Day3 Computed method watch 그리고 lifecycle hooks

자료	Vue
구분	Vue
과목	

Computed

반응형 앱을 구현하는데 큰 역할을 담당하는 중요한 기능 중 하나다.

ref객체 그리고 뒤에 배울 watcher 등은 모두 실시간으로 데이터의 변경을 감시하지만,

computed는 원하는 대로 데이터를 변경해 주는 강력한 기능이다.

물론 computed를 이용하지 않고 v-if 또는 v-for를 이용해서 비슷하게 구현이 가능하다. 하지만 v-if 또는 v-for를 이용한 코드 보다는 computed를 이용하면 복잡하지 않고 간단하게 구현이 가능한 경우가 많다.

아래 코드를 직접 타이핑 해보면서 computed를 사용해 보자.

예제 3개를 준비했다. 모든 예제를 직접 typing을 작성 해보고 브라우저 통해서 결과를 꼭 확인해 보자.

예제1>

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
```

```
<body>
 <script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>
 <div id="app">
  <h2>Big items</h2>
  <p v-for="item in big_item" :key="item.id">{{item.text}}</p>

  <h2>Small items</h2>
  <p v-for="item in small_item" :key="item.id">{{item.text}}</p>
  <h1> 13기 2반 화이팅 </h1>
 </div>

 <script>
  const { createApp, ref, computed } = Vue;

  const app = createApp({

   setup() {
    const arr = ref([
     {id: 1, text: '1번 item'},
     {id: 2, text: '2번 item'},
     {id: 3, text: '3번 item'},
     {id: 4, text: '4번 item'},
     {id: 5, text: '5번 item'},
    ]);

    const small_item = computed(() => {
     return arr.value.filter((i) => i.id < 3);
    });

    const big_item = computed(() => {
     return arr.value.filter((i) => i.id >= 3);
    });

    return {
     small_item,
     big_item
    };
   }
```

```
    });

    app.mount('#app');
  </script>
</body>

</html>
```

예제2>

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>

<body>
  <script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>
  <div id="app">
    <input v-model="message" placeholder="문자를 입력하세요" />
    <p>원본: {{ message }}</p>
    <p>대문자: {{ upperMessage }}</p>
  </div>

  <script>
    const { createApp, ref, computed } = Vue;

    const app = createApp({
      setup() {
        const message = ref('');

        const upperMessage = computed(() => {
          return message.value.toUpperCase();
        });
```

```
      return {
        message,
        upperMessage
      };
    }
  });

  app.mount('#app');
 </script>
</body>
</html>
```

예제3>

```
<!DOCTYPE html>
<html lang="en">

<head>
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width, initial-scale=1.0">
 <title>Document</title>
</head>

<body>
 <script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>

 <div id="app">
  <h2>간단한 계산기</h2>
  <input type="number" v-model.number="num1" placeholder="첫 번째 숫자">
  <input type="number" v-model.number="num2" placeholder="두 번째 숫자"

  <p>합: {{ sum }}</p>
  <p>차: {{ difference }}</p>
 </div>

 <script>
```

```
    const { createApp, ref, computed } = Vue;

    const app = createApp({
      setup() {
        const num1 = ref(0);
        const num2 = ref(0);

        const sum = computed(() => num1.value + num2.value);
        const difference = computed(() => num1.value - num2.value);

        return {
          num1,
          num2,
          sum,
          difference
        };
      }
    });

    app.mount('#app');
  </script>
</body>
</html>
```

computed는 단순하게 데이터의 변환을 위해 사용되는 것 뿐만 아니라,

성능 최적화에도 중요한 역할을 한다. computed의 기본 특성은 캐싱이다.

즉, 의존하는 데이터가 변경되기 전까지는 계산된 값을 재사용이 가능함으로 불필요한 연산을 방지할 수 있다는 장점이 있다.

이게 무슨 의미인지 뒤에 나오는 method랑 비교하면서 다시 한번 살펴 보자.

methods

methods는 주로 동작이나 메서드를 처리하는 데 사용된다.

예를 들어, 버튼 클릭, 데이터 수정 등의 동작을 처리 할 때 사용한다.

방금 학습한 computed 와 비교를 하자면

computed는 계산된 값을 반환을 할 때 사용 하지만,

mtehod는 데이터를 계산하는 것이 아니라 동작을 처리할 때 사용한다.

아래 예시코드를 직접 작성 해보자.

버튼을 클릭할 때 데이터를 변경하고 화면에 그 결과를 보여주는 예제다.

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>

<body>
  <script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>
  <div id="app">

    <h2>Items List</h2>
    <p v-for="item in items" :key="item.id">{{ item.text }}</p>

    <button @click="addItem">아이템 추가버튼</button>
    <button @click="removeItem">아이템 제거버튼</button>

    <h3>Number of Items: {{ itemCount }}</h3>
  </div>

  <script>
    const { createApp, ref, computed } = Vue;  <!-- computed를 임포트하셔요 -->

    const app = createApp({
      setup() {
        const items = ref([
          { id: 1, text: 'Item 1' },
          { id: 2, text: 'Item 2' },
          { id: 3, text: 'Item 3' },
```

```
      ]);

      // 아이템을 증가 시키는 함수
      const addItem = () => {
        const newId = items.value.length + 1;
        items.value.push({ id: newId, text: `Item ${newId}` });
      };

      // 아이템을 제거하는 함수
      const removeItem = () => {
        items.value.pop();
      };

      // 아이템 개수를 계산하는 computed
      const itemCount = computed(() => items.value.length);

      return {
        items,
        addItem,
        removeItem,
        itemCount,
      };
    },
  });

  app.mount('#app');
 </script>
</body>

</html>
```

코드를 확인해 보면 addItem 그리고 removeItem 함수는 computed를 사용하지 않았다.

함수에 computed가 없으다면 그 함수는 그냥 method 이다.

위에서 아이템 카운트는 computed를 사용했다.

```
const itemCount = computed(() => items.value.length);
```

만약에 computed를 사용하지 않고 method로 바꾼다면 어떨까?

```
// itemCount를 method로 변경

const itemCount = () => items.value.length;
```

computed는 의존하고 있는 값을 캐싱을 통해 불필요한 계산을 방지하는 반면,

method는 매번 호출 될 때 마다 재실행 된다.

즉, method를 사용하면 itemCount()를 호출 할 때 마다 매번 item.value.length를 계산하게 되므로

실행 결과는 같다고 느낄 수 있지만 최적화가 되어있지 않다고 말할 수 있다.

computed 그리고 method의 차이점은 분명하다.

- computed 의 특징
  1. computed는 의존하고 있는 상태 데이터의 값을 계산 할 때 사용된다.
  2. 반응형 데이터에 의존하며, 데이터가 변경될 때만 계산을 다시 한다. 이 말은 의존성 추적을 틍해 관련된 데이터가 바뀌지 않으면 다시 계산하지 않으므로 성능 최적화에 유리하다.
  3. 계산되 값은 캐싱이 된다. 따라서 한 번 계산된 값은 의존하는 데이터가 변경될 때까지 재계산하지 않기 때문에 불필요한 연산을 방지할 수 있다.
  4. 읽기 전용값이다. 따라서 데이터를 변경할 때는 computed를 사용하지 않는다.

- method의 특징
  1. method는 동작을 처리하는 함수다. 주로 버튼 클릭과 같은 이벤트나 상태를 변경할 때 사용한다.
  2. method는 매번 호출 될 때 마다 실행이 된다. 따라서 의존하는 데이터가 변경되었든 아니든, 호출될 때마다 항상 실행되므로 최적화와 거리가 있다.
  3. method는 데이터의 읽기 뿐만 아니라 쓰기도 담당한다. 따라서 computed와 달리 methods는 의존하고 있는 데이터의 값의 변경도 처리할 수 있다.

[결론]

따라서 읽기 전용으로 계산된 값을 반환 할 것이라면 computed를 사용하면 되고

이벤트 등의 동작을 처리할 때에는 method를 사용하면 된다.

watch

watch는 ref 객체와 마찬가지로 데이터의 변화를 감시하고 데이터가 변경 될 때마다 어떠한 작업을 추가적으로 실행 시킬 때 사용한다.

주로 데이터 변경에 반응해서 비동기 작업을 할 때 많이 사용이 된다.

아래 코드는 우리가 종종 사용한 나이를 예측해주는 agify API를 이용해서

이름을 적으면 예상되는 나이를 출력해주는 코드다.

직접 타이핑 해보고 브라우저로 결과도 확인해 보자.

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>

<body>
  <script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>
  <script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>

  <div id="app">
    <input v-model="userName" type="text" placeholder="Enter Name">
    <p>지우고 영문으로 천천히 작성: {{ userName }}</p>
    <p>예상되는 나이: {{ userAge }}</p>
  </div>

  <script>
    const { createApp, ref, watch } = Vue;

    const app = createApp({
```

```
    setup() {
      const userName = ref('');  // 초기 이름은 빈문자열
      const userAge = ref(null);     //  예측되는 나이

      // userName이 변경될 때마다 API를 호출하는 watch
      watch(userName, (newName) => {

        console.log(`${newName}`);

        axios.get(`https://api.agify.io?name=${newName}`)
          .then(response => {
            // 예측된 나이를 userAge에 저장
            userAge.value = response.data.age;
          })
          .catch(error => {
            console.error('삐삐빅 에러:', error);
            userAge.value = null;
          });
      });

      return {
        userName,
        userAge
      };
    }
  });

  app.mount("#app");
</script>
</body>

</html>
```

watch를 이용해서 userName이라는 변수의 값이 바뀔 때 마다

Agify API를 호출해서 해당 이른에 대한 예측나이를 출력했다.

watch를 더 쉽게 이해하기 위해서 감시카메라로 비유 할 수 있다.

카메라(watch)는 목표(userName)을 계속 감시하고 있다가, 목표가 움직일 때 마다

(userName이 바뀔 때 마다)

자동으로 알림을 보낸다. 그 알림에 따라 API 요청이 이루어 지고 결과는 화면에 실시간으로 업데이트 되는 것이다.

wach의 동작 원리는 다음과 같다.

- watch(감시대상, callback함수)
  - 감시대상인 userName을 감시하다가 값이 변경이 되면 자동으로 callback 함수 실행
- axios.get()
  - callback 함수 안에서 agify API를 호출해서 예측되는 나이를 응답받는다.
- userAge.value = response.data.age를 이용해서 userAge에 저장한다.

끝으로 watch와 computed의 차이점에 대해서 살펴보고 마무리 하자.

의외로 쉽고 간단하다.

- computed는 값이 변경될 때마다 계산된 값을 자동으로 업데이트 한다. 그런데 주로 **화면에 표시할 값을 계산할 때 사용**한다.
- watch는 **데이터 변경에 반응해서 어떤 작업을 실행할 때 사용**한다. 예를 들어, 데이터가 변경될 때 API 호출, 로그 출력, 다른 상태 업데이트 등을 처리할 수 있다.

Lifecycle hooks (`onMounted`)

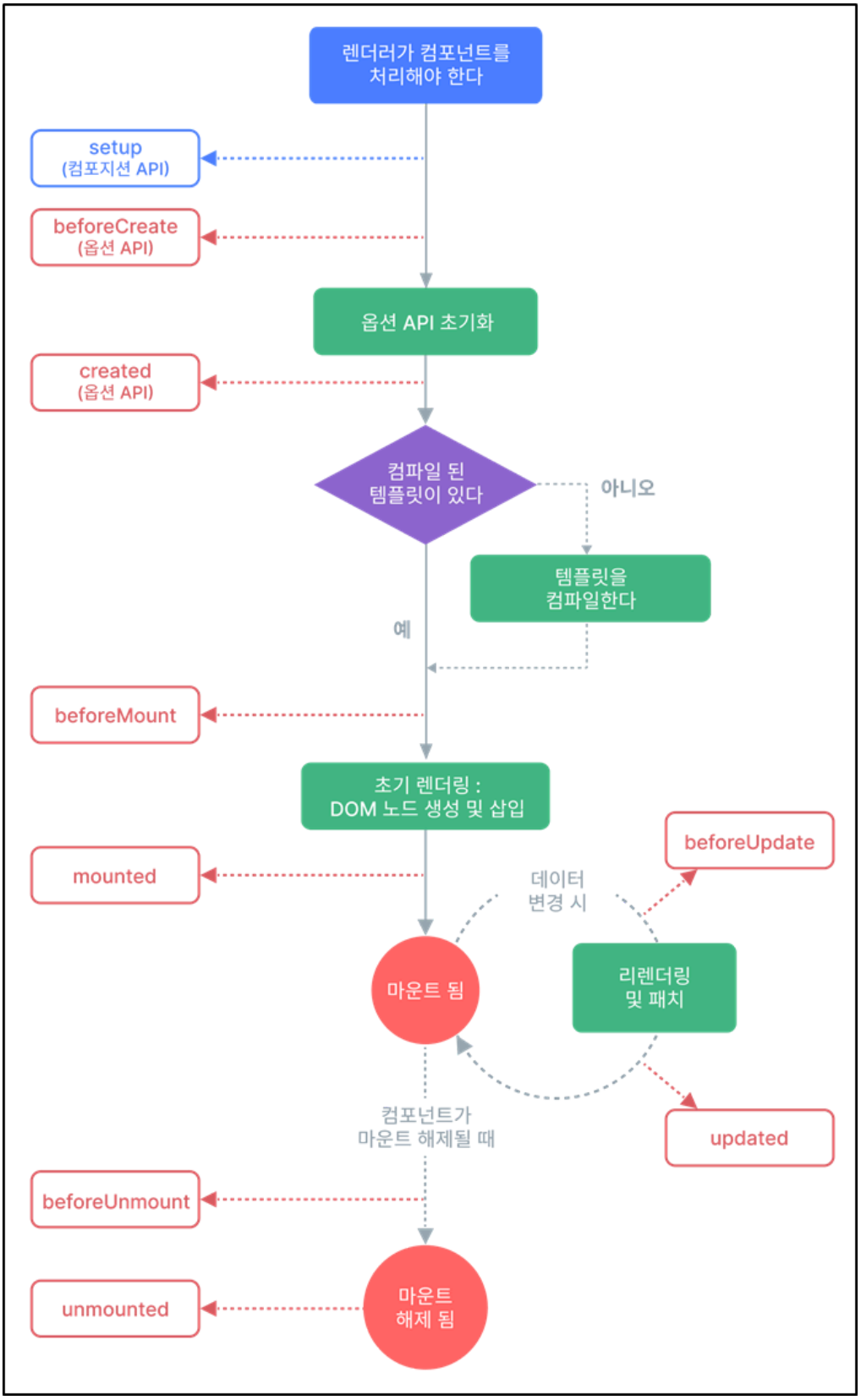
Vue.js 에선 컴포넌트의 생명주기에 따라 자동으로 실행되는 함수들이 존재하며, 이를 lifecycle hooks 라고 한다.

쉽게 이야기 하자면 Vue.js가 실행이 되면서 우리 모니터에 표시가 되는 것들의 내부 과정을 나타낸 것이다.

아직 컴포넌트가 무엇인지는 배우지 않았다. 그래서 지금 모든것을 완벽하게 이해하기에는 무리가 있다.

그래도 눈에 자꾸 익히고 이해하려고 노력 할 필요는 있다.

Lifecycle hooks는 vue에서 컴포넌트 (화면을 구성하는 작은 단위) 가 생성, 업데이트, 소멸이 되는 과정에서 자동으로 호출되는 함수를 이야기 한다. 이를 잘 활용하면 브라우저에 랜더링 하는 특정 단계에서 자신이 의도하는 로직을 실행할 수 있도록 도와준다는 점에서 의미가 있다.



아래 문서에서 전체 API 를 확인할 수 있다.

https://v3-docs.vuejs-korea.org/api/composition-api-lifecycle.html

이해를 못해도 괜찮다. 그래도 위 사이트 접속 후 눈으로 보면서 익숙해 지는것을 목표로 하자.

아래에 조금 더 쉽게 중요한 것을 위주로 설명을 해 놓았다.

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>

<body>
  <script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>

  <div id="app">
    <div>{{ count }}</div>
    <h1> 13기 2반 화이팅 </h1>
  </div>

  <script>
    const { createApp, ref, onBeforeMount, onMounted } = Vue;

    const app = createApp({

      setup() {
        const count = ref(0);
        onBeforeMount(() => {
          console.log('마운트 전(DOM 생성 전)', document.querySelector('h1'));
        });

        onMounted(() => {
          console.log('마운트 되어 DOM이 렌더링됨', document.querySelector('h1'));
        });

        return {
```

```
        count
      };
    }
  });
  console.log('컴포넌트 생성 전')
  app.mount('#app');

</script>
</body>

</html>
```

```
<body>
  <script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>

  <div id="app">
    <div>{{ count }}</div>
    <h1> 13기 2반 화이팅 </h1>
  </div>

  <script>
    const { createApp, ref, onBeforeMount, onMounted } = Vue;

    const app = createApp({

      setup() {
        const count = ref(0);
        onBeforeMount(() => {
          console.log('마운트 전(DOM 생성 전)', document.querySelector('h1'));
        });

        onMounted(() => {
          console.log('마운트 되어 DOM이 렌더링됨', document.querySelector('h1'));
        });

        return {
          count
        };
      }
    });
    console.log('컴포넌트 생성 전')
    app.mount('#app');

  </script>
</body>
```

가독성을 위해서 위 코드를 캡쳐해 놓았다.

컴포넌트의 생성과정은 단계가 복잡해 보이나 크게 4단계로 나눠서 살펴 볼 수 있다.

1단계: CREATE

- Create : 컴포넌트 초기화 단계
  - beforecrete - 뷰 인스턴스 초기화 직후 '생성단계'를 의미한다.
    - 즉 뷰 인스턴스가 딱 만들어 졌지만 셋팅은 덜 된 상태를 의미한다. 코드를 예를 들자면 const { createApp, ref, onMounted } = Vue 코드만 딱 실행 되었을 타이밍을 의미한다. (뷰 인스턴스가 막 생성된 단계로 setup() 함수 실행 전)
  - created - 뷰 인스턴스의 셋팅이 완료가 되었을 때를 의미하며 이때부터 data와 event가 활성화 되어 접근이 가능하다. 컴포넌트가 생성되긴 했지만 아직 화면에 붙지 않은(마운팅 되지 않은) 단계를 의미한다.
  -
- 실행 과정설명
  1. script 태그에서 Vue3 라이브러리를 불러온다.
  2. Vue 앱이 생성되고 Vue 컴포넌트를 정의한다.
     - `const { createApp, ref, onBeforeMount, onMounted } = Vue;`
  3. Vue에서의 옵션객체를 생성 후
     - `const app = createApp({`
  4. setup 함수 내에 컴포넌트 초기 상태를 설정한다.
     - `const count = ref(0)`
     - onBeforeMount() onMounted() 정의만 됨 (실행은 아직!)
  5. console.log('컴포넌트 생성 전') 로그 출력
  6. app.mount('#app'); 를 통해서 컴포넌트 인스턴스를 생성 완료

즉, CREATE 단계에서는 createApp() 메서드로 Vue 앱을 생성하고

Vue 애플리케이션을 #app 요소에 마운트메서드를 통해서

컴포넌트 인스턴스만을 생성하는 단계이다.

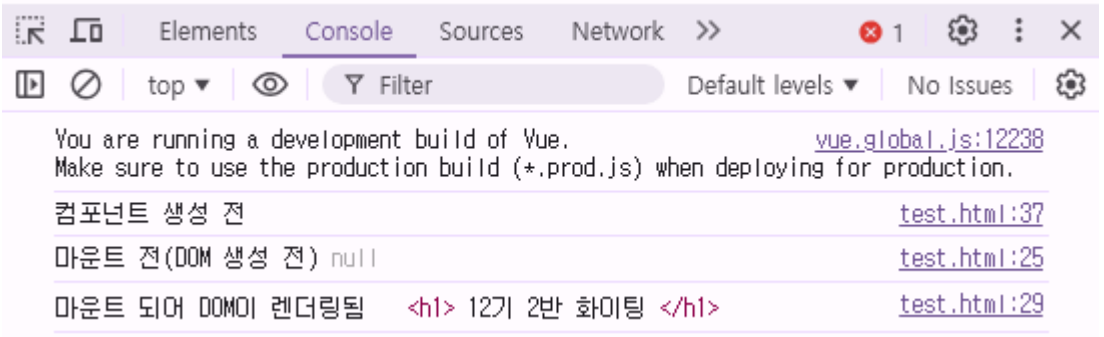
2단계: Mount

- Mounting : 돔(DOM)의 삽입 단계로, 마운트(Mount)는 DOM 객체가 생성되고 브라우저에 나타나는 것을 의미한다.

  보통 화면에 붙는다 라고 표현 한다.

  말 그대로 브라우저에 '나타나는' 것이기 때문에 유저가 직관적으로 확인할 수 있는 부분이다.

  - beforeMount - 컴포넌트가 DOM에 추가 되기 직전 단계를 의미 하는데 DOM에 마운트 되기 전에 외부에서 데이터를 가져오는 비동기 작업이 이루어 지는 단계로서 거의 의식을 하지 않고 사용하곤 한다. (onBeforeMount() 함수)
  - Mounted - 컴포넌트가 DOM에 추가 된 후 호출되는 단계로 DOM에 접근이 가능한 시점이다. 이때, Vue 컴포넌트가 실제 DOM에 추가 된 후에 DOM 요소를 선택하고 조작하는 단계에 이른다. (onMounted() 함수)



위 코드로 라이프 사이클 테스트를 해 보면

`document.querySelector('h1')` 코드가

before Mount 되기 전에는 NULL 값을 갖지만

onMount 된 후에는 h1 태그의 값이 출력 되는 것을 확인 할 수가 있을 것이다.

3단계: Update

- Update - 컴포넌트에서 사용되는 속성들이 변경되는 과정 그리고 컴포넌트가 재랜더링 되면서 실행 되는 라이프 사이클을 의미한다. 우리가 수업시간에 count 버튼을 누르면 값이 update되는것을 예를 들 수 있겠다.

4단계: Destroy

- Destroy - 컴포넌트가 제거 될 때를 의미하는 라이프 사이클 이다. destroyed 단계가 되면 컴포넌
  트의 모든 이벤트 리스너와(@click, @change 등) 디렉티브(v-model, v-show 등)의 바인딩이 해
  제 되고, 하위 컴포넌트도 모두 제거되는 단계를 말한다.

사실, 중급 개발자가 되기 전까지, lifecycle hooks 는 당장 사용할 일이 많이 없다고 봐도 무방하다.

다만 Vue 의 필수 개념 중의 하나로 분류되기 때문에 개념은 살펴보고 일단 넘어 갈 뿐이다.

Vue.js 개발자는 모든 라이프 사이클 훅을 다 알고 활용하지는 않는다.

그나마 활용성 있는 라이프 사이클을 언급하자면 `onMounted` 이다.

`onmounted` 라는건 무슨 뜻일까? 컴포넌트가 화면에 붙은 후의 시점을 의미한다.

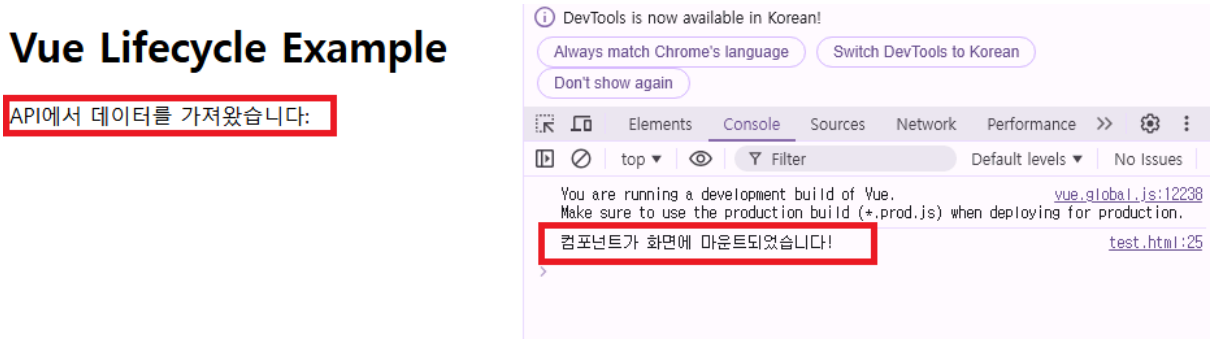
즉, `onMounted` 는 컴포넌트가 화면에 붙은 후 자동 실행되는 콜백함수라고 볼 수 있겠다.

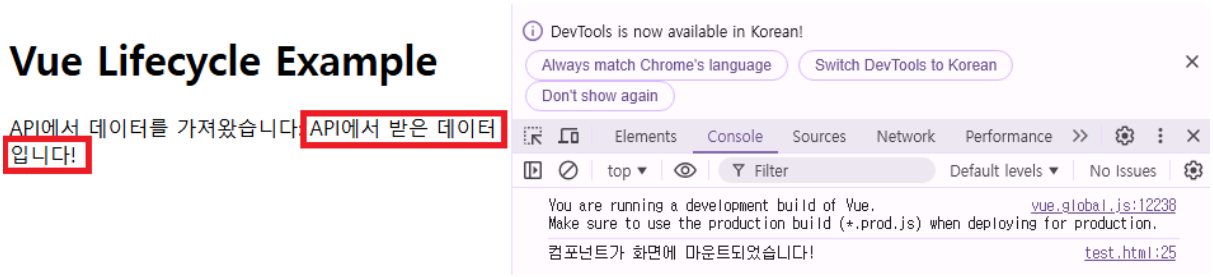
`onMounted` 훅은 컴포넌트가 화면에 붙은 이후에 실행되기 때문에, 주로 **API 호출할 때 유용하게 사용되곤 한다.** 예를 들어, 컴포넌트가 화면에 나타날 때 데이터를 가져오거나, DOM 요소를 조작할 때 `onMounted` 를 사용한다.

아래 예시 코드를 작성하고 실행 해 보자.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
```

```
<body>
  <div id="app">
    <h1>{{ message }}</h1>
    <p>API에서 데이터를 가져왔습니다: {{ apiData }}</p>
  </div>

  <script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>
  <script>
    const { createApp, ref, onMounted } = Vue;

    const app = createApp({
      setup() {
        const message = ref("Vue Lifecycle Example");
        const apiData = ref(null);

        // onMounted 훅 사용 (컴포넌트가 화면에 마운트된 후 실행)
        onMounted(() => {
          console.log("컴포넌트가 화면에 마운트되었습니다!");

          // 예시로 API 호출
          setTimeout(() => {
            apiData.value = "API에서 받은 데이터입니다!";
          }, 2000);
        });

        return {
          message,
          apiData
        };
      }
    });

    app.mount("#app");
  </script>
</body>
</html>
```





`onMounted` 훅을 사용할 때 그 목적과 이유를 좀 더 명확하게 이해하는 것이 중요하다.

`onMounted` 훅은 **컴포넌트가 화면에 마운트된 후** 실행되는 콜백 함수 라고했다. 즉, **컴포넌트가 DOM에 추가되고, 렌더링이 끝난 후에 실행된**다.

그렇다면 왜 그리고 언제 `onMounted` 훅을 사용해야 할까?

DOM이 완전히 랜터링(화면에 보여 질) 준비가 완료된 상태에서 어떠한 작업을 하고자 할 필요가 있을 때 사용하면 된다. 예를 들어, **비동기 작업을 시작하는 시점**으로 예를 들 수 있다.

개발을 하다 보면 보통 `onMounted` 시점에서 비동기 작업(예: API 호출)을 처리하는 것이 일반적인 패턴이다. 컴포넌트가 렌더링된 후에 API 호출, 데이터 로딩, 타이머 설정 등을 처리한다.

그렇다면 왜 api호출을 Dom이 랜더링 후에 api 데이터를 화면에 표시를 할까?

화면에 서버로 부터 응답받은 데이터를 보여주는 페이지를 만들 때, 컴포넌트가 마운트된 후에 API를 호출하고 데이터를 가져오면 화면이 깜빡이지 않고 사용자에게 부드럽게 전달된다.

하지만 만약에 API 호출을 컴포넌트가 마운트되기 전에 한다면, 데이터가 바뀔 때 마다 화면이 리랜더링 되면서 초기 화면이 빈 화면으로 잠깐 보일 수 있기 때문에 사용자 경험을 위해서라도 컴포넌트가 마운트 된 후에 호출한 api 데이터를 화면에 표시하는 것이 좋다.

따라서 `onMounted`에서 API 호출을 하는 이유는 바로 **컴포넌트가 화면에 마운트된 후 데이터를 받아와서 그 데이터를 반영**하기 위함이라고 할 수 있다.

끝으로 Life cycle hooks를 정리를 하자면 다음과 같다.

create - 컴포넌트가 DOM에 추가되기 전

mount - 컴포넌트가 DOM에 삽입되고 웹 브라우저에 나타남

update - 컴포넌트 반응형 속성들이 변경되거나 재 랜더링될때

destroy - Vue 인스턴스 제거될때

각 사이클의 명칭으로도 대략 '아 어떤시점이구나~' 하고 이해할 수 있으면 된다.

<끝>