Function Expression

JavaScript는 function를 특별한 종류의 value(값)으로 취급한다. 다른 언어에서처럼 "특별한 동작을 하는 구조"로 취급되지 않는다.

이전에 function 선언(Function Declaration), function Declaration 방식으로 function를 만들었다.

```
function sayHi() {
  console.log( "Hello" );
}
```

function 선언 방식 외에 Function Expression(함수 표현식)을 사용해서 function를 만들 수 있다.

function expression으로 function를 생성해보자.

```
let sayHi = function() {
  console.log( "Hello" );
};
```

function를 생성하고 변수에 값을 할당하는 것처럼 function이 변수(sayHi)에 할당되었다. function이 어떤 방식으로 만들어졌는지에 관계없이 function는 value(값)이고, 따라서 변수에 할당할 수 있다. 위 예시에선 function가 변수 sayHi에 저장된 값이 되었다.

☑ Function은 value(값)이다.

위 예시를 간단한 말로 풀면 다음과 같다: "function를 만들고 그 function를 변수 sayHi에 할당하기"

function는 값이기 때문에 console.log를 이용하여 function 코드를 출력할 수도 있다.

```
function sayHi() {
  console.log( "Hello" );
}

console.log( sayHi );     // function 코드가 보임
```

마지막 줄에서 sayHi 옆에 괄호가 없기 때문에 function는 실행되지 않는다. 어떤 언어에선 괄호 없이 function 이름만 언급해도 function가 실행된다. 하지만 JavaScript는 괄호가 있어야만 function가 호출된다.
JavaScript에서 function는 값이다. 따라서 function를 값처럼 취급할 수 있다. 위 코드에선 function 소스 코드가 문자형으로 바뀌어 출력되었다.

function는 sayHi()처럼 호출할 수 있다는 점 때문에 일반적인 값과는 조금 다르다. 특별한 종류의 값이다..
그러나 그 본질은 값이기 때문에 값에 할 수 있는 일을 function에도 할 수 있다.
변수를 복사해 다른 변수에 할당하는 것처럼 function를 복사해 다른 변수에 할당할 수도 있다.

```
function sayHi() {               // (1) function 생성
  console.log( "Hello" );
}

let func = sayHi;                // (2) function 복사

func();           // Hello       // (3) 복사한 function를 실행(정상적으로 실행된다)
sayHi();          // Hello       // 본래 function도 정상적으로 실행된다.
```

위 예시에서 어떤 일이 일어났는지 자세히 알아보자.
(1)에서 function 선언 방식을 이용해 function를 생성한다. 생성한 function는 sayHi라는 변수에 저장된다. (2) 에선 sayHi를 새로운 변수 func에 복사한다. 이때 sayHi 다음에 괄호가 없다는 점에 유의한다.
괄호가 있었다면 func = sayHi(); 가 되어 sayHi function 그 자체가 아니라, function 호출 결과(function의 반환 값) 이 func에 저장되었을 것이다. 이젠 sayHi() 와 func()로 function를 호출할 수 있게 되었다.

function sayHi는 아래와 같이 Function Expression을 사용해 정의할 수 있다.

```
let sayHi = function() {
  console.log( "Hello" );
};

let func = sayHi;
// ...
```

동작 결과는 동일하다.

function expression의 끝에 왜 세미 콜론 ;이 붙는지 의문이 들 수 있다. function Declaration에는 세미 콜론이 없다.

```
function sayHi() {
  // ...
}

let sayHi = function() {
  // ...
};
```

이유는 간단하다. if { ... }, for { }, function f { } 같이 중괄호로 만든 코드 블록 끝엔 ;이 없어도 된다.

function expression은 let sayHi = ...;과 같은 구문 안에서 값의 역할을 한다. 코드 블록이 아니고 값처럼 취급되어 변수에 할당된다. 모든 구문의 끝엔 세미 콜론 ;을 붙이는 게 좋다. function expression에 쓰인 세미 콜론은 function expression 때문에 붙여진 게 아니라, 구문의 끝이기 때문에 붙여졌다.

☑ Callback function

function를 값처럼 전달하는 예시, function expression에 관한 예시를 좀 더 살펴보자.
parameter가 3개 있는 function, ask(question, yes, no)를 작성해보자. 각 parameter에 대한 설명은 아래와 같다.

☞ question
질문의 텍스트

☞ yes
"Yes"라고 답한 경우 실행되는 function

☞ no
"No"라고 답한 경우 실행되는 function

function는 반드시 question(질문)을 해야 하고, 사용자의 답변에 따라 yes() 나 no()를 호출한다.

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

function showOk() {
  alert( "동의하셨습니다." );
}

function showCancel() {
  alert( "취소 버튼을 누르셨습니다." );
}

// 사용법: function showOk와 showCancel가 ask function의 인수로 전달됨
ask("동의하십니까?", showOk, showCancel);
```

이렇게 function를 작성하는 방법은 실무에서 아주 유용하게 쓰인다. 면대면으로 질문하는 것보다 위처럼 confirm 창을 띄워 질문을 던지고, 답변을 받으면 간단하게 설문조사를 진행할 수 있다. 실제 상용 서비스에선 confirm 창을 좀 더 멋지게 꾸미는 등의 작업이 동반되긴 하지만, 일단 여기선 그게 중요한 포인트는 아니다.

function ask의 argument, showOk와 showCancel은 callback function 또는 callback 이라고 불린다.

function을 function의 argument로 전달하고, 필요하다면 인수로 전달한 그 function를 "나중에 호출(called back)" 하는 것이 callback function의 개념이다. 위 예시에선 사용자가 "yes"라고 대답한 경우 showOk가 callback이 되고, "no"라고 대답한 경우 showCancel가 callback이 된다.

아래와 같이 function expression을 사용하면 코드 길이가 짧아진다.

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

ask(
  "동의하십니까?",
  function() { alert("동의하셨습니다."); },
  function() { alert("취소 버튼을 누르셨습니다."); }
);
```

ask(...) 안에 function가 선언된 게 보인다. 이렇게 이름 없이 선언한 function는 anonymous function(익명 함수) 라고 부른다. anonymous function는 (변수에 할당된 게 아니기 때문에) ask 바깥에선 접근할 수 없다. 위 예시는 의도를 가지고 이렇게 구현하였기 때문에 바깥에서 접근할 수 없어도 문제가 없다.

JavaScript를 사용하다 보면 callback을 활용한 코드를 아주 자연스레 만나게 된다. 이런 코드는 JavaScript의 정신을 대변한다.

☞ **function는 "action"을 나타내는 값이다.**

string나 number 등의 일반적인 값들은 데이터를 나타낸다.
function는 하나의 action을 나타낸다.
action을 대변하는 값인 function를 변수 간 전달하고, action이 필요할 때 이 값을 실행할 수 있다.

☑ Function Expression vs Function Declaration

Function Expression과 Function Declaration의 차이에 대해 알아본다.
첫 번째는 문법이다. 코드를 통해 어떤 차이가 있는지 살펴본다.

❖ **Function Declaration: function은 주요 코드 흐름 중간에 독자적인 구문 형태로 존재한다.**

```
// Function Declaration
function sum(a, b) {
  return a + b;
}
```

❖ **Function Expression: function은 expression이나 syntax construct(구문 구성) 내부에 생성된다.** 아래 예시에선 function이 할당 연산자 = 를 이용해 만든 "assignment expression" 우측에 생성되었다.

```
// Function Expression
let sum = function(a, b) {
  return a + b;
};
```

두 번째 차이는 JavaScript Engine이 언제 function를 생성하는지에 있다.

☞ **Function Expression은 실제 실행 흐름이 해당 function에 도달했을 때 function를 생성한다.**
따라서 실행 흐름이 function에 도달했을 때부터 해당 function를 사용할 수 있다.

위 예시를 이용해 설명해 보도록 한다. script가 실행되고, 실행 흐름이 let sum = function… 의 우측(function expression)에 도달 했을때 function가 생성된다. 이때 이후부터 해당 function를 사용(할당, 호출 등)할 수 있다.

그러나 Function Declaration은 조금 다르다.

☞ **Function Declaration은 function Declaration이 정의되기 전에도 호출할 수 있다.**
따라서 global Function Declaration은 script 어디에 있느냐에 상관없이 어디에서든 사용할 수 있다.
이 것이 가능한 이유는 JavaScript의 내부 알고리즘 때문이다. JavaScript는 script를 실행하기 전, 준비 단계에서 global에 선언된 Function Declaration을 찾고, 해당 function를 생성한다. script가 진짜 실행되기 전 "initialization stage(초기화 단계)"에서 function 선언 방식으로 정의한 function가 생성되는 것이다.

script는 Function Declaration이 모두 처리된 이후에서야 실행된다. 따라서 script 어디서든 Function Declaration으로 선언한 function에 접근할 수 있는 것이다.

예시를 살펴 보자.

```
sayHi("Smart");  // Hello, Smart

function sayHi(name) {
  console.log( `Hello, ${name}` );
}
```

Function Declaration, sayHi는 script 실행 준비 단계에서 생성되기 때문에, script 내 어디에서든 접근할 수 있다.
그러나 Function Expression으로 정의한 function는 function가 선언되기 전에 접근하는 게 불가능하다.

```
sayHi("John");   // error!

let sayHi = function(name) {  // (1) 마술은 일어나지 않는다.
  console.log( `Hello, ${name}` );
};
```

Function Expression은 실행 흐름이 expression에 다다랐을 때 만들어진다. 위 예시에선 (1)로 표시한 줄에 실행 흐름이 도달했을 때 function가 만들어진다.

세 번째 차이점은 Block Scope 이다.

☞ strict mode(엄격 모드)에서 Function Declaration이 code block 내에 위치하면, 해당 function는 block 내 어디서든 접근할 수 있다. 하지만 block 밖에서는 function에 접근하지 못한다.

예시를 들어 설명해 보자.
runtime에 그 값을 알 수 있는 변수 age가 있고, 이 변수의 값에 따라 function welcome()을 다르게 정의해야 하는 상황이다. 그리고 function welcome()은 나중에 사용해야 하는 상황이라고 가정해 보자.
Function Declaration을 사용하면 의도한 대로 코드가 동작하지 않는다.

```
let age = prompt("나이를 알려주세요.", 18);

// 조건에 따라 function를 선언함
if (age < 18) {
  function welcome() {
    console.log("안녕!");
```

```
  }

} else {
  function welcome() {
    console.log("안녕하세요!");
  }
}

// function를 나중에 호출한다.
welcome(); // Error: welcome is not defined
```

Function Declaration은 function가 선언된 code block 안에서만 유효하기 때문에 이런 error가 발생한다.

또 다른 예시를 살펴보자.

```
let age = 16;    // 16을 저장했다 가정한다.

if (age < 18) {
  welcome();               // \   (실행)
                           //  |
  function welcome() {     //  |
    console.log("안녕!");  //  |  Function Declaration은 function이 선언된 블록 내
  }                        //  |  어디에서든 유효하다
                           //  |
  welcome();               // /   (실행)

} else {
  function welcome() {
    console.log("안녕하세요!");
  }
}

// 여기는 중괄호 밖이기 때문에
// 중괄호 안에서 선언한 Function Declaration은 호출할 수 없다.

welcome(); // Error: welcome is not defined
```

그럼 if 문 밖에서 welcome function를 호출할 방법은 없는 걸까?

Function expression을 사용하면 가능하다. if 문 밖에 선언한 변수 welcome에 Function Expression으로 만든 function를 할당하면 된다.

이제 코드가 의도한 대로 동작한다.

```
let age = prompt("나이를 알려주세요.", 18);

let welcome;

if (age < 18) {
  welcome = function() {
    console.log("안녕!");
  };

} else {
  welcome = function() {
    console.log("안녕하세요!");
  };

}

welcome();        // 제대로 동작한다.
```

물음표 연산자 ?를 사용하면 위 코드를 좀 더 단순화할 수 있습니다.

```
let age = prompt("나이를 알려주세요.", 18);

let welcome = (age < 18) ?
  function() { console.log("안녕!"); } :
  function() { console.log("안녕하세요!"); };

welcome(); // 제대로 동작한다.
```

☞ Function Declaration과 Function Expression 중 무엇을 선택해야 하나?

Function Declaration을 이용해 function를 선언하는 걸 먼저 고려하는 게 좋다. Function Declaration으로 function를 정의하면, function가 선언되기 전에 호출할 수 있어서 코드 구성을 좀 더 자유롭게 할 수 있다.
Function Declaration을 사용하면 가독성도 좋아진다. 코드에서 let f = function(…) {…} 보다 function f(…) {…} 을 찾는 게 더 쉽다. Function Declaration 방식이 더 "눈길을 사로잡는다".

그러나 어떤 이유로 Function Declaration 방식이 적합하지 않거나, (위 예제와 같이) 조건에 따라 function를 선언해야 한다면 Function Expression을 사용해야 한다.

☞ 요약하면 function는 value(값) 이다. 따라서 function도 값처럼 할당, 복사, 선언할 수 있다.
☞ "Function Declaration" 방식으로 function를 생성하면, function가 독립된 구문 형태로 존재하게 된다.
☞ "Function Expression" 방식으로 function를 생성하면, function가 expression의 일부로 존재하게

된다.

- ☞ Function Declaration은 code block이 실행되기도 전에 처리된다. 따라서 block 내 어디서든 활용 가능하다.
- ☞ Function Expression은 실행 흐름이 expression에 다다랐을 때 만들어진다.

function를 선언해야 한다면 function가 선언되기 이전에도 function를 활용할 수 있기 때문에, Function Declaration 방식을 따르는 게 좋다. function 선언 방식은 코드를 유연하게 구성할 수 있도록 해주고, 가독성도 좋다.

Function Expression은 Function Declaration을 사용하는게 부적절할 때에 사용하는 것이 좋다.