4장 함수

💡 **자바스크립트에서 함수란?**

일급(first-class) 객체!
객체를 다루듯이 함수를 변수에 할당하거나, 함수를 다른 함수로 전달하거나, 함수에 기록된 프로퍼티를 읽는 등의 작업을 할 수 있다.

함수의 타입

보통 매개변수의 타입은 명시적으로 정의한다. 왜냐하면 타입스크립트는 함수 본문에서 사용된 타입들을 추론하지만 특별한 상황을 제외하면 **매개변수 타입을 추론**하지 않기 때문이다.
하지만 **반환 타입은 자동으로 추론**한다. 타입스크립트가 해주는 일을 개발자가 하지 않도록 주의하자

다섯 가지 함수 선언 방법

```
// 1. 이름을 붙인 함수
function greet(name: string){
  return 'hello ' + name;
}

// 2. 함수 표현식
let greet2 = function (name: string) {
  return 'hello ' + name;
}

// 3. 화살표 함수 표현식
let greet3 = (name: string) {
  return 'hello ' + name;
}

// 4. 단축형 화살표 함수 표현식
let greet4 = (name: string) => 'hello ' + name;

// 5. 함수 생성자
let greet5 = new Function('name', 'return "hello " + name')
```

💡 **함수 생성자는 벌떼에 쫓기는 상황이 아니라면 사용하지 맙시다!**

타입스크립트를 사용해도 매개변수 타입 및 반환 타입을 지정하지 않았으므로 이 과정에서 이상한 문제가 발생하더라도 타입스크립트가 도와줄 수 없기 때문이다.

```
  let sum = new Function('a', 'b', 'return a + b');
  console.log(typeof sum); // Function
  console.log(sum(1, 1, 1, 1, 1)); // 2
  console.log(sum('1', 'a')); // 1a
```

선택적 매개변수와 기본 매개변수

```
function sum(first: number, second = 3, more?: number) {
  if (more) return first + second + more;
```

```
  return first + second;
}
console.log(sum(1, 2));
```

기본 매개변수: first
선택적 매개변수: more
매개변수에 기본 값 설정: second

나머지 매개변수

```
function sum(...numbers: number[]) {
  return numbers.reduce((total, n) => total + n, 0);
}
console.log(sum(1, 2, 3, 4));
```

💡 **함수 인자를 arguments 객체를 통해 받지 맙시다!**

arguments는 일종의 배열이지만 순수한 배열은 아니므로 진짜 배열로 변환해야 한다. 하지만 진짜 배열로 변환 시 타입을 any로 추론하고 실제로 매개변수를 받지 않는 함수를 선언하면 타입스크립트 입장에선 인수를 받을 수 없다며 TypeError를 발생시킨다.

this의 타입 지정

자바스크립트에서 메서드를 호출할 때 this는 점 왼쪽의 값을 갖는다는 것이 일반적인 원칙이지만 호출이 일어나기 전 a를 다시 할당하면 결과가 달라진다.

```
let x = {
  a() {
    return this;
  },
};

x.a(); // 객체 x를 this로 받음

let a = x.a;
a(); // this는 정의되지 않음
```

하지만 타입스크립트에서 기대하는 this 타입을 함수의 첫 번째 매개변수로 선언한다면 함수에 등장하는 모든 this가 의도한 this가 됨을 보장해준다.

```
function fancyDate(this: Date) {
  return `${this.getDate()} / ${this.getMonth()} / ${this.getFullYear()}`;
}
```

호출 시그니처 || 타입 시그니처

함수 전체 타입을 표현하는 방법이며 타입 수준의 코드, 즉 값이 아닌 타입 정보만 포함한다.

```
function sum(a: number, b: number): number {
  return a + b;
}
```

```
//호출 시그니처 || 타입 시그니처
(a: number, b: number) => number;

// 독립 호출 시그니처
type Sum = (a: number, b: number) => number;
```

문맥적 타입화

위의 sum함수를 호출할 때 아래처럼 타입 없이 선언할 수 있다.

```
sum(1, 2);
```

변수에 함수를 할당할 때 매개변수에 타입을 지정해주지 않았음에도 불구하고 타입스크립트에서 타입을 추론해주는 것이다.

다형성

하나의 객체가 여러 타입을 가질 수 있는 것을 의미한다.

제네릭

지금은 타입이 무엇인지 알 수 없고 호출 시에 타입을 지정해주는 것을 의미한다.
꺾쇠괄호(<> 로 제네릭 타입 매개변수임을 선언할 수 있다.

```
type Filter = {
  <T>(array: T[], f: (item: T) => boolean): T[];
  // T는 플레이스 홀더 타입, 제네릭 타입 매개변수라고 부른다.
};

let filter: Filter = (
  array,
  f // ...
) => filter([1, 2, 3], (_) => _ > 2); // T는 number
filter(['a', 'b'], (_) => _ !== 'b'); // T는 string
```

제네릭 타입은 '제네릭을 사용할 때' (함수라면 함수를 호출할 때) 타입이 한정되며, 제네릭의 위치에 따라 함수를 호출하는 방법 또한 달라진다.

한정된 다형성

```
type TreeNode = {
  value: string;
};
type LeafNode = TreeNode & {
  isLeaf: true;
};
type InnerNode = TreeNode & {
  children: [TreeNode] | [TreeNode, TreeNode];
};

function mapNode<T extends TreeNode>(node: T, f: (value: string) => string): T {
  return {
    ...node,
```

```
    value: f(node.value),
  };
}
```

위 코드에서 extends TreeNode를 생략하고 T타입이라고만 사용을 하면 T타입에 대한 상한 경계가 없으므로 `node.value` 를 읽는 행위가 안전하지 않기 때문에 에러가 발생한다.
이렇게 `T extends TreeNode` 라고 표현함으로써 매핑한 이후에도 입력 노드가 **특정 타입**이라는 정보를 보존할 수 있다.

한정된 다형성으로 인수의 개수 정의하기

```
function call<T extends unknown[], R>(f: (...args: T) => R, ...args: T): R {
  return f(...args);
}

let a = call(fill, 10, 'a');
let b = call(fill, 10); //에러: 3개의 인수가 필요하지만 2개가 전달됨
```

제네릭 타입 기본값

```
type MyEvent<T = HTMLElement> = {
  target: T;
  type: string;
};
```

타입 주도 개발

타입 시그니처를 먼저 채워두고 값을 나중에 넣는 개발 방식