CIS 371 Web Application Programming
TypeScript IV

Lecturer: Dr. Yong Zhuang

TypeScript Functions (& Lambdas)

Important Takeaway Concept

- *Assigned to a variable*
- *Passed as an argument to another function*
- *Returned as a value from other functions*

JS & TS allow variables of type Function

JS & TS variables can hold either data or code

- *JS & TS variables can be assigned typical data values like numbers, strings, and objects,*
- *or they can be assigned functions*

Three variations of Function Declarations

```
function plus2 (a:number, b:number): number {
    return a + b;
}
```
named

```
const plus2 = function (a:number, b:number): number {
    return a + b;
}
```
anonymous func

```
const plus2 = (a:number, b:number) : number => {
    return a + b;
}
```
lambda function

Any of these function declarations can be invoked using ONE syntax:

```
let out:number;
out = plus2(5.0, 2.9);
```

Vars of "function" type

typeless AND 1-line return contraction

```
const plus2 = (a, b) => a + b
```

GRAND VALLEY
STATE UNIVERSITY

Fat Arrow fns: single-line return contraction

```
const plusTwo = (a:number, b:number) : number => {
    const sum = a + b;
    return sum;
}
```

no 'function' keyword.

```
const plusTwo = (a:number, b:number) : number => {
    return a + b;
}
```

If 'return' can be the only statement

omit both the curly braces { } and the 'return' keyword.

```
const plusTwo = (a:number, b:number) : number => a + b;
const plusTwo = (a,b) => a + b;   // typeless
```

implicit return

Variables of func type

plus20 and plus22 are variables that hold your DATA

```typescript
const plus20 = "+20";
const plus22 = { positive: true, value: 22 }
```

```typescript
const plus2 = function (a:number, b:number): number {
    return a + b;
}

const plusTwo = (a:number, b:number) : number => {
    return a + b;
}
```

```typescript
console.log(typeof plus20); // string
console.log(typeof plus22); // object
console.log(typeof plus2);   // function
console.log(typeof plusTwo); // function
```

*plus2 and plusTwo are variables that hold your **CODE***

GRAND VALLEY
STATE UNIVERSITY

Functions as Arguments
(to another Fn)

Array.sort()

```
const atoms = ["Neon", "Iron", "Calcium", "Hydrogen"]
console.log(atoms.sorted())
// ["Calcium", "Hydrogen", "Iron", "Neon"]
```

```
const primes = [23, 17, 5, 101, 19]
const sorted_nums = primes.sort()
console.log(sorted_nums)
```

[101, 17, 19, 23, 5]

Array.prototype.sort()

The `sort()` method of `Array` instances sorts the elements of an array *in place* ⧉ and returns the reference to the same array, now sorted. The default sort order is ascending, built upon converting the elements into strings, then comparing their sequences of UTF-16 code units values.

Online Doc

Array.sort() builtin behavior

[23, 17, 5, 101, 19]

implicit conversion to string

["23", "17", "5", "101", "19"]

sort()

["101", "17", "19", "23", "5"]
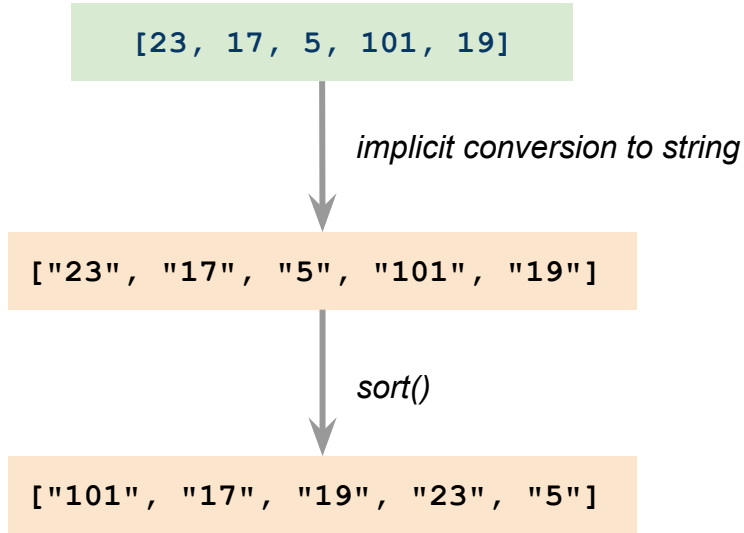
To fix this *"bug"*, we have to tell sort() the **collating order between two data items**

Array.sort() with collating order

```typescript
function numericOrder(a:number, b:number): number {
    if (a < b) return -1;              // any negative number
    else if (a > b) return +1;         // any positive number
    else return 0;
}
```

```typescript
const primes = [23, 17, 5, 101, 19]
const sorted_nums = primes.sort(numericOrder)
console.log(sorted_nums) // [5, 17, 19, 23, 101] Ok
```

The collating function must return a **number**
- *Negative when the "first" item should be placed BEFORE the "second" item*
- *Positive when the "first" item should be placed AFTER the "second" item*
- *Zero when the order of the two items is irrelevant*

Array.sort() on objects

```typescript
type Language = {
    name: string;  yearCreated: number
}
const langs: Language[] = [
    { name: "C", yearCreated: 1970},
    { name: "JavaScript", yearCreated: 1995},
    { name: "Fortran", yearCreated: 1954}
]
function orderByName(a:Language, b:Language): number {
    return a.name.localeCompare(b.name)
}
function orderByYear(a:Language, b:Language): number {
    return a.yearCreated - b.yearCreated
}
langs.sort(orderByName)
```

*The collating function takes two parameters of type Language but must **return a number***

Array.sort() on objects

```typescript
type Language = {
    name: string;  yearCreated: number
}
const langs: Language[] = [
    { name: "C", yearCreated: 1970},
    { name: "JavaScript", yearCreated: 1995},
    { name: "Fortran", yearCreated: 1954}
]
function orderByName(a:string, b:string): number {
    return a.localeCompare(b)
}
function orderByYear(a:number, b:number): number {
    return a - b
}
langs.sort(orderByName)
```

The collating function must take two parameters of type Language

Array.sort() on objects

```typescript
type Language = {
    name: string;  yearCreated: number
}
const langs: Language[] = [
    { name: "C", yearCreated: 1970},
    { name: "JavaScript", yearCreated: 1995},
    { name: "Fortran", yearCreated: 1954}
]
```

```typescript
langs.sort(
    function (a:Language, b:Language): number {
        return a.name.localeCompare(b.name)
    }
)
```
Option 2: unnamed function

```typescript
langs.sort(
    (a:Language, b:Language): number => {
        return a.name.localeCompare(b.name)
    }
)
```
Option 3: lambda function

```typescript
function orderByName(a:Language, b:Language): number {
    return a.name.localeCompare(b.name)
}
langs.sort(orderByName)
```
Option 1: named function

```typescript
langs.sort(
    (a, b) => a.name.localeCompare(b.name)
)
```
Opt 4: typeless lambda & 1-line return contraction

Function Optional Parameters/Arguments

```typescript
// whoAmI can be called with 2, 3, or 4 args
const whoAmI = (name: string, age: number, occupation?: string, spouse?: string): void => {
    console.log("Work as", occupation);
    console.log("Spouse name:", spouse ?? "N/A")
}
```

```typescript
whoAmI("Andy", 22);                         // Work as undefined
                                            // Spouse name: N/A

whoAmI("Bob", 43, "banker");                // Work as banker
                                            // Spouse name: N/A

whoAmI("Chuck", 31, undefined, "Cindy");    // Work as undefined
                                            // Spouse name Cindy

whoAmI("Chuck", 31, null, "Cindy");         // Work as null
                                            // Spouse name Cindy
```

GRAND VALLEY
STATE UNIVERSITY

Function Parameter Default Value

```typescript
const whoAmI = (name: string, age: number, occupation: string = "Student", spouse?: string):
void => {
    console.log("Work as", occupation);
    console.log("Spouse name:", spouse ?? "N/A")
}
```

```typescript
whoAmI("Andy", 22);                        // Work as undefined
                                           // Spouse name: N/A

whoAmI("Bob", 43, "banker");               // Work as banker
                                           // Spouse name: N/A

whoAmI("Chuck", 31, undefined, "Cindy");   // Work as Student
                                           // Spouse name Cindy

whoAmI("Chuck", 31, null, "Cindy");        // Work as null
                                           // Spouse name Cindy
```

Array Operations

Array high-order functions

- Array.every(), Array.some()

- Array.find(), findIndex()

- Array.filter(), Array.map(), Array.flatmap()

- Array.forEach()

- Array.reduce()

- … and **many others**

- flatMap() is available in ES2019

```
// tsconfig.json {
  "compilerOptions": {
    "target": "ES2019",
    // other options go here
  }
  ...
}
```

Array high-order functions

```typescript
type Shape = {
  color: string;
  numSides: number;
  sideDims: Array<number>; // the length of each side
};
```

```typescript
let shapes: Array<Shape> = [_____]
```

Array.some(): do we have any green shape?

shapes.some (?????) ⟶ **Yes**

```typescript
function isGreen(s: Shape): boolean {
    return s.color === "green"
}

const someGreen = shapes.some(isGreen);
console.log(someGreen);  // true
```

```typescript
const someGreen = shapes.some(function (s: Shape): boolean {
  return s.color === "green";             // Anonymous function
});

const someGreen = shapes.some((s: Shape): boolean => {
  return s.color === "green";             // Anonymous fat arrow
});

const someGreen = shapes.some((s: Shape): boolean => s.color === "green");
                                           // 1 line return elimination
const someGreen = shapes.some((s) => s.color === "green");
                                           // No explicit type
```

- *Purpose: Test if at least one element in the array passes the test implemented by the provided function.*
- *Return value: **A Boolean** (true if at least one passes the test, otherwise false).*

GRAND VALLEY
STATE UNIVERSITY

Array.some(): do we have any green shape?

shapes.some (?????)　　　→　　Yes

Incorrect!!!

```typescript
function isGreen(col: string): boolean {
  return col === "green";
}

const someGreen = shapes.some(isGreen);
console.log(someGreen); // true
```

// isGreen must take a Shape as its input parameter
// NOT a string!!!

Array.every(): are all shapes triangle?

`shapes.every (?????)`

```typescript
function isTriangle(s: Shape): boolean {
  return s.numSides === 3;
}


const allTriangle = shapes.every(isTriangle);
console.log(allTriangle); // false
```

```typescript
const allTriangle = shapes.every(function (s: Shape): boolean {
  return s.numSides === 3;          // Anonymous function
});

const allTriangle = shapes.every((s: Shape): boolean => {
  return s.numSides === 3;          // Anonymous fat arrow
});

const allTriangle = shapes.every((s: Shape): boolean => s.numSides === 3);
                                                         // 1 line return elimination

const allTriangle = shapes.every((s) => s.numSides === 3);
                                                         // No explicit type
```

- Purpose: Tests whether all elements in the array pass the test implemented by the provided function.
- Return value: **A Boolean** (true if every element passes the test, otherwise false).

Array.forEach(): inspect all shapes


```
shapes.forEach (?????)
```

```typescript
function printShape(s: Shape): void {
  console.log("# of sides", s.numSides);
}

shapes.forEach(printShape);
```

```typescript
shapes.forEach(function (s: Shape): void {
  console.log("# of sides", s.numSides);
});                          // Anonymous function

shapes.forEach((s: Shape): void => {
  console.log("# of sides", s.numSides);
});                          // Anonymous fat arrow

shapes.forEach((s) => {
  console.log("# of sides", s.numSides);
});                          // No explicit type
```

- *Purpose: Executes a provided function once for each array element.*
- *Return value:* **undefined.**

Array.findIndex(): where is ...?


```typescript
function isTriangle(s: Shape): boolean {
  return s.numSides === 3;
}

const idxTriangle = shapes.findIndex(isTriangle);
console.log(idxTriangle); // 1
```

shapes.findIndex (?????)

- *Purpose: To find the index of the **first element** in the array that satisfies a provided testing function.*
- *Return value: the index of the **first element** in the array that passes the test. If **no elements** pass the test, it returns **-1**.*

```typescript
const idxTriangle = shapes.findIndex(function (s: Shape): boolean {
  return s.numSides === 3;
});

const idxTriangle = shapes.findIndex((s: Shape): boolean => {
  return s.numSides === 3;
});

const idxTriangle = shapes.findIndex((s: Shape): boolean => s.numSides === 3);

const idxTriangle = shapes.findIndex((s) => s.numSides === 3);
```