPropagation()`**.



Questo è un paragrafo

Event target e currentTarget

- **event.target** contiene l'elemento effettivo su cui è avvenuto l'evento, ovvero il nodo più profondo nell'albero DOM che ha generato l'evento.
- **event.currentTarget** contiene invece l'elemento che sta gestendo l'evento (cioè che ha associato l'handler in corso di esecuzione).
- **event.currentTarget** ed **event.target** possono contenere due valori differenti quando l'evento viene gestito tramite bubbling.
- Se **event.target** e **event.currentTarget** hanno lo stesso valore l'evento è gestito direttamente sull'elemento che ha generato l'evento.

Controllare l'event object

- **event.preventDefault()** fa in modo che le azioni che il browser fa di default non vengano eseguite (es: il submit di un form quando si clicca su un bottone di submit)
- A volte è necessario controllare anche il bubbling degli eventi
- Per esempio se gestiamo il click su un elemento interno e sappiamo che ci può essere anche un handler in un elemento più in alto nella gerarchia, può essere utile bloccare il bubbling per evitarne l'esecuzione
- Questo comportamento può essere ottenuto invocando **event.stopPropagation()** per interrompere la propagazione

Funzioni e proprietà DOM

- Abbiamo detto che ogni elemento della pagina HTML è un nodo.
- Ognuno di questi nodi ha delle caratteristiche particolari in base al tipo di elemento (esempio un `<div>`, un `<button>`, un `<form>`).
- Queste caratteristiche sono:
 - proprietà (stato e aspetto es. `innerHTML`, `disabled`)
 - eventi (azioni a cui l'elemento può reagire es. `click`, `mouseover`)
 - funzioni di controllo specifiche (per controllare o modificare il comportamento del nodo es. `submit()`)
- Sarebbe troppo lungo elencarle tutte per ogni elemento
 - nelle slide successive trovate le più usate



Proprietà e metodi del form

Form Object Properties

Property	Description
<u>acceptCharset</u>	Sets or returns the value of the accept-charset attribute in a form
<u>action</u>	Sets or returns the value of the action attribute in a form
<u>enctype</u>	Sets or returns the value of the enctype attribute in a form
<u>length</u>	Returns the number of elements in a form
<u>method</u>	Sets or returns the value of the method attribute in a form
<u>name</u>	Sets or returns the value of the name attribute in a form
<u>target</u>	Sets or returns the value of the target attribute in a form

Form Object Methods

Method	Description
<u>reset()</u>	Resets a form
<u>submit()</u>	Submits a form

Eventi dei form/input

Form Events

Attribute	Description
<u>onblur</u>	The event occurs when a form element loses focus
<u>onchange</u>	The event occurs when the content of a form element, the selection, or the checked state have changed (for <code><input></code> , <code><select></code> , and <code><textarea></code>)
<u>onfocus</u>	The event occurs when an element gets focus (for <code><label></code> , <code><input></code> , <code><select></code> , <code>textarea></code> , and <code><button></code>)
onreset	The event occurs when a form is reset
<u>onselect</u>	The event occurs when a user selects some text (for <code><input></code> and <code><textarea></code>)
onsubmit	The event occurs when a form is submitted



Eventi del mouse

Mouse Events

Property	Description
<u>onclick</u>	The event occurs when the user clicks on an element
<u>ondblclick</u>	The event occurs when the user double-clicks on an element
<u>onmousedown</u>	The event occurs when a user presses a mouse button over an element
<u>onmousemove</u>	The event occurs when the pointer is moving while it is over an element
<u>onmouseover</u>	The event occurs when the pointer is moved onto an element
<u>onmouseout</u>	The event occurs when a user moves the mouse pointer out of an element
<u>onmouseup</u>	The event occurs when a user releases a mouse button over an element



Eventi della tastiera

Keyboard Events

Attribute	Description
<u>onkeydown</u>	The event occurs when the user is pressing a key
<u>onkeypress</u>	The event occurs when the user presses a key
<u>onkeyup</u>	The event occurs when the user releases a key

Eventi del documento/immagini

Frame/Object Events

Attribute	Description
onabort	The event occurs when an image is stopped from loading before completely loaded (for <object>)
onerror	The event occurs when an image does not load properly (for <object>, <body> and <frameset>)
<u>onload</u>	The event occurs when a document, frameset, or <object> has been loaded
<u>onresize</u>	The event occurs when a document view is resized
onscroll	The event occurs when a document view is scrolled
<u>onunload</u>	The event occurs once a page has unloaded (for <body> and <frameset>)

Proprietà Event Object

Property	Description
<u>altKey</u>	Returns whether or not the "ALT" key was pressed when an event was triggered
<u>button</u>	Returns which mouse button was clicked when an event was triggered
<u>clientX</u>	Returns the horizontal coordinate of the mouse pointer, relative to the current window, when an event was triggered
<u>clientY</u>	Returns the vertical coordinate of the mouse pointer, relative to the current window, when an event was triggered
<u>ctrlKey</u>	Returns whether or not the "CTRL" key was pressed when an event was triggered
keyIdentifier	Returns the identifier of a key
keyLocation	Returns the location of the key on the advice
<u>metaKey</u>	Returns whether or not the "meta" key was pressed when an event was triggered
<u>relatedTarget</u>	Returns the element related to the element that triggered the event
<u>screenX</u>	Returns the horizontal coordinate of the mouse pointer, relative to the screen, when an event was triggered
<u>screenY</u>	Returns the vertical coordinate of the mouse pointer, relative to the screen, when an event was triggered
<u>shiftKey</u>	Returns whether or not the "SHIFT" key was pressed when an event was triggered



Vue.js

Vue.js SFC

- Un'applicazione Vue.js è suddivisa in diversi componenti
- Ogni componente è racchiuso in un file *.vue differente
- I componenti di Vue vengono creati utilizzando un formato di file simile all'HTML, chiamato Single-File Component (SFC).
- Gli SFC sono costituiti da tre elementi:
 - <template> HTML </template>
 - <script> JavaScript </script>
 - <style> CSS </style>
- Template è obbligatorio, mentre script e style sono opzionali
- I componenti Vue possono essere creati utilizzando due diversi stili di API:
 - **Options API** (che utilizzeremo maggiormente)
 - **Composition API**

Options API

```
<script>
```

```
export default {
```

```
  data() {
```

```
    return {
```

```
      count: 0
```

```
    },
```

```
  },
```

```
  methods: {
```

```
    increment() {
```

```
      this.count++
```

```
    },
```

```
  },
```

```
  mounted() {
```

```
    console.log(`The initial count is ${this.count}.`)
```

```
  }
```

```
}
```

```
</script>
```

Si definisce la logica di un componente utilizzando un oggetto contenente diverse proprietà.

Le proprietà restituite da `data()` diventano parte dello stato reattivo e saranno accessibili tramite `this``.

I metodi sono funzioni che modificano lo stato e attivano gli aggiornamenti. Possono essere usati come gestori di eventi nei template (tramite `v-on` o `@`).

I metodi del ciclo di vita sono chiamati in diverse fasi del ciclo di vita di un componente. `Mounted()` viene chiamata quando il componente viene montato.

```
<template>
```

```
  <button @click="increment">Count is: {{ count }}</button>
```

```
</template>
```

Composition API

- **Vue 3** offre un'API alternativa per la dichiarazione dei componenti: la **Composition API**

```
<script>
export default {
  data () {
    return {
      todos: [],
      newTodo: ""
    }
  },
  computed: {
    hasNoLabel(){
      return this.newTodo.trim() === ""
    }
  },
  methods: {
    addTodo(){
      this.todos.push({ label: this.newTodo, done: false })
      this.newTodo = ""
    }
  }
}
</script>
```



```
<script setup>
import { ref, reactive, computed } from "vue"

const todos = reactive([])
const newTodo = ref("")
const hasNoLabel = computed(() => newTodo.value.trim() === "")

function addTodo(){
  todos.push({ label: newTodo.value, done: false })
  newTodo.value = ""
}
</script>
```

Composition API

I principali cambiamenti rispetto al Options API che possiamo notare sono:

- Con **<script setup>**, data, computed, methods e altre opzioni possono essere dichiarate liberamente, senza seguire un determinato ordine o suddivisione per tipo (es. metodi raggruppati in methods).
- Non si utilizza più **this**: i dati e i metodi sono variabili indipendenti, definite tramite funzioni di Vue (non stiamo più utilizzando un oggetto).

Esempi:

computed() per creare variabili calcolate

defineProps() per dichiarare le proprietà ricevute

defineEmits() per definire gli eventi emessi

- Per la reattività (che non risulta più implicita):
 - ref()** è usato per valori primitivi (stringhe, numeri, ecc.)
 - reactive()** è usato per oggetti complessi

Vue.js Interpolazione del testo nei template

- Il modo più semplice per inserire dati dinamicamente nei componenti è l'interpolazione del testo: `{{myVariable}}`
- All'interno delle doppie parentesi graffe, è possibile specificare qualsiasi espressione JavaScript valida

```
<template>
  <p>Order ref. {{ orderReference }} - Total: {{ Math.round(price) + "€" }}</p>
</template>

<script>
  export default {
    data() {
      return {
        orderReference: "ABCXYZ",
        price: 17.3,
      };
    },
  };
</script>
```

- L'interpolazione funziona solo sul contenuto testuale degli elementi
~~``~~

Vue.js Direttive

- Le direttive sono gli elementi di sintassi specifici di Vue che possono essere usati nel template dei componenti
- Nelle prossime slide vedremo:
 - v-bind
 - v-model
 - v-if e v-show
 - v-for
 - v-on

Vue.js Direttiva v-bind sulle proprietà

- **v-bind** consente di legare al valore di una proprietà di un elemento o componente HTML un'espressione
- Poiché questa è la direttiva più comunemente usata, di solito si usa la sintassi abbreviata **:property="valore"**

`<a v-bind:href="url">Link`

oppure

`<!-- shortened syntax -->`

`<a :href="url">Link`

```
<script>
export default {
  data(){
    return {
      url: 'https://www.google.com'
    }
  },
  .....
}
</script>
```

Vue.js Direttiva v-bind sulle proprietà

Esercizio: collegate gli attributi src e width dell'immagine

``

```
<template>
<h1>
  I {{likesVue ? "love" : "hate"}}
  <img src="" />
</h1>
</template>

<script>
export default {
  data(){
    return {
      likesVue: true,
      logo: 'https://vuejs.org/images/logo.png',
      logoWidth: 50
    }
  }
}
</script>
```

I love

Vue.js Direttiva v-bind sulle proprietà

```
<template>
<h1>
  I {{likesVue ? "love" : "hate"}}
  
</h1>
</template>

<script>
export default {
  data(){
    return {
      likesVue: true,
      logo: 'https://vuejs.org/images/logo.png',
      logoWidth: 50
    }
  }
}
</script>
```

I love 

Vue.js Direttiva v-bind su class

- Sono disponibili diverse sintassi per assegnare classi

`<p :class="classAsString"></p>`

`<p :class "classAsObject"></p>`

```
<script>
export default {
  data(){
    return {
      classAsString: "foo bar",
      classAsObject: { foo: true, bar: isBar }
    }
  },
  .....
}
```

`</script>`

- Molto utile quando vogliamo cambiare lo stile di un elemento HTML a **seconda di certe condizioni**, come l'attivazione di un bottone, la presenza di un errore, o il passaggio del mouse

Vue.js Direttiva v-model per form e input

- **v-model** consente di legare il valore di un campo di un form a un elemento di `data()` del componente
- Si tratta di un legame bidirezionale: la variabile viene aggiornata quando il contenuto del campo cambia (tipicamente dall'utente) e viceversa

What is your name ?

Hello your name !

What is your name ?

Hello Silvia !

```
<label>
  What is your name ?
  <input v-model="name">
</label>

<p>Hello {{ name }} !</p>
```

```
<script>
export default {
  data(){
    return {
      name: "your name",
    }
  },
  .....,
}
</script>
```

Vue.js Direttiva v-model per form e input

Esercizio: utilizzo del v-model su input, select, radio e checkbox

```
<template>
<div id="icecream-store">
  <h1>Icecream store</h1>

  <label>Quantity: <input type="number">
</label>

  <label>Size:
    <select>
      <option value="100">Small</option>
      <option value="150">Medium</option>
      <option value="200">Giant</option>
    </select>
  </label>

  <label>Flavour:</label>
  <label><input type="radio" name="flavour"
value="#5B2F00" />Chocolate</label>
  <label><input type="radio" name="flavour"
value="#DE0934" />Strawberry</label>

  <label><input type="checkbox">
Napkin</label>

  <IceCreams :quantity="quantity"
:flavour="flavour" :size="size"
:napkin="napkin" />
</div>
</template>
```

```
<script>
import IceCreams from "./IceCreams.vue";

export default {
  components: { IceCreams },
  data() {
    return {
      quantity: 1,
      flavour: "#5B2F00",
      size: 150,
      napkin: true
    }
  }
}
</script>
```

Icecream store

Quantity:

Size:

Flavour:

☐ Chocolate

☐ Strawberry

☐ Napkin



Vue.js Direttiva v-model per form e input

```
<template>
<div id="icecream-store">
  <h1>Icecream store</h1>

  <label>Quantity: <input v-model="quantity"
type="number"></label>

  <label>Size:
    <select v-model="size">
      <option value="100">Small</option>
      <option value="150">Medium</option>
      <option value="200">Giant</option>
    </select>
  </label>

  <label>Flavour:</label>
  <label><input v-model = "flavour"
type="radio" name="flavour"
value="#5B2F00" />Chocolate</label>
  <label><input v-model = "flavour"
type="radio" name="flavour"
value="#DE0934" />Strawberry</label>

  <label><input v-model="napkin"
type="checkbox"> Napkin</label>

  <IceCreams :quantity="quantity"
:flavour="flavour" :size="size"
:napkin="napkin" />
</div>
</template>
```

```
<script>
import IceCreams from "../IceCreams.vue";

export default {
  components: { IceCreams },
  data() {
    return {
      quantity: 1,
      flavour: "#5B2F00",
      size: 150,
      napkin: true
    }
  }
}
</script>
```

Icecream store

Quantity:

Size: Giant ▼

Flavour:

☐ Chocolate

☒ Strawberry

☐ Napkin



Vue.js Direttiva v-if per le condizioni

- Permette di inserire o meno un elemento in base a una condizione
- Se si desidera che l'elemento non venga rimosso dal DOM, ma solo nascosto visivamente, si utilizza invece **v-show**
- Le direttive **v-else-if** e **v-else** funzionano allo stesso modo del loro equivalente in JavaScript e dipendono dalla condizione **v-if** dell'elemento che le precede.

```
<div v-if="type === 'A'"> A </div>  
<div v-else-if="type === 'B'"> B </div>  
<div v-else> Not A nor B </div>
```

```
<script>  
export default {  
  data(){  
    return {  
      type: "A",  
    }  
  },  
  .....  
}
```

Vue.js Direttiva v-if per le condizioni

Esercizio: utilizzare v-if, v-else e v-else-if per alternare le facce in base alla temperatura

```
<template>
<div>
  <input type="range" min="0" max="40" v-
model="temperature" />
  {{ temperature }} °C
  <span>😓</span>
  <span>😬</span>
  <span>😄</span>
</div>
</template>

<script>
export default {
  data(){
    return { temperature: 20 }
  }
}
</script>
```

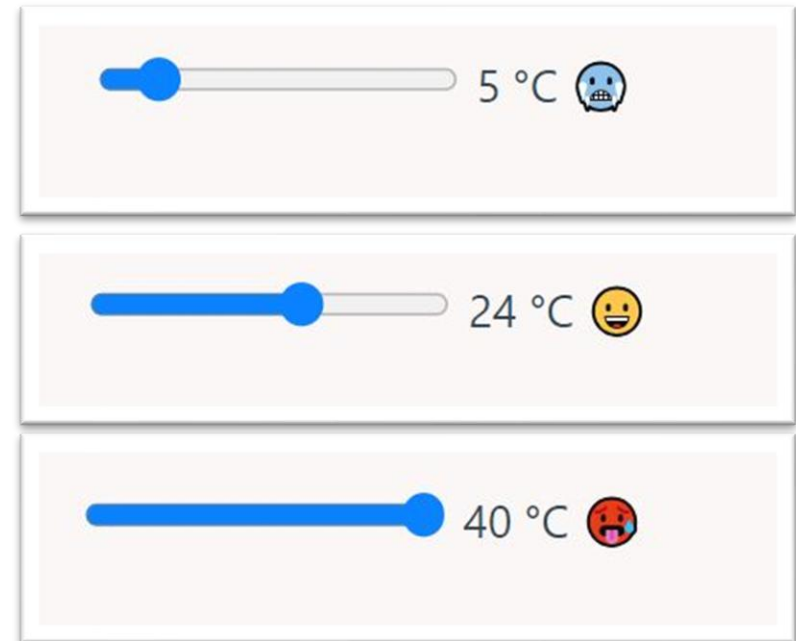
vue



Vue.js Direttiva v-if per le condizioni

```
<template>
<div>
  <input type="range" min="0" max="40" v-
model="temperature" />
  {{ temperature }} °C
  <span v-if="temperature >= 35">🔥</span>
  <span v-else-if="temperature < 15">❄️
</span>
  <span v-else>😊</span>
</div>
</template>
```

```
<script>
export default {
  data() {
    return { temperature: 20 }
  }
}
</script>
```



Vue.js Direttiva v-for per i loop

- Genera elenchi di elementi ripetendo un template.
- **v-for** esegue un ciclo su un valore iterabile:
 - un array,
 - le proprietà di un oggetto,
 - un numero fisso di iterazioni.
- La direttiva v-for dichiara variabili locali che rappresentano l'**elemento corrente** e il suo **indice**, utilizzabili all'interno del template.
- Per ripetere un gruppo di elementi o componenti, si può usare v-for direttamente sul **tag contenitore** o sul **componente**.

```
1 ; 2 ; 3 ; 4 ; 5 ; 6 ; 7 ; 8 ; 9 ; 10 ;  
•apple  
•kiwi  
•mango
```

```
<span v-for="n in 10"> {{ n }} ; </span>
```

```
<ul> <li v-for="item in items">{{ item }}</li> </ul>
```

```
<script>  
export default {  
  data(){  
    return { items: ["apple","kiwi","mango"], }  
  },  
}  
</script>
```

Vue.js Direttiva v-for per i loop

- Oltre alla direttiva **v-for**, si dovrebbe associare una proprietà **key** a un valore che identifichi in modo univoco ciascun elemento dell'elenco
- Questo non è obbligatorio ma aiuta Vue a comprendere meglio i cambiamenti che si verificano in una lista (aggiunte, eliminazioni, ordinamenti, ecc.) e a ottimizzare le transizioni tra due stati della lista.
- Supponiamo di avere una lista di attività che cambia ordine man mano che vengono spuntate. Tutte le attività terminate vengono messe alla fine.

- ☐ Task 0 : Passare la teoria-in progress...
- ☐ Task 1 : Consegnare il progetto-in progress...
- ☒ Task 2 : Seguire le lezioni-DONE !



- ☐ Task 0 : Consegnare il progetto-in progress...
- ☒ Task 1 : Passare la Teoria-DONE !
- ☒ Task 2 : Seguire le lezioni-DONE !

Vue.js Direttiva v-for per i loop

```
<ul>
```

```
<!-- the list is ordered by putting completed tasks at the end -->
```

```
<li v-for="(todo, index) in todos_after_sort" :key="todo.label">
```

```
<label>
```

```
<input type="checkbox" v-model="todo.done">
```

```
Task {{ index }}: {{todo.label}}
```

```
</label>
```

```
{{todo.done ? "DONE !" : "in progress..."}} </li>
```

```
</ul>
```

```
<script>
export default {
  data(){
    return {todos: [{ label: 'Seguire le lezioni', done: true },
                     { label: 'Passare la teoria', done: false },
                     { label: 'Consegnare il progetto', done: false } ]},
    computed: {
      todos_after_sort() { return this.todos.slice().sort((a, b) => a.done - b.done); }
    }
  }
}</script>
```

Vue.js Direttiva v-for per i loop

Esercizio: utilizzare due cicli v-for per visualizzare tutto il contenuto del carrello (tenere conto di type e quantity)

```
<template>
<div id="basket">
  <h1>In my basket:</h1>
  <ul>
    <li>
      <span>🍏 </span>
      <span>🍏 </span>
    </li>
  </ul>
</div>
</template>
```

```
<script>
export default {
  data() {
    return {
      basket: [
        { type: '🍏', quantity: 4 },
        { type: '🍒', quantity: 6 },
        { type: '🍉', quantity: 1 },
      ]
    }
  }
}
</script>
```

In my basket:

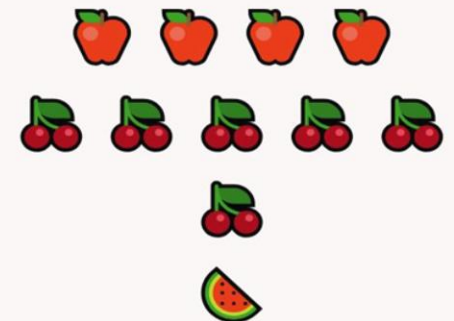


Vue.js Direttiva v-for per i loop

```
<template>
<div id="basket">
  <h1>In my basket:</h1>
  <ul>
    <li v-for="item in basket" :key =
      "item.type">
      <span v-for= "n in item.quantity">
        {{item.type}}
      </span>
    </li>
  </ul>
</div>
</template>
```

```
<script>
export default {
  data() {
    return {
      basket: [
        { type: '🍏', quantity: 4 },
        { type: '🍒', quantity: 6 },
        { type: '🍉', quantity: 1 },
      ]
    }
  }
}
</script>
```

In my basket:



Vue.js Direttiva v-on per gestire gli eventi

- **v-on** ci permette di definire un'azione da eseguire quando si verifica un evento.
- Può essere un evento come clic, mouseover, focus, ecc. o un evento personalizzato emesso da un componente figlio.
- Si può utilizzare la sintassi abbreviata **@event**.
Nell'esempio sotto sarebbe @click.

<button v-on:click="counter += 1"> Click here! </button>

This button has been clicked {{ counter }} times.

Click here! This button has been clicked 0 times.

Click here! This button has been clicked 1 times.

```
<script>
export default {
  data(){
    return {
      counter: 0,
    }
  },
}
</script>
```

Vue.js Componenti - Methods

- I metodi del componente sono dichiarati nella proprietà **methods**.
- Possono essere richiamati da un'espressione nel template o da un altro metodo del componente con **this.myMethod()**.

```
<script>
export default {
  data() {
    return { name: "Mark" };
  },
  methods: {
    greet() {
      this.say("Hi " + this.name); // 'this' si riferisce all'istanza della vista
    },
    say(message) {
      // Se un metodo è indipendente dall'istanza (nessun riferimento a 'this'),
      // potrebbe essere rilevante spostarlo in un modulo separato.
      alert(message + "!");
    }
  }
};
</script>
```

```
<template>
  <button @click="greet">Greet</button>
</template>
```

Vue.js Direttiva v-on per gestire gli eventi

Esercizio: utilizzare gli eventi per aggiungere una scimmia quando si fa clic sul pulsante (**click**) e farle aprire gli occhi al passaggio del mouse (**mouseover**, **mouseleave**).

```
<template>
<div>
  <span v-for="monkey in
monkeys">
    {{ monkey.hasEyesOpen ?
🐵 : 🐵 }}
  </span>
  <br/>
  <button>Add
monkey</button>
</div>
</template>
```

```
<script>
export default {
  data(){
    return {
      monkeys: [
        { hasEyesOpen: false }
      ]
    },
    methods: {
      addMonkey(){
        this.monkeys.push({
          hasEyesOpen: false })
      }
    }
  }
}</script>
```



Add monkey

Vue.js Direttiva v-on per gestire gli eventi

```
<template>
<div>
  <span v-for="monkey in monkeys"
    @mouseover="monkey.hasEyesOpen=true"
    @mouseleave="monkey.hasEyesOpen=false"
  >
    {{ monkey.hasEyesOpen ? '🐵' : '🐶' }}
  </span>
  <br/>
  <button v-on:click='addMonkey'>Add
  monkey</button>
</div>
</template>
```

```
<script>
export default {
  data(){
    return {
      monkeys: [
        { hasEyesOpen: false }
      ]
    },
  },
  methods: {
    addMonkey(){
      this.monkeys.push({ hasEyesOpen: false })
    }
  }
}
</script>
```



Add monkey

Vue.js Direttiva v-on - Modificatori

- È molto comune la necessità di chiamare **event.preventDefault()** o **event.stopPropagation()** all'interno dei gestori di eventi.
- Anche se possiamo farlo facilmente all'interno dei metodi, sarebbe meglio se i metodi potessero riguardare esclusivamente la logica dei dati, invece di dover gestire i dettagli dell'evento DOM.
- Per risolvere questo problema, Vue fornisce modificatori di eventi per v-on.
- I modificatori sono prefissi direttivi denotati da un punto.
- **.stop** - interrompe la propagazione dell'evento
- **.prevent** - impedisce che si verifichi il comportamento predefinito dell'evento
- **.self** - attiva il gestore solo se l'evento è stato inviato dall'elemento stesso
- **.capture** - ascolta l'evento nella fase di tunnelling invece che in quella di bubbling
- **.once** - assicura che l'ascoltatore di eventi sia invocato solo una volta
- **.passive** - il gestore dell'evento non richiamerà mai preventDefault()

```
<!-- the propagation of the click event will be stopped -->
<a @click.stop="onThis">...</a>
<!-- submitting the form will not reload the page -->
<form @submit.prevent="onSubmit">...</form>
<!-- modifiers can be chained -->
<a @click.stop.once="doSomethingOnce">...</a>
```