Part 3 - 요약정리

1. Computed Properties
    - `computed()`
    - Computed Caching
    - Getters should be side-effect free
    - Avoid mutating computed value
2. Conditional Rendering
    - `v-if` on `<template>`
    - `v-if` vs. `v-show`
3. List Rendering
    - Object Rendering
    - `v-for` with a Range
    - `v-for` with `v-if`
    - Maintaining State with `key`
    - Array Change Detection
4. Watchers
    - Watch Source Types
    - Deep Watchers
    - Callback Flush Timing
5. Lifecychle Hooks

1. Computed Properties

```
<p>Has published books:</p>
<span>{{ author.books.length > 0 ? 'Yes' : 'No' }}</span>
```

위와 같이 template expression에 복잡한 로직을 넣게되면 코드를 이해하기 어려워질 수 있는데, 이때 사용해볼 수 있는 것이 `computed property`이다. 이를 사용해 아래와 같이 리팩토링을 할 수 있다.

```
// a computed ref
<script setup>
const publishedBooksMessage = computed(() => {
  return author.books.length > 0 ? 'Yes' : 'No'
})
</script>

<template>
  <p>Has published books:</p>
```

```
  <span>{{ publishedBooksMessage }}</span>
</template>
```

`computed()`

- `computed()` 함수는 getter를 인자로 받고, **computed ref** 를 리턴한다**.**
- computed property는 알아서 dependency의 변화를 추적해준다.

Computed Caching

📌 computed property는 dependency가 바뀔 때만 다시 평가되며, 그 이외에는 캐싱해두었던 값을 바로 리턴하기 때문에 효율적이다.

→ 따라서 아래의 코드는 절대 업데이트 되지 않는다.

```
const now = computed(() => Date.now())
//will never update, because Date.now() is not a reactive dependency
```

Getters should be side-effect free

- `computed()` 함수가 인자로 받는 getter는 비동기 요청이나 DOM을 수정해서는 안 된다.
- 오직 값을 돌려받는 데 집중하자.

→ 비동기 요청이나 DOM을 수정하는 일은 `watcher` 를 통해서.

Avoid mutating computed value

computed property는 dependency에 의존하여 도출된 값이다. 따라서 computed value를 바꿀 생각하지 말고, dependency를 수정하도록 하자.

2. Conditional Rendering

- `v-if`
- `v-else`
- `v-else-if`
- `v-show`

`v-if` on `<template>`

여러 html 요소를 `v-if` 로 처리하고 싶다면 `div` 대신에 `<template>` 을 사용하자.

```
<template v-if="ok">
  <h1>Title</h1>
  <p>Paragraph 1</p>
  <p>Paragraph 2</p>
</template>
```

최종적으로 렌더링 된 결과에는 `<template>` 요소가 포함되지 않는다.

`v-if` vs. `v-show`

- `v-if` 에서 false의 경우 실제 DOM에는 해당 요소가 렌더링되지 않는다.
  - 따라서 조건의 true/false 가 자주 변하는 경우 cost가 많이 든다.
- `v-show` 에서 false의 경우 실제 DOM에 해당 요소가 렌더링되며, 요소의 display(css) 속성이 none으로 처리된다.
  - 따라서 초기 렌더링 cost가 많이 든다.
  - 런타임의 조건의 true/false 가 자주 변하는 경우 v-show를 쓰는 게 일반적으로 효율적이다.

3. List Rendering

Object Rendering

객체는 이런 식으로 가능하다.

```
<li v-for="(value, key, index) in myObject">
  {{ index }}. {{ key }}: {{ value }}
</li>
```

`v-for` with a Range

파이썬 유저는 신나할 기능.

```
<span v-for="n in 10">{{ n }}</span>
```

`v-for` with `v-if`

- `v-if` 가 `v-for` 보다 높은 우선순위를 가지고 있다.

> 📌 따라서 `v-if` 와 `v-for` 를 같은 요소에 쓰지 말자. 이 둘에는 우선순위가 정해져 있어서 둘 중 하나만 적용되는 불상사가 발생한다.

```
<!--
This will throw an error because property "todo"
is not defined on instance.
-->
<li v-for="todo in todos" v-if="!todo.isComplete">
  {{ todo.name }}
</li>
```

→ 대신 `template` 을 활용한다.

```
<template v-for="todo in todos">
  <li v-if="!todo.isComplete">
    {{ todo.name }}
  </li>
</template>
```

Maintaining State with `key`

`key` Expects: `number | string | symbol`

> 📌 **It is recommended** to provide a `key` attribute with `v-for` whenever possible.

Array Change Detection

Vue는 `원본 배열` 에 변화를 주는 메서드를 감지하여 필요한 업데이트를 수행한다. 다음 자바스크립트의 배열 메서드들은 새로운 배열을 생성하지 않고, `원본 배열을 수정` 하는 메서드들이다.

- `push()`
- `pop()`

- `shift()`
- `unshift()`
- `splice()`
- `sort()`
- `reverse()`

> 따라서 원본 배열을 수정하지 않고 새로운 배열을 생성하는 메서드들의 경우, 기존에 `ref`로 참조하고 있던 배열을 새로운 배열으로 완전히 바꿔주어야 한다.

```
// `items` is a ref with array value
items.value = items.value.filter((item) => item.message.match(/Foo/))
```

4. Watchers

- 리액트의 `useEffect`와 유사한 것 같다.
- 이전에 `computed` 섹션에서 비동기 요청이나 DOM을 수정하는 일은 `watcher` 를 통해서 하자고 했다.

Watch Source Types

- `watch()` 의 첫 번째 인자로는 ref, computed ref, getter, 혹은 이것들을 담은 배열 등이 올 수 있다.
- 두 번째 인자로 수행할 callback function이 온다.

```
const x = ref(0)
const y = ref(0)

// single ref
watch(x, (newX) => {
  console.log(`x is ${newX}`)
})

// getter
watch(
  () => x.value + y.value,
  (sum) => {
    console.log(`sum of x + y is: ${sum}`)
  }
```

```
)

// array of multiple sources
watch([x, () => y.value], ([newX, newY]) => {
  console.log(`x is ${newX} and y is ${newY}`)
})
```

Deep Watchers

- 객체를 추적하고 있을 경우 객체 안의 값까지 자동으로 추적한다.

```
const obj = reactive({ count: 0 })

watch(obj, (newValue, oldValue) => {
  // fires on nested property mutations
  // Note: `newValue` will be equal to `oldValue` here
  // because they both point to the same object!
})

obj.count++
```

Callback Flush Timing

- 만약 ref의 state가 변하면, 아래 두 가지가 모두 실행된다.
  - 뷰 컴포넌트 업데이트
  - `watch` 의 콜백함수
- 그렇다면 어떤 게 먼저 실행될까?
  - `watch` 의 콜백함수가 먼저 실행된다.

따라서 만약 `watch` 의 콜백함수에서 DOM에 접근한다면, 해당 DOM은 이전의 상태값을 가지고 있을 것이기 때문에 주의해야 한다.

만약 뷰 컴포넌트의 업데이트를 먼저 실행되게 하고 싶다면 다음과 같이 `watch` 의 세 번째 인자로 `flush: 'post'` 옵션을 전달할 수 있다.

```
watch(source, callback, {
  flush: 'post'
})

watchEffect(callback, {
  flush: 'post'
})

import { watchPostEffect } from 'vue'
```

```
watchPostEffect(() => {
  /* executed after Vue updates */
})
```

5. Lifecychle Hooks