# Motivation

Imperative vs Functional programming

- Functional programming is recommended
- But world (IO) is not functional
- You need a backdoor to access world in an imperative way
  - For example : unSafePerformIO (Haskell)

- useRef is related to this idea

# useRef

https://reactjs.org/docs/refs-and-the-dom.html

- ref  means the memory reference
- Refs provide a way to access DOM nodes or React elements
- In the typical React dataflow, props are the only way that parent components interact with their children.
- To modify a child, you re-render it with new props.
- However, there are a few cases where you need to imperatively modify a child outside of the typical dataflow.

# When to Use Refs

There are a few good use cases for refs:

- Managing focus, text selection, or media playback.
- Triggering imperative animations.
- Integrating with third-party DOM libraries.

Avoid using refs for anything that can be done declaratively.

```
const ref = useRef(value)
```

- Object property : current
  { current: value }

- To read the value :
  - ref.current

- To update the value:
  - ref.current = newValue  // using assignment

# useRef : imperative programming

- The values of ref variables are address:
  - `<Tag  ref={refVar}  …. />`
  - `Similar to  const refVar = document.getElementById`

- Update is made by assignment

```
const refVar = useRef(“hi”)
refVar.current = “hello”
refVar.current = “nice”
```

# Values of variables

- File : UseRefStarEx2.jsx
- ordinary variables, useState, useRef
- Values of Variables
  - State change  => Rendering -> Re-initialization for ordinary variables
  - State Change => Rendering => values of ref variables are preserved
  - Ref change  => No Rendering => values of ordinary variables are preserved
  - Variables of useState and useRef are not re-initialized by re-rendering