1. **HTML Semantic**

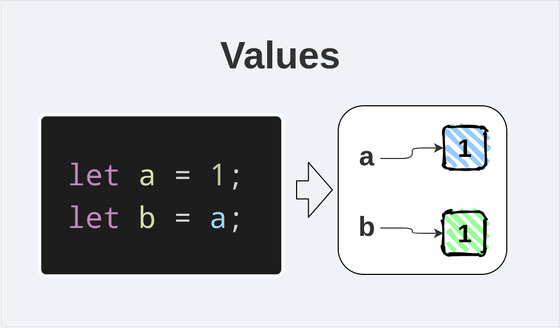
* Semantic elements = elements with a meaning

1. **CSS animation**

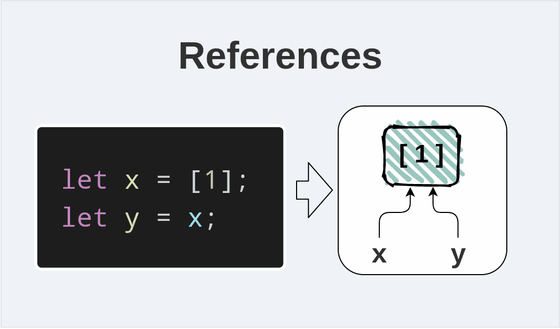
* Animation delay: the amount of time to wait before perform the animation
* Animation direction: set whether an animation should play forward, backward or go back and forth
* Animation duration: the length of time to complete a cycle
* Animation iteration count: set the number of times an animation should be play
* Animation play state: set the animation run or stop
* Animation timing function: set how animation process through each cycle

1. **Reference type & Value type**

* Value type (primitive value): when assign value for variable, a copy of that value is created

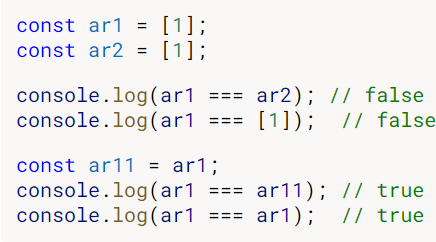
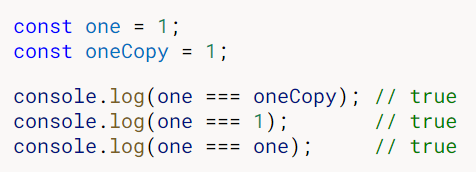


* Reference type (object, array): when create an object, you’re given a reference to that object



1. **Compare value & compare reference**

* When using strict comparison (===), 2 variables have the same values are equal. But 2 reference (object, array) are equal only when they reference the same object



1. **Primitive data in JS**

* String
* Number
* Bigint: represent integers, floats, doubles
* Boolean
* Undefined
* Symbol: create symbol syntax is the same, but it value always unique
* Null

1. **Null & Undefined & NaN**

* Null: object is empty and not point to any memory address
* Undefined: a object don’t have any value => because declare variable but don’t assign value
* Nan: not a number => because invalid math operations

1. **JS this keyword**

* This keyword refers to an object. Which object depends on how this is being used
  + In object method => refers to the object
  + Alone => refers to global object
  + In function => refers to global object
  + In function strict mode => this is undefined
  + In event => refers to element that received the event

1. **Var, let, const**

|  |  |  |
| --- | --- | --- |
| Var | Let | Const |
| Function scope | Block scope | Block scope |
| Can re-declared and updated | Cant be re-declared but can updated | Cant be re-declared and updated |

* Hoisting of var -variable can be used before declared: a JS behavior of moving all declarations to the top of the curent scope. But variables declare with let and const hoisted to the top of the block but not initialized – cannot be used until it has been declared.

1. **== & === (double equals & triple equals)**

* Double equals (==) operator does the type conversion of the operand before comparison. So you can use double equals to compare the identity of 2 operands even though, they are not of the similar type.
  + 2 == “2” => true
  + 2 == “Hello” (Nan type) => false
  + Object == object (not the same reference) => false
* Triple equals (===) operator compares the values as well as the data types of the operands.
  + 2 === “2” => false
  + 2 === 2 => true
  + Object === object => false

1. **Arrow function? Why arrow function**

* Arrow function allow us to write shorter function syntax. In arrow function, there is no binding of this keyword. Because the this keyword always represents the object that defined the arrow function.

1. **Class component vs functional component**

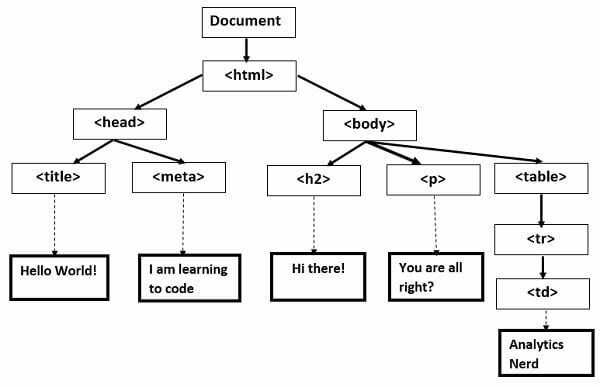
* Class component extend from React with render method to return React elements.
* Functional component wit hooks are Javascript function that return React elements
* Functional component with hooks (version 16.8) are introduced to replace class component because:
  + It is hard to reuse login between components
  + Complex component are hard to undestand
  + Class confused both people and machine

1. **React Life Cycle**

* Mouting (Add to DOOM – show to UI): Run only 1 time => call constructor => render => componentDidMount
* Updating: run when props or state change
* Unmounting (Remove from DOOM – Remove from UI): Run only 1 time => componentWillUnmount

1. **Real DOM & Virtual DOM**

* DOM stands for Document Object Model. It represents the entire UI of the web application as tree data structure



* When a state change, the DOM gets manipulated and re-render to update the UI. Therefore, with more components and complex structure of the DOM, the DOM updates will be more costly as with every change it needs to re-rendered.
* Virtual DOM is also represented as a tree structure. If a state of any components change, a new virtual DOM tree is created. Then the current version of DOM is compared to the previous version to know which object are changed. Based on that, it only change those objects in the real DOM, making minimal operations.

1. **Closure ? closure in react**

* Closure are functions that have access to the variables that are present in their scope chain even if the outer function ceases to exit

1. **useEffect return? When ?**

* The return of useEffect is where the cleanup happens. The cleanup function prevents momory leaks and removes some unnecessary and unwanted behaviors.
* Cleanup function run right before the execution of the next scheduled effect or when component wants to unmount.
* If component unmounts before promise resolves, useEffect will try to updae state on an unmounted component cause the error. To fix this error, we use the cleanup function to resolve it.



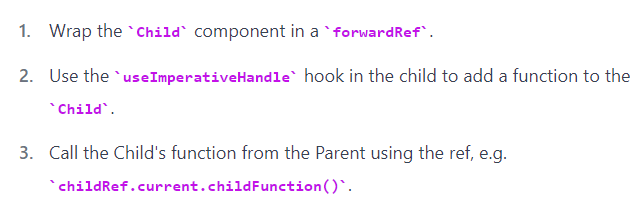
* Example of cleanup:

Text

Description automatically generated

1. **useRef, forwardRef**

* useRef
  + useRef is used to keep variables between rerendering react component. Changing the useRef value doesn’t make rerender. useState also keep variables between rerendering but when its value change, the component which use this state will be rerender.
  + useRef is used to access DOM directly why React work with virtual DOM. That why we should not overuse useRef.
  + Some useRef use cases: work with animation, text selection, focus, media playback
* forwardRef:
  + If we try passing a ref as prop from parent to child component, we will get the error that say ref is not a prop, and react suggest us to use forwardRef
  + Call child’s function from parent component



1. **useMemo**

* useMemo keeps a function from being executed again if its dependencies not change. When 1 of dependencies is changed, the function will be called. Then, useMemo will returns the results of that function execution and will store it in memory to prevent the function from running again if the same parameters were used. That why overuse useMemo will make the result of functions execusion grow big and harm the app performance.
* Only use useMemo when you’re trying to prevent re-running expensive functions, that run a lot of time, or using a lot of resources.

1. **useCallback**

* useCallback keeps a function from being re-created again when component re-render, based on a list of dependencies. It returns the function itself.
* In many cases, useCallback doesn’t provide any actual benefit. That why only use it to prevent a costly function from re-running. For example, use useCallback to prevent re-running a function that performs an API call

1. useReducer
2. **useContext**

* useContext make a particular data available to all components no matter how they are nested.
* useContext is not designed for frequent updates, and when the context value changed, then all of it comsumers are re-rendered and you cannot prevent re-render even if you use useMemo or useCallback.
* In a simple application, you might be able to get things done with just using useContext for global data storage. But in large scale production enviroment, you’re likely to use Redux for global state management.



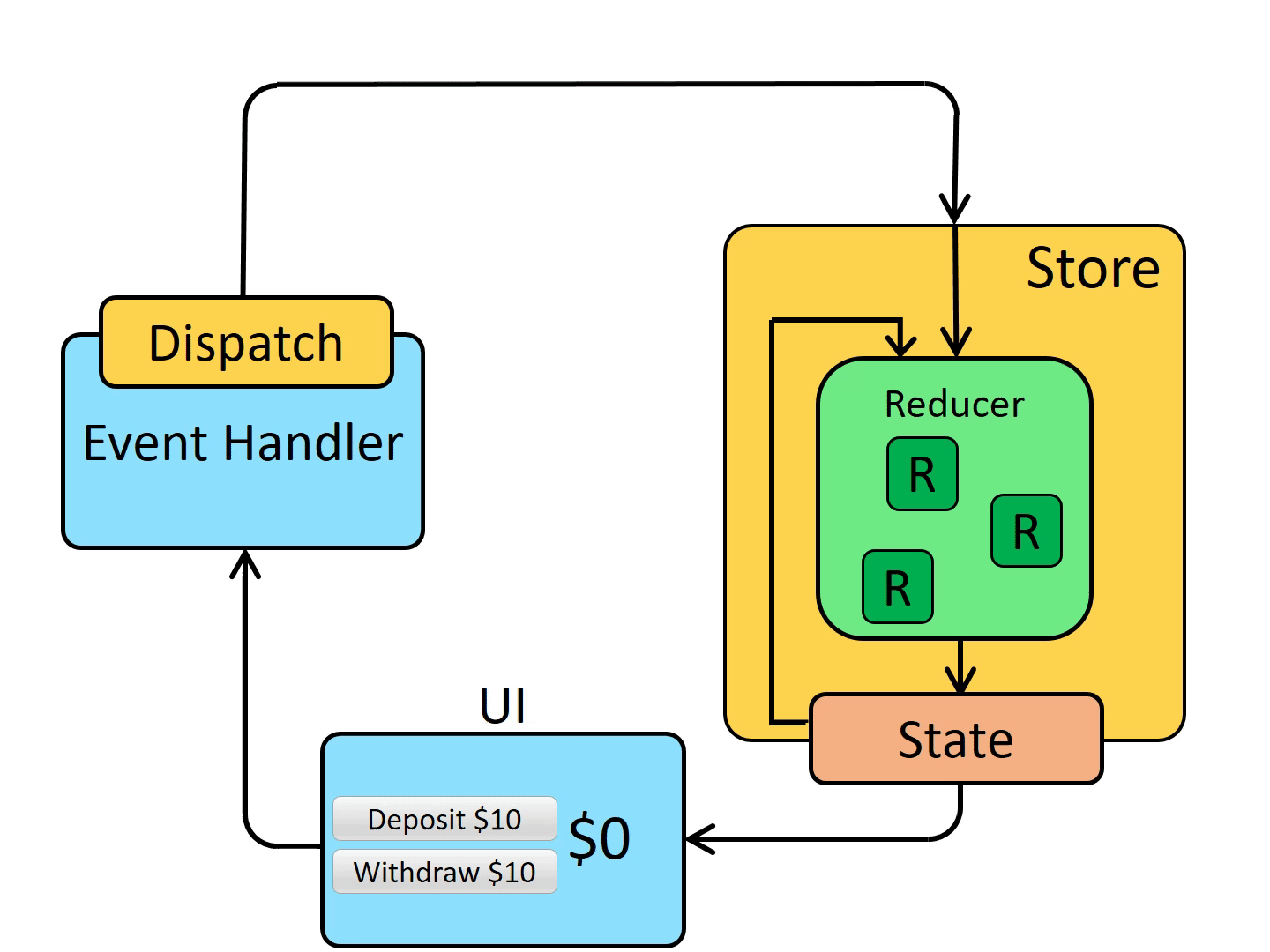
Text

Description automatically generatedText

Description automatically generated

1. **Redux ? Redux flow ? redux thunk & redux saga**

* Redux is a predictable state container for javascript app
* In the initial step, redux store is created and its initial value is passed to UI. When users interact with UI, an action – javascript object – is sendt to store. The store run reducer with the current state and action, and save the return value as new state. Each components that use redux store checks if the state their need have changed. If yes, then component forge re-render to update the new state in UI .



1. useLayoutEffect
2. useImperativeHandle