TYPESCRIPT + VITE.js

Pour avoir la transpilation à chaque enregistrement de code dans le terminal :
- cmd = `tsc -w`

Typer une Fonction = Typer sa signature

Une **SIGNATURE** d'une fonction, c'est 3 choses:
- Le NOM de la fonction
- Le TYPE des valeurs d'entrée de la fonction (ex: paramètres)
- La TYPE des valeurs de sortie de la fonction (ex: valeur de return)

Le type 'ANY' est un type que TypeScript **DETESTE**. Il existe toujours un type bien défini attendu. ANY permet d'accueillir n'importe quel autre type à l'interieur.

Pour connaitre le vrai type d'un paramètre de type 'any' dans une fonction il faut aller chercher ce paramètre dans son contexte d'utilisation .
Faire cmd +d en sélectionnant la fonction nous amène aux endroits ou elle est utilisée.
Ensuite il faut regarder le type de son paramètre la ou elle est utilisée.
Une fois le paramètre connu il faut typer quand la fonction est définie

Ex: `const handleDelete = (id:number): void => {}`

:void = typage EXPLICITE du return. permet de typer la valeur du return, ici on a pas de return, on attend rien donc on sait que c'est 'void'. Il il se place après le paramètre de la fonction

On est pas obligé de le mettre car en utilisant 'l'inférence du contexte' TS sait déjà que le return attend un type :void. Donc on est pas obligé de l'écrire:

```
const handleDelete = (id:number) => {}
```

Typer un objet qui est en (paramètre) d'une fonction :
Le type d'un objet = le type de ses propriétés.

ex: on passe de : `const handleAdd = (fruitAAjouter) => {}`

à:

ex: `const handleAdd = (fruitAAjouter: { id:number; nom: string} ) => {}`

L'objet fruitAAjouter à 2 propriétés que l'on va typer.

ou alors on peut typer cet objet en (paramètre) de manière EXPLICITE avec un custom type :
(Soit avec le mot **type** / soit **interface** (sans le '=')

```
type FruitAAjouter  = { id:number; nom: string}
// Déclaration d'un type personnalisé (custom Type) de maniere EXPLICITE
//ici le type est FruitAAjouter

interface FruitAAjouter {
  id: number;
  nom: string;
}
```

⚠️ Interface ne peut typer QUE des objets , utiliser type pour typer les primitifs (number/string/ boolean, void et null) ⚠️

Definition d'un custom type dans un dossier à part pour réutilisation dans tout le projet:

Dans votre dossier src -> créer un dossier typescript (ou autre nom) puis créer dedans un fichier **.ts** *(car ce n'est pas un composant, pas de .tsx)* dans lequel je vais inscrire mes type custom que je réutiliserai dans tout le projet.

Ex: **export type** FruitAAjouter = { id:**number**; nom:**string** }
→ utiliser export devant le type car on va l'utiliser ailleurs dans le projet.

L'utiliser dans un fichier externe:
→ écrire FruitAAjouter (là ou j'en ai besoin) puis le sélectionner, puis cmd + i et cliquer sur FruitAAjouter dans la liste déroulante pour l'autoImport.
Sinon écrire : `import { FruitAAjouter } from ''./ typescript/FruitType''`

Typer un composant

C'est comme typer une fonction : il suffit de préciser sa signature.

export default function Fruit(**{** fruitInfo, onClick **}**) {
}. ⚠️ Il n'ya QU'UN seul (paramètre) à Fruit qui est un objet (props) déstructuré avec 2 propriétés 'fruitInfo' et 'OnClick' . Donc il ne faudra typer que l'objet props. ⚠️

Ce qui donne: export default

```
Export default function Fruit ({fruitInfo,onClick,}:
{
    fruitInfo: FruitType;
    onClick: () => void;
  }) {Le reste du composant}
```

Rappel: Le typapge d'un objet se fait à travers ses propriétés.

OU

On peut utiliser le **custom type** pour aérer l'écriture:

```
type FruitProps = {
  fruitInfo: FruitType;
  onClick: () => void;
};

export default function Fruit({ fruitInfo, onClick }: FruitProps) {
```

(On remplace bien sur {
fruitInfo: FruitType;
onClick: () => void;
} **par le type 'FruitProps'.** Il est d'usage d'appeler un type de props **'NomComposantProps' (ici 'FruitProps').**

Et enfin pour **la valeur de return**: En passant le curseur sur le composant **TS** nous indique qu'il a inféré automatiquement et trouvé que c'est du JSX.Element.
Pas besoin donc de l'ajouter comme ceci : export default function Fruit({ *fruitInfo*, *onClick* }: FruitProps)**:JSX.Element** {}

Il y aura toujours du JSX dans un composant et il sera toujours inféré par TS. Donc finalement : Typer un composant = Typer ses props .

— — — — — —

Définition : Le terme **"Binding element"** dans TypeScript fait référence à une variable ou une constante qui est créée via une **destructuration** d'objet ou de tableau.
ex: export default function FruitForm({ *handleAdd* }) {} (handleAdd est un Biding Element)

— — — — — —

Autre cas de tapage:

Ci-dessous, le type de *handleAdd* = tout ce qui est après le marqueur « : »
Donc:' (fruitAAjouter: Fruitype) => void'

```
  ))}
</ul>
<FruitForm handleAdd={handleAdd} />
```

```
const handleAdd: (fruitAAjouter: FruitType) => void
```

— —

TYPER un EVENEMENT :

```
const handleSubmit = (event) =>{
  event.preventDefault();
    const id = new Date().getTime();
    const nom = nouveauFruit;
    const fruitAAjouter = { id, nom };
    //fruitsCopy.push(fruitAAjouter);
    handleAdd(fruitAAjouter);
    setNouveauFruit("");
  };
```

Dans l'exemple ci-dessus il faut typer le paramètre (*event*) qui est de type 'any'
→Faire un cmd+d pour observer le contexte d'utilisation: on le retrouve dans le eventListener OnSubmit.

```
onSubmit={handleSubmit}
```

→ Copier coller tout ce qui est après le '=' et remplacer { handleSubmit } par tout ce qui a été copié car c'est la même chose.
→ Survoler à nouveau (event) : On remarque qu'il est de type :

```
React.FormEvent<HTMLFormElement>)
```

Maintenant il n'y a plus qu'à typer (event) dans sa déclaration et remettre onSubmit={handleSublmit}

```
const handleSubmit = (event: React.FormEvent<HTMLFormElement>) =>{
    event.preventDefault();
    const id = new Date().getTime();
    const nom = nouveauFruit;
    const fruitAAjouter = { id, nom };
    //fruitsCopy.push(fruitAAjouter);
    handleAdd(fruitAAjouter);
    setNouveauFruit("");
  };
```

TYPER un HOOK :

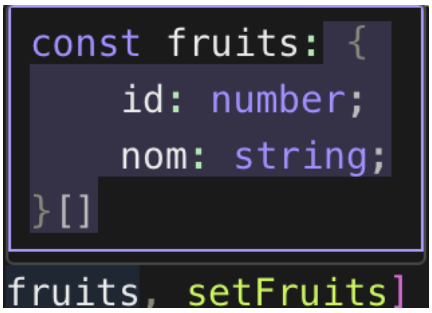
Pour typer un hook, il faut préciser le type de la valeur initiale et les autres valeurs potentielles que peut prendre le hook.

Dans le cas de **useState** : ça revient simplement à **préciser le type de la valeur d'initialsation.**

Ex:

```
const [fruits, setFruits] = useState([
   { id: 1, nom: "Abricot" },
   { id: 2, nom: "Banane" },
   { id: 3, nom: "Cerise" },
 ]);
```

On type un hook avec des chevrons: permet de préciser le type de valeurs possibles du hook.



Ceci et le type de ma valeur d'initalisation du hook useState. Je l'obtient en survolent la variable 'fruits'.
Donc il faut mettre entre <crochets> tout ce qui est souligné.
Dans mon projet je sais que ça correspond à FruitType.

```
const [fruits, setFruits] = useState<FruitType[]>([
  { id: 1, nom: "Abricot" },
  { id: 2, nom: "Banane" },
  { id: 3, nom: "Cerise" },
]);
```

Donc: en survolent 'fruit' à nouveau je vois le nouveau type du hook useState qui est de type un tableau de FruitType (FruitType[]):

```
const fruits: FruitType[]
```

Il existe une autre notation pour dire un tableau de FruitType: `Array<FruitType>`
C'est pareil que dire: FruitType[] (<<-privilégier cette dernière)

Pourquoi typer un HOOK ?

→ Car si le code est amené à évoluer , par exemple si on veut rajouter à l'avenir une propriété supplémentaire au custom type FruitType , TypeScript détectera automatiquement plusieurs erreurs. Car partout ailleurs ou on utiliser ce type là, **il faudra actualiser** les propriétés.
C'est une sécurité supplémentaire contre les bugs.

———————————————————

TypeScript en équipe :

———————————————————

Rendre optionnelle la propriété d'un objet dans un Custom Type

```
  type FruitProps = {
      fruitInfo: FruitType;
      onClick: () => void;
backgroundColor?: string;
  };
```

Dans le JSX : `<button style={{ backgroundColor: 'red' }}> Click me </button>`

Je peux aussi **cloisonner le type de string utilisable en tant que backgroundColor?** .
Si par exemple le designer a un souci de cohérence de couleur pour toute l'app, il est possible de n'autoriser que 3 couleurs par exemple.
Grace à un **union type**: `backgroundColor?: "Red" | "Yellow" | "blue";`

```
  type FruitProps = {
      fruitInfo: FruitType;
      onClick: () => void;
backgroundColor?: "red" | "yellow"| "blue" ;
  };
```

`ctrl+ espace` afin de savoir quel valeur est disponible après une variable ou une propriété.

En résumé: TypeScript est utile pour maintenir plus facilement le code surtout de gros projets mais également créer une documentation automatisée, tout en prévenant les bugs**.**

- Typer **une fonction**
  - Typer un **objet en paramètre d’une fonction**
  - Définir un **custom type dans un fichier à part** en **.**ts pour l’utiliser partout dans le projet
  - Typer **un composant**
  - Typer **un event**
  - Typer **un Hook**
  - Rendre **optionnelle une propriété dans un objet d’un custom type**
    - **union type**

fffdd