# Cheat Sheet - Les Hooks React

## Qu'est-ce qu'un Hook ?

Un Hook = une **fonction spéciale** qui permet d'utiliser des fonctionnalités React dans un composant fonction.

> Tous les hooks commencent par `use` : useState, useEffect, useRef...

---

## LA règle fondamentale des Hooks

### Les hooks doivent TOUJOURS être appelés :

1. **Au niveau supérieur** du composant (pas dans des if, boucles, fonctions imbriquées)
2. **Dans le même ordre** à chaque rendu

```
// CORRECT
function MonComposant() {
  const [count, setCount] = useState(0);      // Hook #1
  const [name, setName] = useState("");       // Hook #2
  const inputRef = useRef(null);              // Hook #3

  // ... reste du composant
}

// INTERDIT : hook dans une condition
function MonComposant({ isAdmin }) {
  if (isAdmin) {
    const [data, setData] = useState(null);   // ERREUR !
  }
}

// INTERDIT : hook dans une boucle
function MonComposant({ items }) {
  for (let item of items) {
    const [state, setState] = useState(item);  // ERREUR !
  }
}
```

**Pourquoi ?** React identifie les hooks par leur **position**, pas par leur nom.

---

## useState - Gérer l'état local

### Syntaxe

```
const [valeur, setValeur] = useState(valeurInitiale);
//      |         |                   |
//      |         |                   └── valeur de départ
//      |         └── fonction pour modifier
//      └── valeur actuelle
```

### Exemples par type

```
// String
const [name, setName] = useState("");
const [name, setName] = useState("Victor");

// Number
const [count, setCount] = useState(0);
const [age, setAge] = useState(26);

// Boolean
const [isOpen, setIsOpen] = useState(false);
const [isLoading, setIsLoading] = useState(true);

// Array
const [items, setItems] = useState([]);
const [users, setUsers] = useState([{ id: 1, name: "Victor" }]);

// Object
const [user, setUser] = useState(null);
const [form, setForm] = useState({ email: "", password: "" });
```

### Mise à jour de l'état

```
// Mise à jour simple
setCount(5);
setName("Alice");
setIsOpen(true);

// Mise à jour basée sur la valeur précédente (RECOMMANDÉ)
setCount(prev => prev + 1);
setCount(prev => prev - 1);
setIsOpen(prev => !prev);  // toggle

// Mise à jour d'un tableau
setItems(prev => [...prev, newItem]);           // Ajouter
setItems(prev => prev.filter(i => i.id !== id)); // Supprimer
setItems(prev => prev.map(i =>
  i.id === id ? { ...i, name: "New" } : i        // Modifier un élément
));

// Mise à jour d'un objet
setUser(prev => ({ ...prev, name: "Alice" }));
setForm(prev => ({ ...prev, email: "a@b.com" }));
```

### Piège : le state est asynchrone !

```
function handleClick() {
  setCount(count + 1);
  console.log(count);  // Affiche l'ANCIENNE valeur !
}

// Solution : utiliser la forme fonctionnelle
function handleClick() {
  setCount(prev => {
    const newValue = prev + 1;
    console.log(newValue);  // Affiche la nouvelle valeur
    return newValue;
  });
}
```

---

## useEffect - Effets de bord

### Syntaxe

```
useEffect(() => {
  // Code de l'effet (s'exécute APRÈS le rendu)

  return () => {
    // Cleanup (optionnel, s'exécute avant le prochain effet ou au démontage)
  };
}, [dépendances]);
```

### Les 3 cas d'utilisation

```
// 1. Exécuter UNE FOIS au montage (tableau vide)
useEffect(() => {
  console.log("Composant monté !");
  fetchData();
}, []);  // [] = aucune dépendance = 1 seule fois

// 2. Exécuter QUAND une valeur change
useEffect(() => {
  console.log("Count a changé:", count);
  document.title = `Compteur: ${count}`;
}, [count]);  // Se relance quand count change

// 3. Exécuter À CHAQUE rendu (rarement utilisé)
useEffect(() => {
  console.log("Rendu effectué");
});  // Pas de tableau = à chaque rendu
```

### Cleanup : nettoyer après soi

```
// Exemple : event listener
useEffect(() => {
  function handleResize() {
    setWidth(window.innerWidth);
  }

  window.addEventListener('resize', handleResize);

  // Cleanup : retirer le listener quand le composant disparaît
  return () => {
    window.removeEventListener('resize', handleResize);
  };
}, []);

// Exemple : interval
useEffect(() => {
  const id = setInterval(() => {
    setSeconds(s => s + 1);
  }, 1000);

  return () => clearInterval(id);  // Stopper l'interval
}, []);

// Exemple : subscription
useEffect(() => {
  const subscription = api.subscribe(data => setData(data));

  return () => subscription.unsubscribe();
}, []);
```

### Pattern : Fetch de données

```
function UserProfile({ userId }) {
  const [user, setUser] = useState(null);
  const [isLoading, setIsLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    setIsLoading(true);
    setError(null);

    fetch(`/api/users/${userId}`)
      .then(res => {
        if (!res.ok) throw new Error('Erreur réseau');
        return res.json();
      })
      .then(data => {
        setUser(data);
        setIsLoading(false);
      })
      .catch(err => {
        setError(err.message);
        setIsLoading(false);
      });
  }, [userId]);  // Se relance si userId change

  if (isLoading) return <p>Chargement...</p>;
  if (error) return <p>Erreur : {error}</p>;
  return <div>{user.name}</div>;
}
```

---

## useRef - Références et valeurs persistantes

### Deux utilisations

```
// 1. Référencer un élément DOM
function InputFocus() {
  const inputRef = useRef(null);

  function handleClick() {
    inputRef.current.focus();  // Accès direct au DOM
  }

  return (
    <>
      <input ref={inputRef} type="text" />
      <button onClick={handleClick}>Focus</button>
    </>
  );
}

// 2. Stocker une valeur qui persiste sans causer de re-render
function Timer() {
  const intervalId = useRef(null);

  function start() {
    intervalId.current = setInterval(() => console.log('tick'), 1000);
  }

  function stop() {
    clearInterval(intervalId.current);
  }

  return (
    <>
      <button onClick={start}>Start</button>
      <button onClick={stop}>Stop</button>
    </>
  );
}
```

### Différence useState vs useRef

| | useState | useRef |
|---|---|---|
| Modification | Cause un re-render | PAS de re-render |
| Accès | `value` | `ref.current` |
| Usage | Données affichées | DOM, timers, valeurs internes |

---

## Tableau récapitulatif des Hooks

| Hook | Usage | Re-render ? |
|---|---|---|
| `useState` | Gérer un état local | Oui |
| `useEffect` | Effets de bord (API, timers, DOM) | Non |
| `useRef` | Référence DOM ou valeur persistante | Non |
| `useContext` | Accéder au contexte global | Oui (si contexte change) |
| | | |

| Hook | Usage | Re-render ? |
| --- | --- | --- |
| `useReducer` | État complexe avec actions | Oui |
| `useMemo` | Mémoriser une valeur calculée | Non |
| `useCallback` | Mémoriser une fonction | Non |

---

## Schéma mental : Cycle de vie avec Hooks