CIS 371 Web Application Programming

TypeScript III

Lecturer: Dr. Yong Zhuang

Based on the original version by Professor Hans Dulimarta.

Recall

- Objects: Typeless, Typed, Sub-Objects, For-in loop to enumerate object
- Array of Typed Objects: Typeless, Typed, Sub-Objects
- Spreading:
  - Array, Array Destructuring,
  - Object, with duplicate props, copy and modify object
- Optional Chaining (?) operator & Function Optional Parameters
- Coalesce operator (??) & non-null assertion operator (!)
- Logical OR (||) operator
- Enum vs. Literal Types
- String Interpolation
- ES6 key/value Shortcut

Type Alias vs. Interface

```ts
type Book = {
  title: string;
  author: string;
};

const novel: Book = {
  title: "To Kill a Mockingbird",
  author: "Harper Lee",
};
```

```ts
interface Book {
  title: string;
  author: string;
};

const novel: Book = {
  title: "To Kill a Mockingbird",
  author: "Harper Lee",
};
```

Type Alias vs. Interface

```ts
type Book = {
  title: string;
  author: string;
};

type Book = {
  pages: number;
};
```

Error: Duplicate identifier 'Book'.

```ts
const novel: Book = {
  title: "To Kill a Mockingbird",
  author: "Harper Lee",
};
```

```ts
interface Book {
  title: string;
  author: string;
};

interface Book {
  pages: number;
};

const novel: Book = {
  title: "To Kill a Mockingbird",
  author: "Harper Lee",
  pages: 281,
};
```

Adding new fields to an existing interface can be really handy when you're extending 3rd party libraries.

Type Alias vs. Interface

```ts
type Book = {
  title: string;
  author: string;
};

type Book = {
  pages: number;
};
```

Error: Duplicate identifier 'Book'.

```ts
const novel: Book = {
  title: "To Kill a Mockingbird",
  author: "Harper Lee",
};
```

```ts
type Book = {
  title: string;
  author: string;
};

type Novel = Book & {
  pages: number;
};

const novel: Novel = {
  title: "To Kill a Mockingbird",
  author: "Harper Lee",
  pages: 281,
};
```

Type Alias vs. Interface

- A type cannot be re-opened to add new properties
- An interface which is always extendable.

Online Doc

Inheritance

```ts
// Base interface for common properties
interface Book {
  title: string;
  author: string;
  pages: number;
  price: number;
}

// Extending Book for Physical Book
interface PhysicalBook extends Book {
  coverType: "Hardcover" | "Paperback";
}

// Extending Book for Digital Book
interface DigitalBook extends Book {
  format: "PDF" | "EPUB" | "MOBI";
}
```

```ts
const novel: Book = {
  title: "To Kill a Mockingbird",
  author: "Harper Lee",
  pages: 281,
  price: 56,
};

const hardcoverBook: PhysicalBook = {
  title: "1984",
  author: "George Orwell",
  pages: 328,
  coverType: "Hardcover",
  price: 56,
};

const eBook: DigitalBook = {
  title: "Sapiens",
  author: "Yuval Noah Harari",
  pages: 498,
  format: "EPUB",
  price: 35,
};
```

```ts
function purchase(book: Book) {
  console.log(book.price);
}

purchase(novel);
purchase(hardcoverBook);
purchase(eBook);
```

Class

```
enum coverType {
  "Hardcover",
  "Paperback",
}

class Book {
  title: string;
  author: string;
  pages: number;
  price: number;
  coverType: coverType;
  purchase() {
    console.log(this.price);
  }
}

const novel = new Book();
novel.purchase();
```

```
class Book {
  title: string;
  author: string;
  pages: number;
  price: number;
  coverType: coverType | undefined;
  constructor(title: string, author: string, pages: number, price: number) {
    this.title = title;
    this.author = author;
    this.pages = pages;
    this.price = price;
  }
```

Error: Property '...' has no initializer and is not definitely assigned in the constructor..

```
const novel = new Book("To Kill a Mockingbird", "Harper Lee", 281, 56);
novel.coverType = coverType.Hardcover;
novel.purchase();
```

Inheritance

```
class Book {
  title: string;
  author: string;
  pages: number;
  price: number;

  constructor(title: string, author
    this.title = title;
    this.author = author;
    this.pages = pages;
    this.price = price;
  }
}
```

```
class DigitalBook extends Book {
  fileSize: number; // File size in MB
  format: string; // Format like PDF, EPUB, etc.

  constructor(
    title: string,
    author: string,
    pages: number,
    price: number,
    fileSize: number,
    format: string
  ) {
    // Call the parent class constructor with the common properties
    super(title, author, pages, price);
    this.fileSize = fileSize;
    this.format = format;
  }
}
```

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

typeless AND 1-line return contraction

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

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

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

Vars of "function" type

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

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

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

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

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

Functions as Arguments (to another Fn)

Array.sort()

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

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

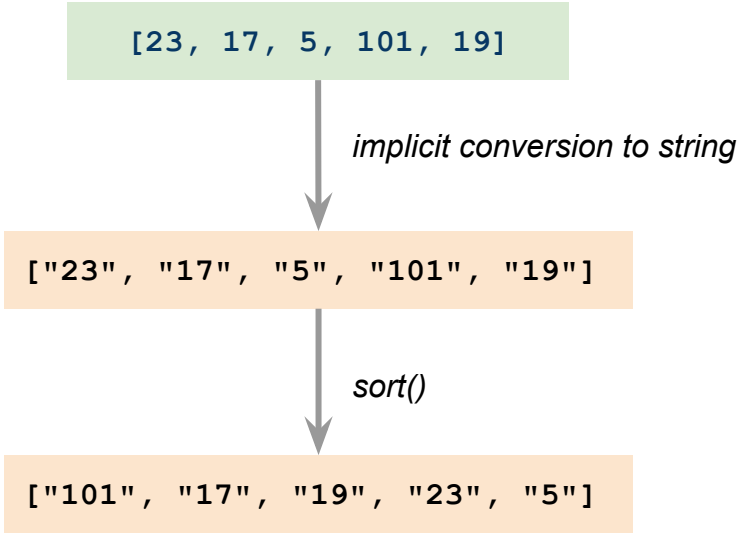
[101, 17, 19, 23, 5]

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



To fix this ***"bug"***, we have to tell sort() the **collating order between two data items**

Array.sort() with collating order

```
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;
}
```

```
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

```
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(orderByYear)
```

ascending or descending?

- *Negative when the referenceStr occurs before compareString*
- *Positive when the referenceStr occurs after compareString*
- *Returns 0 if they are equivalent*

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

Array.sort() on objects

```
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)
}
langs.sort(orderByName)
```

Option 1: named function

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

Option 2: unnamed function

```
langs.sort(
    (a:Language, b:Language): number => {
        return a.name.localeCompare(b.name)
    }
)
```

Option 3: lambda function

```
langs.sort(
    (a, b) => a.name.localeCompare(b.name)
)
```

Opt 4: typeless lambda & 1-line return contraction