Functions

Call Signatures

```
/*
call signatures
함수 위에 마우스를 올렸을 때 뜨는 설명문 같은 것
(함수를 어떻게 호출해야하는지 알려줌)
*/

//일반 함수 형식
function add (a : number, b : number) {
   return a + b
}


//화살표 함수 형식
const add = (a:number , b:number) => a + b


//custom call signature 선언 방식
type Add = (a: number, b: number) => number;    //Add 타입 지정

//add 함수 자체에 타입 설정하였기 때문에
//인자 및 리턴을 별도로 지정하지 않아도 된다.
const add:Add = (a, b) => a + b
const add:Add = (a, b) => {a + b}   //이렇게하면 리턴을 하지 않는다
```

polymorphism(다형성)

```
//polymorphism
//ts에게 타입을 유추하도록 알려줌

//concrete type : 전부터 봐왔던 타입 ex))numbers, boolean
//generic : 타입의 placeholder, concrete 타입 대체 가능
type SuperPrint = {
```

```
    (arr:number[]):void
    (arr: boolean[]):void
    // (arr:string[]): void     //좋지 않은 예시
}

const superPrint:SuperPrint = (arr) => {
    arr.forEach(i => console.log(i))
}

superPrint([1,2,3,4])
superPrint([true, false, true])
superPrint(["a", "b", "c"])
superPrint([1,2 ,true, false])


//변형
//superPrint함수는 많은 형태를 가짐
//모든 형태의 배열을 입력 받을 수 있음
type SuperPrint = {
    <TypePlaceholder>(arr:TypePlaceholder[]):TypePlaceholder
    <T>(arr:T[]):T  //이런 형식으로 많이 사용함
}

const superPrint:SuperPrint = (arr) => arr[0]

const a = superPrint([1,2,3,4])
const b = superPrint([true, false, true])
const c = superPrint(["a", "b", "c"])
const d = superPrint([1,2 ,true, false, "hello"])
```

overloading(오버로딩)

```
// overloading(오버로딩)

//call signature를 길게 작성할 수 있는 방법
type Add = {
    (a: number, b: number): number
    (a: number, b: string): number
}

const add: Add = (a, b) => {
    if(typeof b === "string") {
        return a
    }
    return a + b
```

```
}


//argument수가 다른 오버로딩
type Add = {
    (a: number, b: number): number
    (a: number, b: number, c: number): number,
}

const add:Add = (a,b, c?: number) => {
    if(c) {
        return a + b + c
    }
    return a + b
}

add(1,2)
add(1,2,3)
```

generics

```
//generic
//제네릭은 기본적으로 placeholder를 사용하여 작성한 코드의 타입 기준으로 바꿔준다
//따라서 any[]를 사용하지 않는다.
type SuperPrint = <T>(a :T[]) => T
type SuperPrint1 = <T, M>(a :T[], b:M) => T //인자가 여러 개인 generic

const superPrint:SuperPrint = (a) => a[0]
const superPrint1:SuperPrint1 = (a) => a[0]

const a = superPrint([1,2,3,4])
const b = superPrint([true, false, true])
const c = superPrint(["a", "b", "c"])
const d = superPrint([1,2 ,true, false, "hello"])

const a1 = superPrint1([1,2,3,4],"X")
const b1 = superPrint1([true, false, true],false)
const c1 = superPrint1(["a", "b", "c"],1)
const d1 = superPrint1([1,2 ,true, false, "hello"],[])

//오류 발생 CASE
// type SuperPrint = (a:any[]) => any
// const superPrint: SuperPrint = (a) => a[0]
```

```
// const d = superPrint([1,2,true, false, "hello"])
// d.toUpperCases()    //⇒에러발생(배열의 첫번째 요소를 리턴하여, number을 받기 때문)
```

Conclusions_실제 사용

```
function superPrint <T> (a :T[]) {
   return a[0]
}

//에러 발생
// const a = superPrint<boolean>([1,2,3,4])
//앞에서 generic으로 boolean 형식을 주었기 때문에
//뒤에 같은 타입을 사용하는 인자 a 또한 boolean이 되어야 하기 때문에 에러가 발생
const a = superPrint([1,2,3,4])
const b = superPrint([true, false, true])
const c = superPrint(["a", "b", "c"])
const d = superPrint([1,2 ,true, false, "hello"])

// //1단계
 type Player<E> = {
    name : string
    extraInfo:E
 }

 const nico: Player<{favFood:string}> = {
    name : "nico",
    extraInfo : {
       favFood: "Kimchi"
    }
 }

//2단계
type Player<E> = {
   name : string
   extraInfo:E
}

type NicoExtra = {
   favFood:string
}
type NicoPlayer = Player<NicoExtra>

const nico: NicoPlayer = {
   name : "nico",
```

```
    extraInfo : {
        favFood: "Kimchi"
    }
}

const lynn: Player<null> = {
    name:"lynn",
    extraInfo:null
}


//////////////////////////////////////
type A = Array<number>

let a:A = [1,2,3,4]

function printAllNumbers (arr: Array<number>){

}
```