Function Declaration vs Expression in React

Intro

- 리액트 공식 document 에서는 함수 선언식을 이용하여 예제들을 작성
  (https://react.dev/reference/react/useState)

```
import { useState } from 'react';

function MyComponent() {
  const [age, setAge] = useState(28);
  const [name, setName] = useState('Taylor');
  const [todos, setTodos] = useState(() => createTodos());
  // ...
```

- 리액트를 개발할 때 자주 사용하는 코드는, 주로 arrow function을 이용한 함수 표현식

```
// ex) HyperClova Studio FE
const Token: FC<TokenProps> = ({
  showProbabilities,
  token,
  children,
}: TokenProps) => {
  if (showProbabilities) {
    return <HighlightedToken token={token}>{children}</HighlightedToken>;
  }
  return <span style={{ fontWeight: 700 }}>{children}</span>;
};

export default Token;
```

Function Declaration

- 함수 선언문(Function declaration) 방식으로 정의한 함수는 `function` 키워드와 이하의 내용으로 구성

```
// Function Declaration
function square(number) {
  return number * number;
}
```

함수명

- 함수 선언문의 경우, 함수명은 생략할 수 없음
- 함수명은 함수 몸체에서 자신을 재귀적(recursive) 호출하거나 자바스크립트 디버거가 해당 함수를 구분할 수 있는 식별자

매개변수 목록

- 0개 이상의 목록으로 괄호로 감싸고 콤마로 분리

함수 몸체

- 함수가 호출되었을 때 실행되는 문들의 집합
- 중괄호({ })로 문들을 감싸고 return 문으로 결과값을 반환
- 결과값을 반환값(return value)이라 칭함

Function Expression

- 함수의 **일급객체 특성을 이용**하여 함수 리터럴 방식으로 함수를 정의하고 변수에 할당

```
// 함수 표현식
var square = function(number) {
  return number * number;
};
```

일급객체

1. 무명의 리터럴로 표현이 가능
2. 변수나 자료 구조(객체, 배열…)에 저장 가능
3. 함수의 파라미터로 전달 가능
4. 반환값(return value)으로 사용 가능

익명 함수 (anonymous function)

- 함수 표현식 방식으로 정의한 함수는 함수명을 생략 가능
- 함수 표현식에서는 함수명을 생략하는 것이 일반적

```
// 기명 함수 표현식(named function expression)
var foo = function multiply(a, b) {
  return a * b;
};

// 익명 함수 표현식(anonymous function expression)
var bar = function(a, b) {
  return a * b;
};
```

arrow function

- 함수 표현식을 보다 단순하고 간결한 문법으로 함수를 만들 수 있는 방법

```
var sum = (a, b) => a + b;
```

함수호출

- 함수는 일급객체이기 때문에 변수에 할당할 수 있는데 이 변수는 함수명이 아니라 할당된 함수를 가리키는 참조값을 저장
- 함수 호출시 함수명이 아니라 함수를 가리키는 변수명을 사용해야 함
- **함수가 할당된 변수를 사용해 함수를 호출하지 않고 기명 함수의 함수명을 사용해 호출하게 되면 에러가 발생한다. 이는 함수 표현식에서 사용한 함수명은 외부 코드에서 접근 불가능하기 때문**

```
// 기명 함수 표현식(named function expression)
var foo = function multiply(a, b) {
  return a * b;
};

console.log(foo(10, 5)); // 50
console.log(multiply(10, 5)); // Uncaught ReferenceError: multiply is not defined
```

- 함수 표현식과 함수 선언문에서 사용한 함수명은 함수 몸체에서 자신을 재귀적 호출(Recursive function call)하거나 자바스크립트 디버거가 해당 함수를 구분할 수 있는 식별자의 역할을 함
- 함수 선언문으로 정의한 함수의 경우, 함수명으로 호출할 수 있었는데 이는 자바스크립트 엔진에 의해 아래와 같은 함수 표현식으로 형태가 변경되었기 때문

```
var sum = function sum(number) {
  return number + number;
};
```

- 함수명과 함수 참조값을 가진 변수명이 일치하므로 함수명으로 호출되는 듯 보이지만 사실은 변수명으로 호출
- 결국 함수 선언문도 함수 표현식과 동일하게 함수 리터럴 방식으로 정의

Hoisting

Function Declaration

- 함수 선언문으로 정의된 함수는 자바스크립트 엔진이 스크립트가 로딩되는 시점에 바로 초기화하고 이를 VO(variable object)에 저장
- 즉, 함수 선언, 초기화, 할당이 한번에 이루어 짐
- 함수 선언의 위치와는 상관없이 소스 내 어느 곳에서든지 호출이 가능

```
var res = square(5);

function square(number) {
  return number * number;
}
```

Function Expression

- 함수 표현식의 경우 함수 호이스팅이 아니라 변수 호이스팅이 발생
- 변수 호이스팅은 변수 생성 및 초기화와 할당이 분리되어 진행
- 호이스팅된 변수는 undefined로 초기화 되고 실제값의 할당은 할당문에서 이루어 짐

```
var res = square(5); // TypeError: square is not a function

var square = function(number) {
  return number * number;
}
```

Export Default

- 해당 모듈엔 개체가 하나만 있다는 사실을 명확히 제시

```
export default function Test(){}
```

- named export와 default export를 같은 모듈에서 동시에 사용해도 문제없지만 실무에서는 잘 사용하지 않음

```
export default function Test(){};
export const person = {name: 'lee', age: '24'};
```

- 파일당 최대 하나의 default export가 있을 수 있으므로 내보낼 개체엔 이름이 없어도 무관

```
export default class { // 클래스 이름이 없음
  constructor() { ... }
}
export default function(user) { // 함수 이름이 없음
  alert(`Hello, ${user}!`);
}
// 이름 없이 배열 형태의 값을 내보냄
export default ['Jan', 'Feb', 'Mar','Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];
```

Function Declaration

- export default 가능

```
export default function Fn(){}
```

Function Expression

- export default 불가능
- 함수 표현식 하단에서 별도로 export

```
// incorrect
export default const Fn = () => {}

// correct
const Fn = () => {}

export default Fn;
```

Function Expression for Closure

- https://joshua1988.github.io/web-development/javascript/function-expressions-vs-declarations/
- https://string.tistory.com/117

```
function tabsHandler(index) {
    return function tabClickEvent(event) {
        console.log(index);
    };
}

var tabs = document.querySelectorAll('.tab');
var i;

for (i = 0; i < tabs.length; i += 1) {
    tabs[i].onclick = tabsHandler(i);
}
```

▼ 함수 표현식의 장점 중 closure를 포함하며 위와 같은 코드를 제시하는 예시가 많음

- 일급함수 객체의 특성을 이용하여 return으로 tabClickEvent를 반환
- 반환한 함수를 onClick에 할당하므로 함수 표현식의 예제라 볼 수 있음

Difference in React with Typescript

- React with TypeScript: Components as Function Declarations vs. Function Expressions
- https://blog.variant.no/a-better-way-to-type-react-components-9a6460a1d4b7
  https://velog.io/@jjunyjjuny/번역-The-Better-Way-to-Type-React-Components

Function Declaration

- 함수의 return에 대한 타입을 지정

```
function MyComponent(): React.ReactNode {
  return <h1>Hello, world</h1>
}

// @types/react
type ReactNode = ReactChild | ReactFragment | ReactPortal | boolean | null | undefined;
```

Function Expression

- 변수에 할당되는 함수 자체에 대한 타입을 지정

```
const MyComponent: React.FC<Props> = () => <h1>Hello, world!</h1>
```

Typing Props With Function Declaration

```
type Props = {
  name: string;
}
function MyComponent({ name }: Props): React.ReactNode {
  return <h1>Hello, {name}!</h1>
}
```

Typing Props With Function Expression

```
type Props = {
  name: string;
}
const MyComponent: React.FC<Props> = ({ name}) => <h1>Hello, {name}!</h1>
```

Children With Function Declaration

- 직접 지정

```
type Props = {
  name: string;
  children: React.ReactNode;
}
function MyComponent({ name, children }: Props): React.ReactNode {
  return <h1>{children}, {name}!</h1>
}
```

- PropsWithChildren (children을 props에 포함)

```
type Props = {
  name: string;
}
function MyComponent({ name }: PropsWithChildren<Props>): React.ReactNode {
  return <h1>{children}, {name}!</h1>
}
```

Children With Function Expression

- 직접 지정

```
type Props = {
  name: string;
  children: React.ReactNode;
}
const MyComponent: React.FC<Props> = ({ name, children }) => <h1>{children}, {name}!</h1>
```

- PropsWithChildren (children을 props에 포함)

```
type Props = {
  name: string;
}
const MyComponent: React.FC<PropsWithChildren<Props>> = ({ name, children }) => <h1>{children}, {name}!</h1>
```

- PropsWithChildren 을 명시적으로 사용하지 않아도, Recat.FC는 Children을 포함
  이것은 편리해 보이지만, **실제로는 나쁜 것**

```
type FC<P = {}> = FunctionComponent<P>;

interface FunctionComponent<P = {}> {
    (props: PropsWithChildren<P>, context?: any): ReactElement<any, any> | null;
    propTypes?: WeakValidationMap<P>;
    contextTypes?: ValidationMap<any>;
    defaultProps?: Partial<P>;
    displayName?: string;
}
type PropsWithChildren<P> = P & { children?: ReactNode };
```

No Generics Issue

- React.FC를 사용할 때 TypeScript는 Generics를 허용하지 않음

```
type MyDropDownProps<T> = {
 items: T[];
 itemToString(item: T): string;
 onSelected(item: T): void;
};

// Neither of these examples are valid
const MyDropDown: React.FC<MyDropDownProps> = (props) => {};
const MyDropDown: React.FC<MyDropDownProps<T>> = <T>(props) => {};
const MyDropDown: React.FC<MyDropDownProps<T>> = (props) => {};
const MyDropDown<T>: React.FC<MyDropDownProps<T>> = (props) => {};
```

- Function Declaration 을 통해 Generic 사용

```
type MyDropDownProps<T> = {
  items: T[];
  itemToString(item: T): string;
  onSelected(item: T): void;
};
// Valid code
function MyDropDown<T>(props: MyDropDownProps<T>) {}
```

Common Recommend Style

Function Expression

- JavaScript: The Good Parts의 저자이며 자바스크립트의 권위자인 더글러스 크락포드(Douglas Crockford)는 위와 같은 문제 때문에 함수 표현식만을 사용할 것을 권고
- 함수 호이스팅이 함수 호출 전 반드시 함수를 선언하여야 한다는 규칙을 무시하므로 코드의 구조를 엉성하게 만들 수 있다고 지적
- 또한 함수 선언문으로 함수를 정의하면 사용하기에 쉽지만 대규모 애플리케이션을 개발하는 경우 인터프리터가 너무 많은 코드를 변수 객체(VO)에 저장하므로 애플리케이션의 응답속도는 현저히 떨어질 수 있으므로 주의해야 함

React with Typescript Recommend Style

Function Declaration

- 호이스팅 허용 (컨텐츠 우선 순위 관점)
  메인 컨텐츠를 파일의 가장 윗부분에 서술 가능

```
// 1. Setup variables. Works as expected
const BoundFooItem = partial(FooItem, { title: "Foo" });
// 2. Main export
```

```
export default function Foo() {}

// 3. Additional exports. Often children types that can be semantically grouped with main export.
// (Many prefer not to mix default with named exports, but a discussion for another time).
export function FooItem() {}
// 4. Helper components
function MyInternalComponent() {}
// 5. Utils
function add() {}
function partial() {}
function identity() {}
```

- export default 지원
- children에 대한 오탐 방지 (avoiding false positives in React.FC)
- generics을 지원
- React 컴포넌트의 props 타입을 항상 함수의 매개변수로 직접 지정 권장

react/function-component-definition (Eslint)

- https://github.com/jsx-eslint/eslint-plugin-react/blob/master/docs/rules/function-component-definition.md
- 함수 정의 유형을 결정하는 eslint rule

```
"react/function-component-definition":[
      <enabled>,
      {
        // 기명 함수 옵션
      "namedComponents": "function-declaration" | "function-expression" | "arrow-function" | Array<"function-declaration" | "function-expre
        // 익명 함수 옵션
      "unnamedComponents": "function-expression" | "arrow-function" | Array<"function-expression" | "arrow-function">
      }
    ],

// Recommend
"rules": {
  "react/function-component-definition":[
      2,
      { "namedComponents": [
        "arrow-function",
        "function-declaration"
      ] }
    ],
}
```

```
// only function declarations for named components
// [2, { "namedComponents": "function-declaration" }]
function Component (props) {
  return <div />;
}

// only function expressions for named components
// [2, { "namedComponents": "function-expression" }]
const Component = function (props) {
  return <div />;
};

// only arrow functions for named components
// [2, { "namedComponents": "arrow-function" }]
const Component = (props) => {
  return <div />;
};

// only function expressions for unnamed components
// [2, { "unnamedComponents": "function-expression" }]
function getComponent () {
  return function (props) {
    return <div />;
  };
}
```

```
// only arrow functions for unnamed components
// [2, { "unnamedComponents": "arrow-function" }]
function getComponent () {
  return (props) => {
    return <div />;
  };
}
```

Unfixable patterns

- 자바스크립트에는 수정할 수 없는 패턴이 하나 있음
- 아래는 유효한 구문

```
export default function getComponent () {
  return <div />;
}
```

- 아래는 유효하지 않은 구문

```
export default var getComponent = () => {
  return <div />;
}

export default var getComponent = function () {
  return <div />;
}
```

- 이러한 패턴은 수동으로 수정해야 함

Discussion

- 모듈 시스템에서 권장하는 방식 혹은 차이
- 번들된 리액트에서는 선언식 혹은 표현식의 유의미한 차이 존재 유무