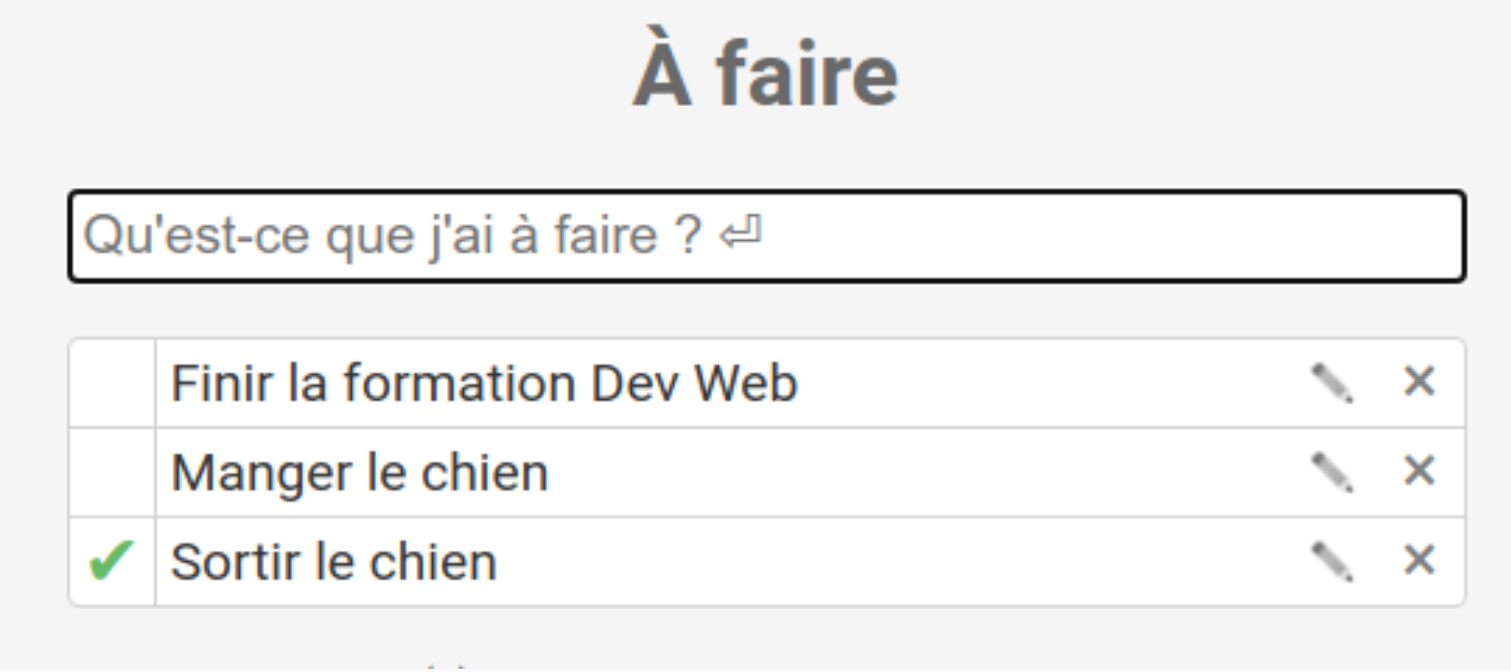
Formation Dev Web
Partie I – Front

Séance 2 : JavaScript

Ressources à télécharger sur http://front2.viarezo.fr

La situation

Une liste de tâches très belle… Mais pas très interactive !



→ La solution : Utiliser du **JavaScript** !

Ressources à télécharger sur http://front2.viarezo.fr

JavaScript (*JS*)

→ Langage de programmation (top 10 des plus utilisés !)

→ Scripts exécutés dans le navigateur, qui peuvent interagir avec la page

→ Plus ou moins normalisé

→ Typage dynamique, similaire à Python, mais en plus laxiste

Syntaxe – Valeurs et litéraux

Comme sur Python : Nombres (`10, 0.1, 1e3`), chaines de caractères (`"hello, world!"`), listes (`[1.0, "un"]`)
(Note : tous les nombres sont des flottants)

Booléens : **`true, false`**
(Pas de majuscule !)

Valeurs spéciales : **`undefined, null`**

Objets (≈ dictionnaires) :

```
{
        clé: "valeur",
        "hissez haut": santiano,
        0: true
}
```

Syntaxe – Variables

Variable locale mutable :

```
let variable = 0;
```

Variable locale immutable :

```
const variable = 0;
```

(Il existe aussi une syntaxe avec 'var' et une syntaxe pour les variables globales)

Assignement :

```
variable = 0;
```

(Attention à avoir déclaré la variable !)

```
variable += 1; variable++;
```

Syntaxe – Opérations de base

Comme sur Python : Opérations numériques de bases (sauf ** et //), comparaisons, appel de fonction, indexation de liste/objet, …

(Note : + fonctionne sur les nombres/strings, mais pas les listes)

Clé de type variable dans un objet : `objet.clé == objet["clé"]`

Existence d'une clé dans un **objet** : `"clé" in objet`

(Pas pour une liste !)

Opérations logiques : `! (non), && (et), || (ou)`

(Attention à doubler les symboles !)

Syntaxe – Contrôle de flux

```
// Commentaire uniligne
/* Commentaire
   multiligne */

if(ma condition) {
      instruction;
      instruction;
} else if(ma condition) {
      ...
} else {
      ...
}
```

Syntaxe – Contrôle de flux (la suite)

```
while(ma condition) { ... }

do { ... } while(ma condition);

for(let i = 0; i < 10; i++) { ... }

for(const clé in objet) { ... }

for(const élément of liste) { ... }
```

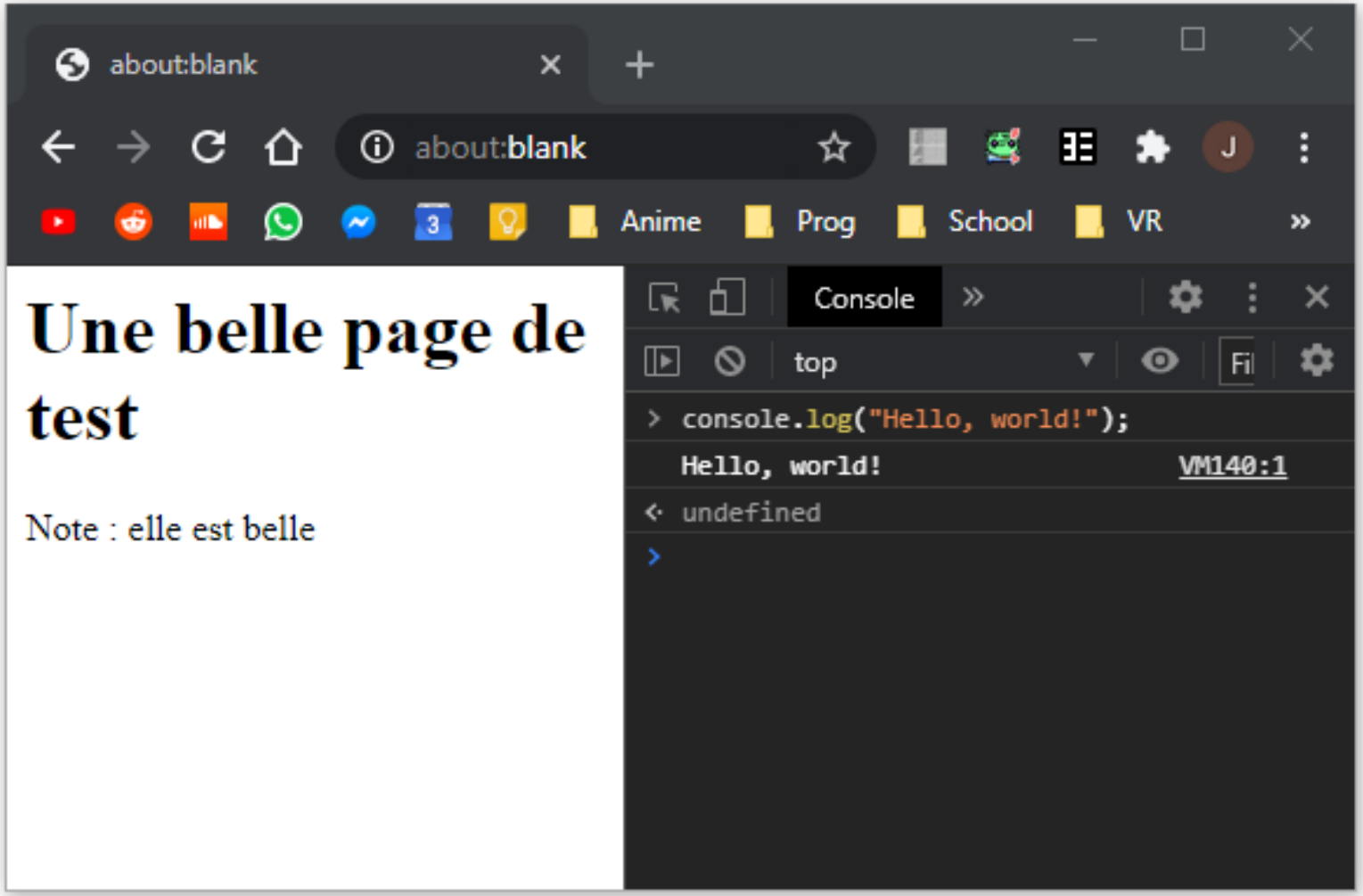
Syntaxe – Fonctions

```
const maFonction = (arg1, arg2) => {
      ...
      return valeur;
};

const ajouter1 = x => x + 1;
// syntaxe raccourcie, équivalent de 'lambda'
```

(Il existe aussi une syntaxe avec le mot clé "function", mais elle est plus longue et quasi-équivalente)

La console



Ctrl + Maj + J (Win/Linux)
⌘ + Option + J (MacOS)

Ctrl + Maj + K (Win/Linux)
⌘ + Option + K (MacOS)

La POO* – En deuspi

```
class Vector {
	constructor(x, y) {
		this.x = x;
		this.y = y;
	}
	setX(newX) {
		this.x = newX;
	}
	getNorm() {
		return Math.sqrt(this.x*this.x + this.y*this.y);
	}
}

const monVecteur = new Vector(0.5, -2);
monVecteur.setX(1);
console.log(monVecteur.getNorm());
```

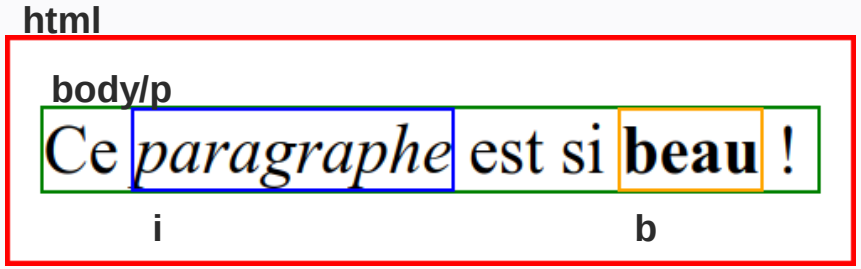
Trucs utiles de la bibliothèque standard

(Référez-vous au PDF docu, ou à MDN (developer.mozilla.org))

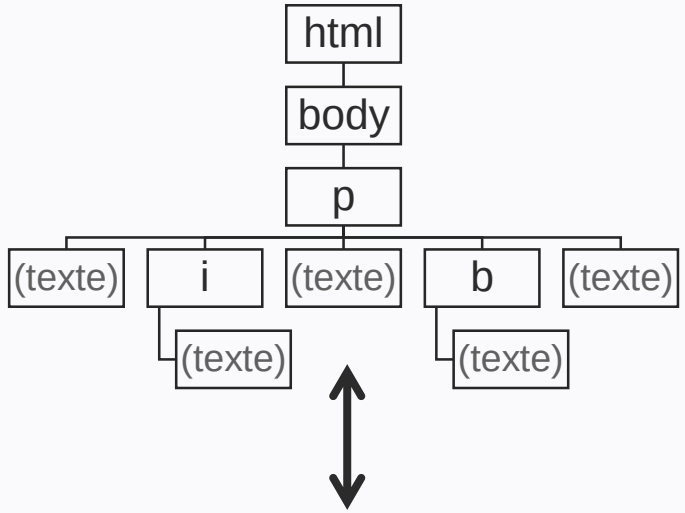
- Array
- String
- Object
- Date
- JSON
- Math
- console
- parseFloat, parseInt

Interaction avec la page

Rendu de la page



DOM



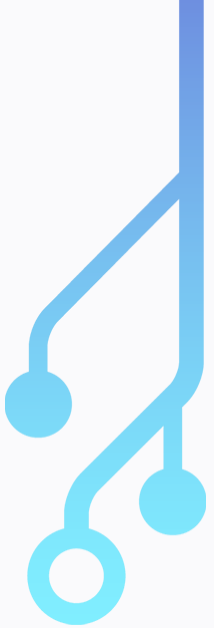
CSS

```css
p {
  color: red;
  font-size: 15px;
}
div {
  display: block;
  background: blue;
}
```

JS

(Chargé via un `<script src="script.js"></script>`)

Les éléments du DOM



Quelques propriétés utiles :

`attributes, children, parentElement, classList, id, style, value` (pour les champs de formulaire)

Quelques méthodes utiles :

```
querySelectorAll, querySelector
appendChild, removeChild, remove, ...
focus
```

Point d'entrée : l'objet global `document`, correspond à `<html>`

Fonctions de création :

```
document.createElement(type)
document.createTextNode(text)
```

Les évènements

Pour réagir aux actions de l'utilisateur :

```
element.addEventListener("type d'évènement", event => {
      ...
});
```

Quelques types d'évènement utiles : `click, keydown, change, …`

Propriétés utiles sur l'objet élément : `key, target, …`

Exemple de manipulation du DOM

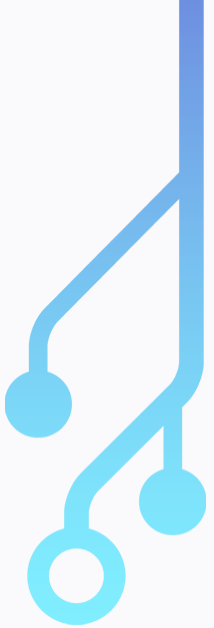
```
const todoList = document.querySelector('.todo-list');

for(const taskLi of todoList.querySelectorAll('li.todo-item')) {
  const taskDelete = taskLi.querySelector('.delete-button');
  taskDelete.addEventListener('click', event => {
    const taskLi = event.target.parentElement;
    taskLi.remove();
  });
}
```

Le TD !

Exercice : Modifier la liste de tâches du TD de la dernière fois (ou corrigé) de manière à pouvoir :

- éditer les items avec le bouton "crayon"
- rajouter des items en appuyant sur entrée dans le champ du haut



Programmation asynchrone

Et toutes les joyeusetés qui viennent avec

Programmation asynchrone – Callbacks

Programmation **synchrone** :

```
const résultat = calculer(paramètres);
utiliser(résultat);
```

Programmation **asynchrone** avec **callbacks** :

```
calculer(paramètres, résultat => {
    utiliser(résultat);
});
```

Le problème des callbacks

```
function hell(win) {
 // for listener purpose
 return function() {
   loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {
     loadLink(win, REMOTE_SRC+'/lib/async.js', function() {
       loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {
         loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {
           loadLink(win, REMOTE_SRC+'/lib/underscode.min.js', function() {
             loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {
               loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {
                 loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {
                   loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {
                     async.eachSeries(SCRIPTS, function(src, callback) {
                       loadScript(win, BASE_URL+src, callback);
                     });
                   });
                 });
               });
             });
           });
         });
       });
     });
   });
 };
}
```

Programmation asynchrone – Promises

```
calcul1() // renvoie une Promise
      .then(résultat1 => {
            return calcul2(résultat1);
            // renvoie une autre Promise, chainée à la première
      })
      .then(résultat2 => calcul3(résultat2))
      .then(résultat3 => afficher(résultat3));
```

La méthode utilisée : `Promise.then(callback)`

Autres méthodes de manipulation de Promises utiles :

`Promise.catch(cb)`, `Promise.all(promiseList)`

Programmation asynchrone – async/await

Nouvelle syntaxe pour simplifier l'usage des Promises :

```
const monGrosCalcul = async () => {
      const résultat1 = await calcul1();
      const résultat2 = await calcul2(résultat1);
      const résultat3 = await calcul3(résultat2);
      afficher(résultat3);
};
```

(Attention, **`await`** ne peut être utilisée que dans une function marquée **`async`**)

"Équivalence" await / Promise

```
const monGrosCalcul = () => {
  return calcul1()
    .then(résultat1 => calcul2(résultat1))
    .then(résultat2 => calcul3(résultat2))
    .then(résultat3 => afficher(résultat3));

};
```

~

```
const monGrosCalcul = async () => {
  const résultat1 = await calcul1();
  const résultat2 = await calcul2(résultat1);
  const résultat3 = await calcul3(résultat2);
  afficher(résultat3);
};
```

Les avantages de await/async

```
const horloge = async () => {
      while(true) {
            await attendre1s();
            console.log('Tick !');
            await attendre1s();
            console.log('Tock !');
      }
}
```

Mais attention…

```
const mesCalculs = async () => {
  const mesDonnées = [1, 2, 3, 4, 5];
  const mesRésultats = [];
  for(const donnée of mesDonnées) {
    const résultat = await monCalcul(donnée);
    mesRésultats.push(résultat);
  }
  return mesRésultats;
};
```

Idiot !

…c'est mieux comme ça

```
const mesCalculs = async () => {
  const mesDonnées = [1, 2, 3, 4, 5];
  const mesPromesses = mesDonnées.map(
    donnée => monCalcul(donnée));
  const mesRésultats = await Promise.all(mesPromesses);
  return mesRésultats;
}
```

Pour une version textuelle et plus en détail de ces explications, aller voir le PDF correspondant, toujours sur http://front2.viarezo.fr

Petits exercices d'asynchrone

Exercice 1 : Écrire une fonction asynchrone qui compte de 1 à 10 (en affichant dans la console), en attendant une seconde entre chaque nombre.

Exercice 2 : Écrire une fonction asynchrone qui affiche les lettres de l'alphabet, avec une demi-seconde entre chaque lettre

Exercice 3 : Écrire une fonction asynchrone qui lance les 2 effets précédents en même temps, et attend qu'ils finissent tous deux avant de finir.

Code donné : fonction asynchrone qui attend un nombre donné de secondes, sans rien renvoyer

```
const wait = time =>
  new Promise((resolve, reject) =>
    setTimeout(resolve, time * 1000));
```

C'est la fin 

On se revoit (j'espère) pour Back 1 !