5.Vue component

1-1. 컴포넌트 정의 및 등록

1. 컴포넌트 정의

1) Single-File Component(SFC)를 사용하는 방법

- 실무에서 일반적으로 사용하는 방법
- 빌드 도구를 사용할 때 컴포넌트는 일반적으로 **Single-File Component(SFC)**로 정의

2) 문자열 템플릿(string template)으로 정의하는 방법

- 빌드 도구를 사용하지 않을 때 컴포넌트는 Vue 옵션인을 포함하는 일반 JavaScript 객체로 정의

2. 컴포넌트 등록

1) 등록 : Vue 컴포넌트는 <template>안에서 발견 되었을 때 Vue가 구현 위치를 알 수 있도록 "등록

2) 컴포넌트를 등록하는 방법

① 전역 등록(Global Registration)

- **app.component() 메서드**를 사용하여 현재 Vue 애플리케이션에서 전역적으로 사용

- 문제점

[1] Webpack(또는 Vite)과 같은 빌드 시스템을 사용하는 경우 컴포넌트를 전역 등록하는 것은 컴포넌트를 사용하지 않더라도 계속해서 최종 빌드에 해당 컴포넌트가 포함

[2] 전역 등록을 계속 하게 되면 애플리케이션의 컴포넌트간 종속 관계를 확인 어려움

② 지역 등록(Local Registration)

- 지역 등록된 컴포넌트는 현재 컴포넌트 영역 안에서만 사용

1-2-1. 컴포넌트 정의 및 등록

1. Single-File Component(SFC) 구성
  - SFC는 확장자 *.vue를 가진 단일 파일

```
<template>
  <button @click="counter++">클릭 횟수 {{ counter }}</button>
</template>

<script>
import { ref } from 'vue';

export default {
  setup() {
    const counter = ref(0);
    return {
      counter,
    };
  },
};
</script>

<style></style>
```

1-2-1. 문자열 템플릿 (string template)

1. 빌드 도구를 사용하지 않을 때 컴포넌트는 Vue 옵션인을 포함하는 일반 JavaScript 객체로 정의

```
import { ref } from 'vue/dist/vue.esm-bundler.js';
export default {
  setup() {
    const counter = ref(0);
    return {
      counter,
    };
  },
  template: `
                <button @click="counter++">클릭 횟수 {{ counter }}</button>
  `,
};
```

1-3-1. 컴포넌트 등록

1. Vue 컴포넌트는 <template>안에서 발견 되었을 때 Vue가 구현 위치를 알 수 있도록 "등록"
 - 컴포넌트를 등록하는 방법는 전역(Global) 및 지역(Local) 두 가지

 1) 전역 등록
 ① 'app.component()' 메서드를 사용하여 현재 Vue 애플리케이션에서 전역적으로 사용

```
import { createApp } from 'vue';
import App from './App.vue';

import GlobalComponent from './components/GlobalComponent.vue';

const app = createApp(App)
app.component('GlobalComponent', GlobalComponent)
app.mount('#app');
```

 ② app.component() 메서드는 다음과 같이 연결(메서드 체인)
 - .component('ComponentA', ComponentA)

 ③ 전역 등록된 컴포넌트는 애플리케이션 어떤 곳에서든 사용 가능

1-3-2. 컴포넌트 등록

2) 지역 등록
- 지역 등록된 컴포넌트는 **현재 컴포넌트 영역 안에서만 사용**
- Vue 컴포넌트 인스턴스의 components 옵션을 사용해서 등록

① // ParentComponent.vue 파일

```
import ChildComponent from './ChildComponent.vue'

export default {
        components: {
                ChildComponent
        },
        setup() {

                // ...
        }
}
```

ParentComponent 컴포넌트에 로컬 등록된 ChildComponent는 현재 컴포넌트인 ParentComponent 컴포넌트에서만 사용 가능

② 컴포넌트 사용

등록된 컴포넌트는 <template>에서 원하는 만큼 사용

```
<h2>Single-File Component</h2>
<ButtonCounter></ButtonCounter>
<ButtonCounter></ButtonCounter>
<ButtonCounter></ButtonCounter>
```

③ PascalCase
- Single-File Commponent(SFC)에서는 기본 HTML요소와 구분하기 위해 자식 컴포넌트에 PascalCase 이름 사용 권장
- 기본 HTML 태그 이름은 대소문자를 수분하지 않지만 Vue SFC는 컴파일된 형식이므로 대소문자 구분 태그 이름을 사용
- 모든 단어를 띄어 쓰기 없이 대문자로 시작 예시) VisitorName

1-4-1. Props

1. Props 개념
 - **컴포넌트에 등록할 수 있는 사용자 정의 속성**. 블로그 게시글 컴포넌트에 사용자 정의 속성을 선언하면 이 컴포넌트를 사용하는 부모 컴포넌트에서 데이터(속성)를 전달

2. Props 선언
 - 모든 prop들은 부모와 자식 사이에 **단방향으로 내려가는 바인딩** 형태
 - 반대 방향으로는 전달되지 않음 (사용자가 prop을 자식 컴포넌트 안에서 수정해서는 안된다는 것)
 - Vue 컴포넌트에는 명시적 props 선언이 필요. 왜냐하면 컴포넌트에 전달된 외부 props가 fallthrough 속성으로 처리되어야 함
 - **fallthrough 속성 :** props 또는 emits에 명시적으로 선언되지 않은 속성 또는 이벤트 컴포넌트에 전달되는 속성 또는 v-on 이벤트 리스너 이지만, 이것을 받는 컴포넌트의 props 또는 emits에서 명시적으로 선언되지 않은 속성(일반적인 예로는 class , style , id 속성)

3. 문자열 배열 선언
 - 컴포넌트에 props 옵션을 사용하여 선언

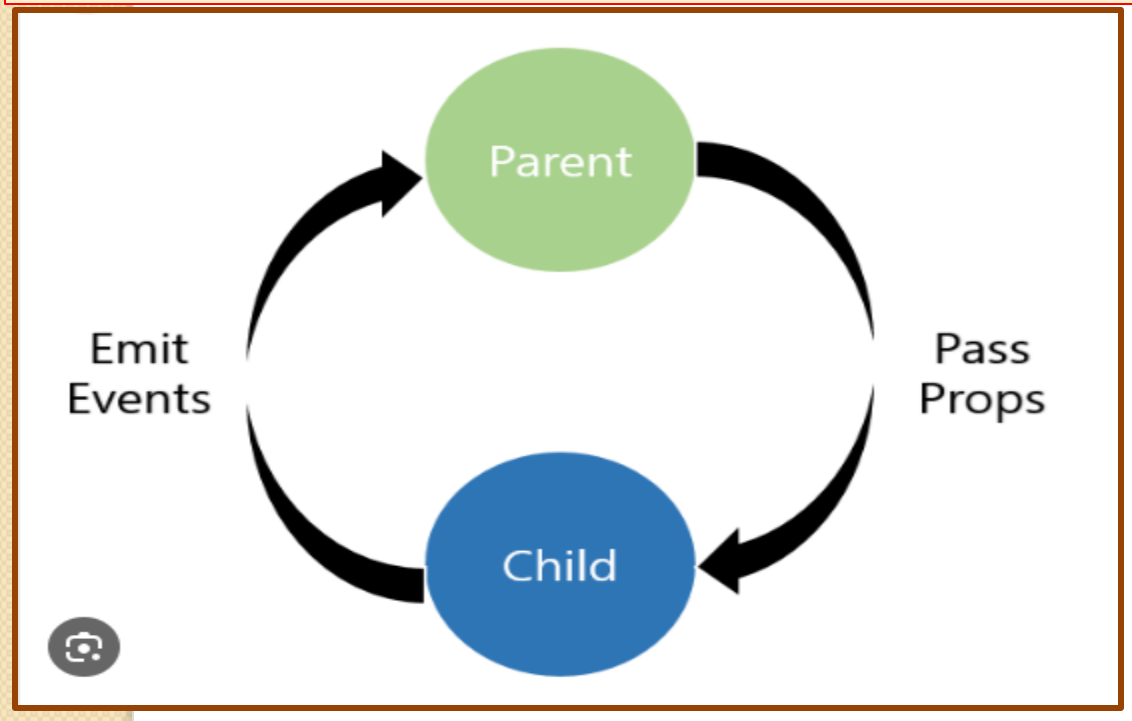
```
// BlogPost.vue
export default {
  props: ['title'],
  setup(props) {
    console.log(props.title)
  }
}
```

3. 객체문법 선언
 - 문자열 배열을 사용하여 props를 선언하는 것 외에도 객체 문법을 사용하여 속성 타입과 함께 선언
 - 문자열 배열을 사용하여 `props`를 선언하는 것 외에도 객체 문법을 사용하여 속성 타입과 함께 선언

1-4-2. Props

```
export default {
  props: {
    title: String,
    likes: Number
  },
                setup(props) {
                   console.log(props.title)
                   console.log(props.likes)
                }
}
```

props 선언시 key는 속성명이고 value는 속성 타입. 더 자세히 선언하고 싶다면 value에 고급 옵션인 객체를 선언



1-4-3. Props

props 선언시 key는 속성명이고 value는 속성 타입. 더 자세히 선언하고 싶다면 value에 고급 옵션인 객체를 선언

```
{
  // Basic type check
  //  (`null` and `undefined` values will allow any type)
  propA: Number,
  // Multiple possible types
  propB: [String, Number],
  // Required string
  propC: {
    type: String,
    required: true
  },
  // Number with a default value
  propD: {
    type: Number,
    default: 100
  },
  // Object with a default value
  propE: {
    type: Object,
    // Object or array defaults must be returned from
    // a factory function
    default() {
      return { message: 'hello' }
    }
  },
```

1-4-4. Props

props 선언시 key는 속성명이고 value는 속성 타입. 더 자세히 선언하고 싶다면 value에 고급 옵션인 객체를 선언

```
// Custom validator function
  propF: {
    validator(value) {
      // The value must match one of these strings
      return ['success', 'warning', 'danger'].includes(value)
    }
  },
  // Function with a default value
  propG: {
    type: Function,
    // Unlike object or array default, this is not a factory function - this is a function to serve as a default value
    default() {
      return 'Default function'
    }
  }
}
```

- 컴포넌트 사용시 type, required, validator 명시된 사항을 위반할 때 개발모드에서 콘솔 경고가 발생

항목	주요내용
type	String, Number, Boolean, Array, Object, Date, Function, Symbol 모든 기본 생성자 또는 모든 사용자 정의 타입이 가능 . (예: Person, Animal) [Number, String] 배열을 이용하여 여러개의 타입을 선언 가능
default	속성값이 비어 있거나 undefined를 전달 받는 경우 기본값을 선언 객체 또는 배열 타입인 경우 기본값을 팩토리 함수를 사용하여 반환
required	속성이 필수값이라면 true로 해서 설정
validator	속성값의 유효성 검사가 필요할 때 사용

1-4-5. Props 사용

① 선언된 props를 <template>에서 바로 사용

```
<template>
        <p>{{ title }}</p>
</template>
```

② setup() 함수의 첫 번째 매개변수로 props 객체를 받아 사용

```
export default {
        setup(props) {
                return { };
        },
};
```

③ 컴포넌트 인스턴스(this)의 $props 객체로 접근(Options API)

```
<template>
        <p>{{ $props }}</p>
</template>
<script>
export default {
    created() {
      // 객체로 접근
      this.$props
      //
      this.title
    }
};
</script>
```

1-4-6. Props Name Casing

① props 선언시에는 camelCase를 사용하여 이름을 선언합니다. 이렇게 하면 속성 키로 사용할 때 따옴표를 사용할 필요가 없고 유효한 JavaScript 식별자이기 때문에 템플릿 표현식에서 직접 참조

```
export default {
  props: {
    greetingMessage: String
  }
}

<span>{{ greetingMessage }}</span>
```

② 속성에 값을 전달할 때는 kebab-case를 사용하는 것을 권장

```
<MyComponent greeting-message="hello"></MyComponent>
```

유형	Case	적용후
Camel case	USER LOGIN LOG	userLoginLog
Pascal case	USER LOGIN LOG	UserLoginLog
Kebab case	USER LOGIN LOG	user-login-log
Snake case	USER LOGIN LOG	user_login_log 또는 USER_LOGIN_LOG

1-4-7. 객체를 사용하여 다중 속성 전달

① 객체의 모든 속성을 props로 전달하려는 경우 v-bind에 전달인자(예, v-bind:prop-name)없이 사용 가능

```
export default {
    setup() {
                const post = ref({
                            id: 1,
                            title: 'Learn Vue3'
                })
                return {
                            post
                }
    }
}
<span>{{ greetingMessage }}</span>
```

② 두 가지 전달 방법은 동일

```
    < <!-- 객체를 사용한 다중 속성 전달(전달인자 없음) -->
    <BlogPost v-bind="post" />
    <BlogPost :id="post.id" :title="post.title" />
```

- 모든 props는 상위 속성과 하위 속성간에 **단방향 바인딩**으로 형성
  만약 **상위 속성**이 **업데이트**되면 **하위 속성**도 **업데이트**되지만 그 반대는 아님

```
export default {
    props: ['title'],
    setup(props) {
        // ✕ warning, props are readonly!
        props.title = 'changed title';
    }
 }
```

1-4-8. 단방향 데이터 흐름

[1] 두 가지 전달 방법은 동일

① prop은 **초기 값을 전달하는 데 사용**됩니다. 자식 컴포넌트에서 속성 값을 로컬 데이터 속성으로 사용 하고자 할 때 prop을 초기 값으로 사용하는 로컬 변수를 선언하는 것이 가장 좋음

```
export default {
        props: ['initialWidth', 'initialHeight'],
        setup(props) {
                // width는 props.initialWidth 값으로 초기화 됩니다.
                // 향후 props 업데이트의 연결이 끊어집니다.
                const width = ref(props.initialWidth)
                const height = ref(props.initialHeight)
                return {
                        width,
                        height
                }
        }
}
```

② **prop**의 값의 변환이 필요할 때 , 이 경우 computed를 사용하면 좋음 .
그리고 상위 속성의 변경을 유지가능

```
export default {
        props: ['size'],
        setup(props) {
            // 향후 props 업데이트의 연결이 유지.
            const rectangleSize = computed(() => props.size.trim().toUpperCase());
            return {
                rectangleSize
            }
        }
}
```

- computed 속성은 종속 대상을 따라 저장(캐싱),
- computed 속성은 해당 속성이 종속된 대상이 변경될 때만 함수를 실행

1-4-9. 객체 / 배열 Props 업데이트

[1] 객체(object)나 배열(array)이 props로 전달되면 자식 컴포넌트에서는 prop 바인딩(값 변경)을 변경할 수 없지만 객체 또는 배열의 중첩 속성은 변경가능

[2] JavaScript에서 객체와 배열이 참조 타입(Reference Type)으로 전달되고 Vue가 이러한 변경을 방지하는것은 부당한 비용 발생가능

[3] 만약 변경이 필요하다면 자식 컴포넌트에서 **emit을 이용하여 부모 컴포넌트가 스스로 변경을 수행할 수 있도록** 이벤트를 적용

1-5-1. Events

[1] Events 개념
- 자식 컴포넌트에서도 부모 컴포넌트로 데이터를 전달 또는 트리거의 목적으로 이벤트를 내보는것
- 이벤트는 컴포넌트의 emit 메서드를 통하여 발생

[2] 이벤트 발생 및 수신
① 컴포넌트의 <template> 블록 안에서 내장 함수 $emit()을 사용하여 직접 커스텀한 이벤트를 내보낼 수 있음

```
<template>
    <button @click="$emit('someEvent')">버튼</button>
</template>
```

② 그러면 부모 컴포넌트에서 **v-on(또는 @)을 사용하여 이벤트를 수신**

```
<MyComponent @some-event="callFunction" />
```

③ .once 수식어는 컴포넌트 커스텀 이벤트에서도 지원

```
<MyComponent @some-event.once="callFunction" />
```

[3] 이벤트 파라미터
① 이벤트와 함께 특정 값 전달 , **$emit 함수 이벤트명에 추가로 파라미터를 전달**

```
<template>
        <button @click="$emit('someEvent', 'Hello', 'World', '!')">버튼</button>
</template>
```

② 그런다음 부모 컴포넌트에서 이벤트와 함께 파라미터를 받음

```
<template>  <MyComponent @some-event="callFunction" /></template>

<script setup>
export default {
        setup() {
            const callFunction = (word1, word2, word3) => {
                alert(word1, word2, word3);
            };
            return {
                callFunction
            }
        }
}
</script>
```

1-5-2. Events (v-model)

[1] v-model 만들기
- 컴포넌트를 만든후 해당 Component에 v-model을 적용하려면 **@update:modelvalue** Event를 사용하여 v-model 생성

[2] 우리가 만든 Component는 아래와 같이 수행

```
<input
    :modelvalue = "username"
    @update:modelValue = "newValue => username = newValue"
/>
```

1-6-1. Non-Prop 속성

[1] Non-Prop 속성 (fallthrough 속성)
- **Non-Prop 속성**은 props 또는 event 에 명시적으로 선언되지 않은 속성 또는 이벤트
- 예시 : class, style, id

[2] 속성 상속
- 컴포넌트가 단일 루트 요소로 구성되어 있으면 Non-Prop **속성**은 루트 요소의 속성에 자동으로 추가

① 컴포넌트를 사용하는 부모 컴포넌트
```
<MyButton class="large" />
```
② 루트 요소의 속성을 자동으로 추가하는 자식 컴포넌트
```
<!-- template of <MyButton> -->
<button>click me</button>
```
③ 최종 렌더링된 DOM
```
<button class="large">click me</button>
```

[3] class, style 속성 병합

① 자식 컴포넌트 루트요소에 이미 class와 style속성이 정의되어 있으면, 부모로 받은 class와 style속성과 병합
```
<!-- template of <MyButton> -->
<button class="btn">click me</button>
```
② 병합된 DOM
```
<button class="btn large">click me</button>
```

[4] v-on 이벤트 리스너 상속

① `<MyButton @click="onClick" />`
@click 리스너는 <MyButton>의 컴포넌트 루트요소인 <button>요소에 추가

② <button>요소에 이미 바인딩된 이벤트가 있다면 이벤트가 추가되어 두 리스너 모두 트리거

1-6-2. Non-Prop 속성

[5] 속성 상속 비활성화

① 컴포넌트가 자동으로 Non-Prop **속성**을 상속하지 않도록 하려면 컴포넌트의 inheritAttrs: false 옵션을 설정

```
<template>
    <button class="btn" data-link="hello">click me</button>
</template>
<script>
export default {
    inheritAttrs: false,
};
</script>
```

② 적용해야 하는 요소에 <template>에서 Non-Prop속성에 접근할 수 있는 내장 객체 $attrs로 직접 접근

```
<template>
   <p>Non-Prop 속성: {{ $attrs }}</p>
</template>
```

③ **$attrs** 객체에는 컴포넌트에 선언되지 않은 모든 속성 props, emits (예: class, style, v-on 등)을 포함

- `props`와 달리 **Non-Prop 속성**은 JavaScript에서 원래 대소문자를 유지하므로 `foo-bar`와 같은 속성은 `$attrs['foo-bar']`로 접근.
- `@click`과 같은 `v-on`리스너는 `$attrs.onClick`과 같이 함수로 접근

[6] Non-Prop 속성을 특정 요소에 모두 적용

- inheritAttrs: false와 $attrs를 이용하면 **Non-Prop 속성**을 특정 요소에 모두 적용

```
<template>
  <label>
    이름:
    <input type="text" v-bind="$attrs" />
  </label>
</template>
<script>
export default {
  inheritAttrs: false,
};
</script>
```

1-6-3. Non-Prop 속성

[7] 다중 루트 노드(multiple root node)
① 3.x 에서 컴포넌트는 다중 루트 노드(multiple root node)를 가질 수 있음
개발자가속성을 배포(상속)해야 하는 위치를 명시적으로 정의
② 예시

```
<!-- Layout.vue -->
<template>
  <header>...</header>
  <main v-bind="$attrs">...</main>
  <footer>...</footer>
</template>
```

[8] 여러 루트노드의 속성 상속
① **단일 루트 요소**가 있는 컴포넌트와 달리 **여러 루트 요소**가 있는 컴포넌트에는 자동으로 **Non-Prop 속성**이 상속되지 않음 . 만약 명시적으로 $attrs를 바인딩 하지 않을 경우 런타입 경고가 발생

② **자식 컴포넌트**

```
<!-- CustomLayout.vue -->
<template>
  <header></header>
  <main></main>
  <footer></footer>
</template>
```

③ **부모 컴포넌트**

```
<CustomLayout id="custom-layout"></CustomLayout>
```

$attrs이 명시적으로 바인딩된 경우 경고가 표시되지 않음

1-6-4. Non-Prop 속성

[8] 여러 루트노드의 속성 상속

④

```
<!-- CustomLayout.vue -->
  <template>
    <header></header>
    <main v-bind="$attrs"></main>
    <footer></footer>
  </template>
```

⑤ JavaScript에서 Non-Prop 속성 접근

setup() 함수의 context.attrs 속성으로 노출

```
export default {
  setup(props, context) {
    console.log(context.attrs)
  }
}
```

1-7-1. Slots

[1] Slot 개념
HTML 요소와 마찬가지로 **우리가 만든 컴포넌트**에 **콘텐츠를 전달**할 수 있으면 유용
[2] slot 사용 예시
① <FancyButton> 컴포넌트를 만든 후 콘텐츠를 전달

```
<!-- FancyButton.vue -->
<template>
    <button class="fancy-btn">
        <slot></slot>
    </button>
</template>
```

② Style
위에 정의한 컴포넌트를 부모 컴포넌트에서 사용

```
<FancyButton>
    <!-- 슬롯 콘텐츠 -->
    Click!!
</FancyButton>
```

③ **<slot> 요소는 부모 컴포넌트에서 제공하는 콘텐츠**를 나타내는 슬롯 콘텐츠 .
슬롯은 텍스트 뿐만 아니라 HTML요소, 컴포넌트 등 다양한 모든 콘텐츠

```
<FancyButton>
    <!-- 슬롯 콘텐츠 -->
  <span style="color: red">Click me</span>
  <i>!</i>
</FancyButton>
```

[3] **Fallback Content**
상위 컴포넌트에서 슬롯 콘텐츠가 제공되지 않을때 슬롯에 대한 폴백(기본 콘텐츠)을 지정

```
<!-- FancyButton.vue -->
<template>
  <button class="btn">
    <slot>Default Click!!</slot>
  </button>
</template
```

1-7-2. Slots

[4] Named Slots
  <slot> 요소에 이름을 부여하여 여러개의 <slot>을 정의

```
<!-- BaseCard.vue -->
<template>
  <article>
    <div>
      <slot name="header"></slot>
    </div>
    <div>
      <slot></slot>
    </div>
    <div">
      <slot name="footer"></slot>
    </div>
  </article>
</template>
```

①<slot>에 name속성을 부여하여 특정 slot 콘텐츠가 렌더링 되어야 할 위치를 설정
② **name이 없는** <slot>의 이름은 **암시적으로 default**

```
<!-- 부모 컴포넌트 사용 예시 -->
<template>
  <BaseCard>
    <template v-slot:header>제목</template>
    <template v-slot:default>안녕하세요</template>
    <template v-slot:footer>푸터</template>
  </BaseCard>
</template>
```

name이 부여된 <slot>에 콘텐츠를 전달하려면 **v-slot 디렉티브**를 사용하여 전달.
그리고 v-slot:전달인자를 사용하여 지정한 **슬롯 콘텐츠**에 전달

1-7-3. Slots

[5] default 슬롯
  default 슬롯은 **암시적**으로 처리

```
<!-- 부모 컴포넌트 사용 예시 -->
<template>
  <BaseCard>
    <template #header>제목</template>
    <!-- 암시적으로 default slot -->
     안녕하세요
    <template #footer>푸터</template>
  </BaseCard>
</template>
```

[6] Dynamic Slot Named
  - v-slot 디렉티브 전달인자에 데이터를 바인딩하여 동적으로 변경

```
<BaseCard>
  <template v-slot:[dynamicSlotName]>
    ...
  </template>

  <!-- with shorthand -->
  <template #[dynamicSlotName]>
    ...
  </template>
</BaseCard>
```

1-7-4. Slots

[7] Render Scope
슬롯 콘텐츠는 상위 컴포넌트에 정의되어 있으므로 상위 컴포넌트의 데이터 영역에 접근은 가능하지만 **하위 컴포넌트의 영역**에는 **접근불가**

[8] Scoped Slots
- Render Scope에서 언급했던 것처럼 슬롯 콘텐츠는 자식 컴포넌트의 데이터에 접근할 수 없음
- 하지만 슬롯 콘텐츠에서 상위 컴포넌트와 하위 컴포넌트 데이터를 모두 사용할 수 있다면 우리는 개발할 때 매우 유용.
- 이러한 방법으로 우리는 자식 컴포넌트에서 `<slot>` 요소를 사용할 때 props를 전달하는 것처럼 속성을 슬롯 콘텐츠에 전달

```
<!-- MyComponent.vue -->
<template>
  <div>
    <slot :text="greetingMessage" :count="count"></slot>
  </div>
</template>
<script>
import { ref } from 'vue';

export default {
  setup() {
    const greetingMessage = ref('Hello World!');
    const count = ref(1);
    return {
      greetingMessage,
      count,
    };
  },
};
</script>
```

1-7-5. Slots

[9] v-slot 디렉티브

① default <slot>이 **하나 밖에 없는 경우**에는 v-slot 디렉티브를 사용하여 props를 전달 가능

```
<MyComponent v-slot="slotProps">
    {{ slotProps.text }} {{ slotProps.count }}
</MyComponent>
```

② 구조분해할당 문법으로 더 사용하기 편리

```
<MyComponent v-slot="{ text, count }">
    {{ text }} {{ count }}
</MyComponent>
```

[10] Named Scoped Slots

- 이름이 부여된 슬롯도 유사하게 작동. v-slot:name="slotProps"

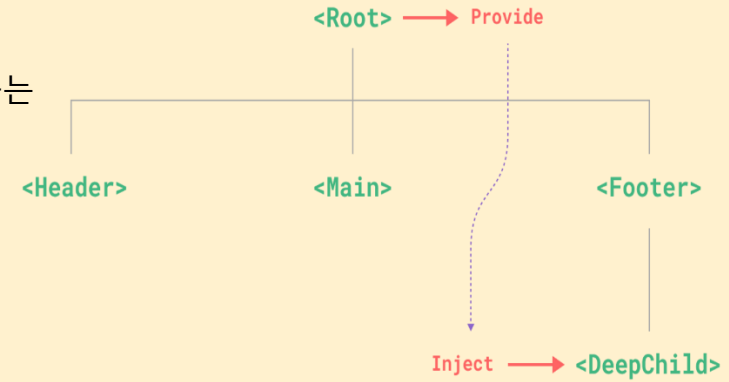
```
<MyComponent>
  <template #header="headerProps">
    {{ headerProps }}
  </template>

  <template #default="defaultProps">
    {{ defaultProps }}
  </template>

  <template #footer="footerProps">
    {{ footerProps }}
  </template>
</MyComponent>
```

1-8-1. Provide / Inject

[1] Prop Drilling
① 일반적을 부모 컴포넌트에서 자식 컴포넌트로 데이터를 전달해야 할 때 **props**를 사용
② 해당 자손 컴포넌트와 연관된 모든 자식 컨포넌트에게 동일한 prop을 전달
③ <Root>에서 <DeepChild> 컴포넌트에 데이터를 전달하기 위해서는 <Footer> 컴포넌트를 거쳐 데이터를 전달
Prop Drilling" 문제는 Vue3의 provide와 inject로 해결



[2] Provide
- 하위 컴포넌트 항목에 데이터를 제공하려면 **provider** 역할을 하는 상위 컴포넌트 setup() 함수 사용

```
<import { provide } from 'vue';

export default {
  setup() {
    provide('message', 'hello!');
  },
};
```

provide() 함수는 두 개의 파라미터
① 첫 번째 파라미터는 **주입 키** : 문자열 또는 Symbol이 될 수 있으며, **주입 키**는 하위 컴포넌트에서 주입된 값을 조회하는 데 사용
② 두 번째 파라미터는 **제공된 값** : 값은 refs와 같은 반응성 데이터를 포함하여 모든 유형이 될수 있음

```
import { provide, ref } from 'vue';

export default {
  setup() {
    const message = ref('Hello World!');
    provide('message', message);
    return {
      message,
    };
  },
};
```

1-8-2. Provide / Inject

[3] Inject
① 상위 컴포넌트에서 제공한 데이터를 삽입하려면 하위 컴포넌트 setup() 함수 내부에서 inject() 함수를 사용
② 사용 예시

```
import { inject } from 'vue';
export default {
  setup() {
    const message = inject('message');
    const appMessage = inject('appMessage');
    return {
      message,
      appMessage,
    };
  },
};
```

③ 주입된 값이 ref이면 반응성 연결을 유지

[4] Injection 기본값
① inject로 주입된 키가 상위 체인 어디에서든 제공되지 않을경우 런타임 경고가 표시되며, 이 때 두 번째 인자로 기본값(Default Value)을 설정

```
const defaultMessage = inject('defaultMessage', 'default message');
```

② 기본값으로 팩토리함수를 제공가능

```
const defaultMessage = inject('defaultMessage', () => 'default message');
```

[5] Reactivity
① Provide/Inject를 반응성 데이터로 제공할 때 가능한 모든 변경을 Provider 내부에서 하는 것 권장
Provider 내부에 배치되면 향후 유지관리가 용이
② injector 내부 컴포넌트에서 반응성 데이터를 변경해야 하는 경우 데이터 변경을 제공하는 함수를 함께 제공하는 것 권장

1-8-3. Provide / Inject

[6] Reactivity 예시

① 상위 컴포넌트에서 제공한 데이터를 삽입하려면 하위 컴포넌트 setup() 함수 내부에서 inject() 함수를 사용

```
// Provider
const message = ref('Hello World!');
const updateMessage = () => {
  message.value = 'world!';
};
provide('message', { message, updateMessage });
```

② injector 사용 예시

```
// Injector
const { message, updateMessage } = inject('message');
```

③ 주입된 컴포넌트에서 제공된 값을 변경할 수 없도록 하려면 readonly() 함수를 사용 가능

```
import { provide, readonly, ref } from 'vue';

provide('count', readonly(count));
```

1-8-4. Provide / Inject

[6] Symbol 키 사용

① 대규모 애플리케이션에서 다른 개발자와 함께 작업할 때 잠재적 충돌을 피하기 위해 Symbol 주입 키를 사용하는 것 권장

```
// keys.js
export const myInjectionKey = Symbol()
```

②

```
// in provider component
import { provide } from 'vue'
import { myInjectionKey } from './keys.js'

provide(myInjectionKey, {
  /* data to provide */
})
```

③ 주입된 컴포넌트에서 제공된 값을 변경할 수 없도록 하려면 readonly() 함수를 사용 가능

```
// in injector component
import { inject } from 'vue'
import { myInjectionKey } from './keys.js'

const injected = inject(myInjectionKey)
```

1-8-5. Provide / Inject

[6] App-level Provide

① 컴포넌트에서 데이터를 제공하는 것 외에도 App-level에서 제공 가능

```
import { createApp } from 'vue';
import App from './App.vue';
const app = createApp(App);
app.provide('appMessage', 'Hello app message');
app.mount('#app');
```

② // in provider component

```
import { provide } from 'vue'
import { myInjectionKey } from './keys.js'

provide(myInjectionKey, {
  /* data to provide */
})
```

② Provide/Inject 사용 예시

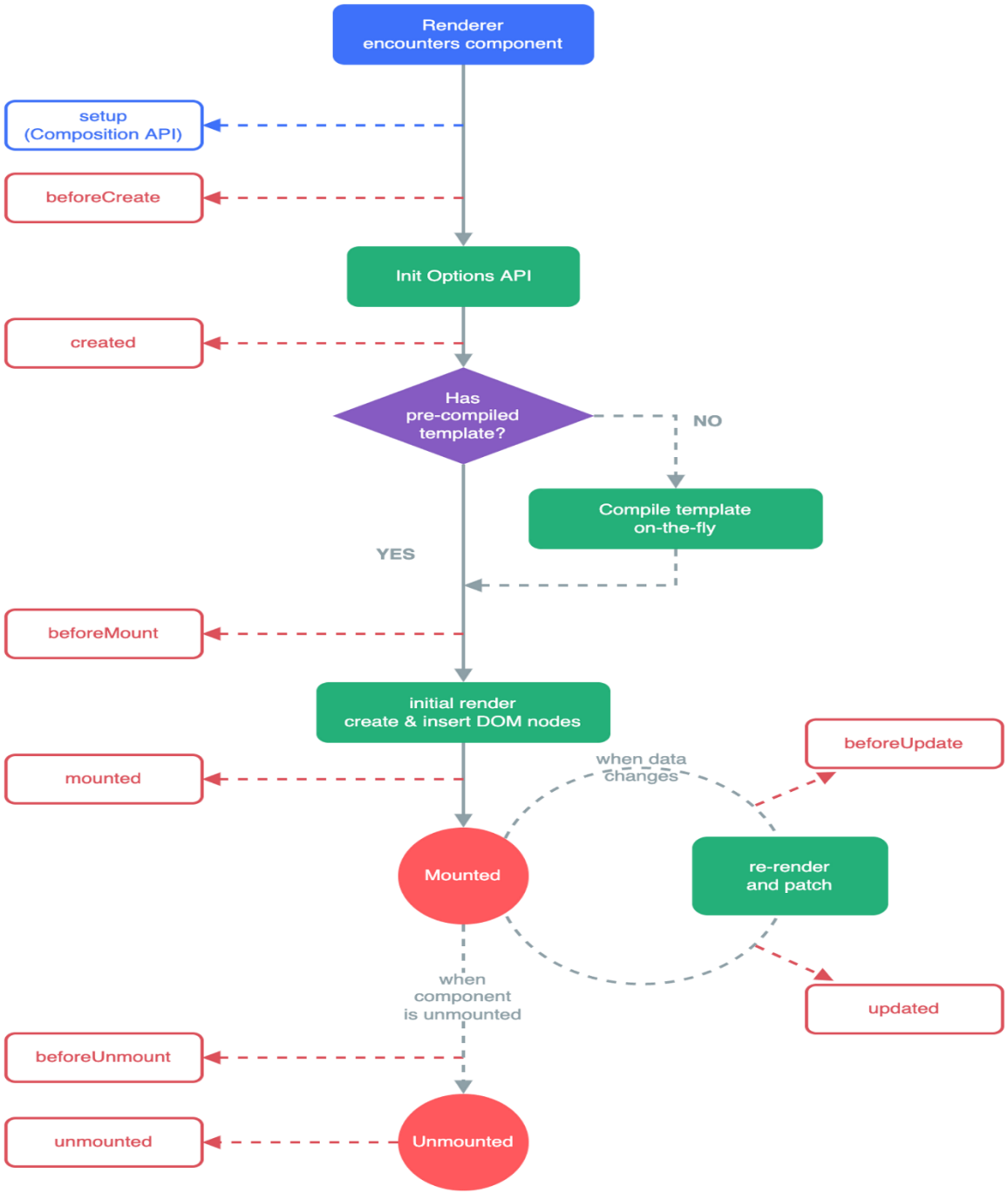
- App-level에서의 Provide는 앱에서 렌더링되는 모든 컴포넌트에서. 이것은 Plugin을 작성할 때 유용
- Vue2에서 컴포넌트 인스턴스 객체를 추가할 때 global property에 추가 했으나, Vue3에서 Composition API Setup 함수에서는 컴포넌트 인스턴스에 접근할 수 없고, 이때 대신 Provide/Inject를 사용가능

1-9-1. Lifecycle hooks

[1] lifecycle / Lifecycle hooks 개념
① Vue 컴포넌트 인스턴스는 생성되고 소멸될 때
사전에 정의된 몇 단계의 과정을 거치게 되는데
이를 라이프사이클(lifecycle)이라 함
② 라이프사이클 훅(Lifecycle hooks)은 라이프사이클
단계에서 사용자가 자신의 코드를 추가할 수 있는
단계별 기능(function)
[2] Lifecycle hooks 등록
컴포넌트가 렌더링을 완료하고 DOM 노드를
만든 후 onMounted hooks를 사용하여 코드를
실행 가능

```
import { onMounted } from 'vue';

export default {
  setup() {
    onMounted(() => {
      console.log('컴포넌트 mounted');
    });
  },
};
```



1-9-2. Lifecycle hooks

[3] Lifecycle Hooks 함수
① Lifecycle Hooks : 컴포넌트 라이프 사이클의 각 단계에서 실행되는 함수들
② Lifecycle Hooks 에 접두사 on을 붙여 컴포넌트의 Lifecycle Hooks 에서 코드를 실행
Lifecycle Hooks 은 setup() 함수 내에서 동기적으로 호출

[4] 여러 라이프사이클 훅 단계와 setup() 함수 내에서 호출하는 방법

Options API	**setup 내부에서 사용**
beforeCreate	필요하지 않음*
created	필요하지 않음*
beforeMount	onBeforeMount
mounted	onMounted
beforeUpdate	onBeforeUpdate
updated	onUpdated
beforeUnmount	onBeforeUnmount

Options API	**setup 내부에서 사용**
unmounted	onUnmounted
errorCaptured	onErrorCaptured
renderTracked	onRenderTracked
renderTriggered	onRenderTriggered
activated	onActivated
deactivated	onDeactivated
serverPrefetch	onServerPrefetch

1-9-3. Lifecycle hooks 생성 ,장착,수정,소멸

[1] 라이프사이클 훅 생성 ,장착,수정,소멸

- Creation(생성) → Mounting(장착) → Updating(수정) → Destruction(소멸)

[2] Creation

- 컴포넌트 초기화 단계이며 Creation Hooks은 라이프사이클 단계에서 가장 먼저 실행
- 아직 컴포넌트가 DOM에 추가되기 전이므로 DOM에 접근할 수 없음
- 서버 렌더링에서 지원되는 단계
- 클라이언트나 서버 렌더 단에서 처리해야 할 일이 있으면 이 단계에서 진행

① beforeCreate

컴포넌트 인스턴스가 초기화 될 때 실행. data() 또는 computed와 같은 다른 옵션을 처리하기 전에 즉시 호출

② created : 컴포넌트 인스턴스가 초기화를 완료한 후 호출되는 훅

③ setup : Composition API의 setup() 훅은 Options API 훅 보다 먼저 호출

Vue3 Composition API를 활용하여 개발을 진행할 때는 setup()함수로 대체 예시

```
export default {
  beforeCreate() {
  },
  created() {
  },
  setup() {
                    // coding...
      }
}
```

1-9-4. Lifecycle hooks 생성 ,장착,수정,소멸

[3] Mounting

- DOM에 Component 를 삽입하는 단계
- 아직 컴포넌트가 DOM에 추가되기 전이므로 DOM에 접근할 수 없음
- 서버렌더링에서 지원되지 않음
- 초기 렌더링 직전에 돔을 변경하고자 한다면 이 단계에서 진행

① onBeforeMount : Component 가 Mount되기 직전에 호출, 권장 않함
② onMounted : Component 가 Mount된 후에 호출되며, DOM에 접근
모든 자식 Component가 Mount 되었음을 의미

[4] Updating

- Component에서 사용되는 반응형 Data가 변경되거나 어떠한 이유로 재 Rendering이 발생될 때 호출
- 디버깅이나 프로파일링 등을 위해 Component 재 렌더링 시점을 알고 싶을 때 사용

① onBeforeMount : 반응형 상태 변경으로 인해 Component가 DOM 트리를 Update하기 직전에 호출
Component 에서 사용되는 반응형 상태 값이 변해서, DOM에도 그 변화를 적용시켜야 할 때

② onUpdated : 반응 상태 변경으로 인해 컴포넌트가 DOM 트리를 업데이트한 후에 호출
이 혹은 다른 상태 변경으로 인해 발생할 수 있는 컴포넌트의 DOM 업데이트 후에 호출

[5] Destruction

- 해체(소멸)단계

① onBeforeUnmount : 컴포넌트가 마운트 해제되기 직전에 호출
② onUnmounted : 컴포넌트가 마운트 해제된 후 호출

1-10-1. Template refs

[1] Template refs 개념
- Vue의 선언적 렌더링 모델은 대부분의 직접적인 DOM의 작업을 대신 수행
- 기본 DOM요소에 직접 접근해야 하는 경우 , ref 특수 속성을 사용해서 쉽게 접근
- ref는 특수 속성입니다. 이 ref 특수 속성을 통해 마운트된 DOM 요소 또는 자식 컴포넌트에 대한 참조 얻음
- <input type="text" ref="input" />

[2] Refs 접근하기
- Composition API로 참조를 얻으려면 동일한 이름의 참조를 선언
- 컴포넌트가 마운트된 후에 접근
- <template> 안에서 input으로 Refs참조에 접근하려는 경우 렌더링되기 전에는 참조가 null일 수 있음
- <template> 안에서 $refs 내장 객체로 Refs 참조에 접근 가능

[3] 사용 예시

```
<template>
  <input ref="input" type="text" />
  <div>{{ input }}</div>
  <div>{{ $refs.input }}</div>
  <div>{{ input === $refs.input }}</div>
</template>
```

```
<script>
    import { onMounted, ref } from 'vue';

    export default {
      components: {},
      setup() {
        const input = ref(null);

        onMounted(() => {
          input.value.value = 'Hello World!';
          input.value.focus();
        });
        return {
          input,
        };
      },
    };
</script>
```

1-10-2. Template refs

[4] v-for 내부 참조
- v3.2.25 이상에서 동작
- v-for내부에서 ref가 사용될 때
  ref는 마운트 후 요소 배열로 채워짐
- **현재 v3.2.31 버전에서 정상작동 확인**

```
<script>
import { ref, onMounted } from 'vue'

export default {
        setup() {
                const list = ref([1, 2, 3])

                const itemRefs = ref([])

                onMounted(() => console.log(itemRefs.value))

                return {
                        list,
                        itemRefs
                }
        }
}
</script>

<template>
  <ul>
    <li v-for="item in list" ref="itemRefs">
        {{ item }}
    </li>
  </ul>
</template>
```

1-10-3. Template refs

[5] 컴포넌트 Refs
- ref를 자식 컴포넌트에도 사용할 수 있음.
- ref로 자식 컴포넌트에 참조값을 얻게 되면 자식 컴포넌트의 모든 속성과 메서드에 대한 전체를 접근
- 부모/자식 컴포넌트간 의존도가 생기기 때문에 이러한 방법은 반드시 필요한 경우에만 사용.
- 일반적으로 ref 보다 **표준 props**를 사용하여 부모/자식간 상호작용을 구현

```
// 자식 컴포넌트를 정의
// Child.vue
<template>
  <div>Child Component</div>
</template>
<script>
import { ref } from 'vue';

export default {
  setup() {
    const message = ref('Hello Child!');
    const sayHello = () => {
      alert(message.value);
    };
    return {
      message,
      sayHello,
    };
  },
};
</script>
```

```
// 부모 컴포넌트에서 자식 컴포넌트의
  상태나 메서드에 접근
<template>
  <button
@click="child.sayHello()">child.sayHello()
</button>
  <Child ref="child"></Child>
</template>

<script>
import { onMounted, ref } from 'vue';
import Child from
'./components/templateRefs/Child.vue';
export default {
  components: {
    Child,
  },
  setup() {
    const child = ref(null);
    onMounted(() => {
      console.log(child.value.message);
    });
    return { child };
  },
};
</script>
```

```
// $parent
// 자식 컴포넌트에서
    상위 컴포넌트
  참조하기 위해서는 $parent
  내장객체
  를 사용

<template>
  <div>Child Component</div>
  <div>{{ $parent.message }}</div>
</template>
```

1-11-1. script setup

[1] script setup

- Single-File Component 내에서 Composition API를 사용하기 위한 syntactic sugar.
- SFC와 Composition API를 사용하는 경우 **<script setup>을 사용하는 것을 권장**
- <script>구문보다 많은 장점을 제공

① 간결한 문법으로 상용구(boilerplate)를 줄일 수 있음

② 타입스크립트를 사용한 **props**와 **emits** 선언 가능 (공식문서)

③ 런타임 성능의 향상(템플릿이 setup 스크립트와 같은 스코프(scope)에 있는 render 함수로 컴파일되므로 프록시가 필요 없음)

④ 더 뛰어난 IDE 타입 추론 성능 (language 서버)가 코드로부터 타입을 추론해내는 데 비용이 절감

[2] 기본 문법

① 내부 코드는 컴포넌트의 setup() 함수 안의 코드로 컴파일

② 컴포넌트를 처음 가져올 때 한 번만 실행되는 일반 <script>와 달리, <script setup>는 컴포넌트의 인스턴스가 생성될 때마다 <script setup>내부 코드가 실행

```
<script setup>
</script>
```

1-11-2. script setup

[3] Top-level에 선언
- <script setup> 내부 최 상위에 선언된 변수, 함수, import 는 <template>에서 직접 사용

```
// 자식 컴포넌트를 정의
<script setup>
const msg = 'Hello!'

function log() {
  console.log(msg)
}
</script>

<template>
  <div @click="log">{{ msg }}</div>
 <HelloComponent></HelloComponent>
</template>
```

```
// import 된 자원(Component, Utils 등)도 동일한 방식으로
  <template>에서      직접 사용


<script setup>
  import HelloComponent from './components/HelloComponent.vue'
</script>


<template>
    <HelloComponent></HelloComponent>
</template>
```

```
// Reactivity APIs(ref, reactive, computed, watch 등) 를 <script setup> 안에서
   생성하면 <template>에서 직접적으로 사용가능
<template>
   <p>{{ message }}</p>
</template>
<script setup>
import { ref } from 'vue';

const message = ref('Hello World!');
</script>
```

1-11-3. script setup

[4] defineProps() & defineEmits()

- **defineProps**()와 **defineEmits**() APIs를 <script setup> 내에 선언하여 props와 emits을 사용

① defineProps와 defineEmits는 <script setup>**내부에서만 사용**할 수 있는 컴파일러 매크로
그렇기 때문에 import할 필요가 없으면 <script setup>이 처리될 때 컴파일

② defineProps는 props옵션과 동일한 값을 허용, defineEmits는 emits옵션과 동일한 값을 허용

③ setup() 영역에 선언된 지역 변수를 참조할 수 없음. 만약 그렇게 하면 컴파일 오류가 발생

④ defineProps와 defineEmits에 전달된 옵션은 setup()에서 모듈 영역(module scope)으로 호이스트

```
// script setup 내에 선언
<script setup>
const props = defineProps({
  foo: String
})

const emit =
   defineEmits(['change', 'delete'])
</script>
```

1-11-4. script setup

[5] **defineExpose**
- <script setup>을 사용하는 컴포넌트는 기본적으로 Template Refs나 $parent와 같이 컴포넌트간 통신이 닫혀 있음
- <script setup>을 사용하는 컴포넌트의 내부 데이터나 메서드를 명시적으로 노출하려면 `defineExpose()`
  컴파일러 매크로를 사용

```
// defineExpose
<script setup>
import { ref } from 'vue'

const a = 1
const b = ref(2)

defineExpose({
  a,
  b
})
</script>
```

```
// expose는 일반 <script>에서도 사용

export default {
  setup(props, context) {
    // Expose public properties (Function)
    console.log(context.expose)
  }
}
```

1-11-5. script setup

[6] useSlots() & useAttrs()
- <slots과 attrs는 <template> 내부에서 $slots와 $attrs로 직접 접근해서 사용
- 만약 **<script setup> 내부**에서 slots과 attrs를 사용하고 싶다면 각각 **useSlots(), useAttrs() helper 메서드**를 사용

```
// useSlots() & useAttrs()

<script setup>
import { useSlots, useAttrs } from 'vue'

const slots = useSlots()

// fallthrough 속성 접근하기
const attrs = useAttrs()

</script>
```

```
// slots과 attrs 는 일반 <script>에서도 사용
export default {
  setup(props, context) {
    // Attributes (Non-reactive object, equivalent to $attrs)
    console.log(context.attrs)

    // Slots (Non-reactive object, equivalent to $slots)
    console.log(context.slots)
  }
}
```

1-11-6. script setup

[7] **<script>와 <script setup>**
- <script setup> 은 normal <script> 와 함께 사용가능. 예를 들면 다음과 같은 경우에 normal <script>가 필요.
- <script setup>에서 표현할 수 없는 inheritAttrs옵션이나 Plugin을 통해 활성화된 Custom 옵션을 사용하고자 할때 normal <script>를 함께 선언

```
<script>
// 일반 스크립트, 모듈 범위에서 한 번만 실행
runSideEffectOnce()

// 옵션 선언
export default {
  inheritAttrs: false,
  customOptions: {}
}
</script>

<script setup>
// 각 인스턴스 생성시 setup() 범위에서 실행
</script>
```

1-11-7. script setup

[8] Top-level await
- <script setup> 내의 Top-level에서 await을 사용.
- 코드는 async setup() 이렇게 컴파일

```
<script setup>
    const post = await fetch(`/api/post/1`).then((r) => r.json())
</script>
```

[9] vue/setup-compiler-macros
- defineProps 및 defineEmits와 같은 컴파일러 매크로는 un-undef 경고를 생성
- ESLint config 파일에서 컴파일러 매크로 환경을 활성화.
  이러한 변수를 전역적으로 노출하지 않으려면 /* 전역 defineProps, defineEmits */를 대신 사용