CIS 658 Web Architectures

TypeScript V

Lecturer: Dr. Yong Zhuang

Based on the original version by Professor Hans Dulimarta.

Recall

- Functions as Arguments (to another Fn) : High-order functions
  - Array.reduceRight()
  - Array.some()
  - Array.every()
  - Array.forEach()
  - Array.find(), Array.findIndex()
  - Array.filter()
  - Array.map() ,Array.flatmap()

Practice

Answer

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'.

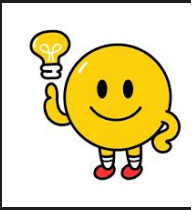
```ts
const novel: Book = {
  title: "To Kill a Mockingbird",
  author: "Harper Lee",
};
```

```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.



Inheritance: Type

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

type Book = {
  pages: number;
};

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

Error: Duplicate identifier 'Book'.

```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;
  }
}
```

Generics in TypeScript

Suppose we define a copy function named **`clone`**.

```
const clone = (value: any) => {
  const json = JSON.stringify(value);
  return JSON.parse(json);
};
```

```
const cloneBook = clone(novel);
```

```
console.log(cloneBook.);
```

Why does VS Code not provide auto-complete for object properties?

```
interface Book {
  title: string;
  author: string;
  pages: number;
  price: number;
}

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

Generics in TypeScript

Suppose we define a copy function named **clone**.

```
const clone = (value: any) => {
  const json = JSON.stringify(value);
  return JSON.parse(json);
};
```

The return type of the function is **any**, so VS Code cannot provide property auto-completion.

```
const cloneBook = clone(novel);
```

```
console.log(cloneBook.);
```

Why does VS Code not provide auto-complete for object properties?

```
interface Book {
  title: string;
  author: string;
  pages: number;
  price: number;
}

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

Generics in TypeScript

Suppose we define a copy function named **clone**.

```
const clone = (value: Book): Book =>  {
  const json = JSON.stringify(value);
  return JSON.parse(json);
};
```

Explicit type

```
const cloneBook = clone(novel);
```

```
console.log(cloneBook.);
```

- author
- pages
- price
- title

Is this solution good enough?

Not flexible enough for all object types

```
interface Book {
  title: string;
  author: string;
  pages: number;
  price: number;
}

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

Generics in TypeScript

With generics, we tell TypeScript that the type of the parameter is also the type returned by the function.

```
const clone = <T>(value: T): T => {
  const json = JSON.stringify(value);
  return JSON.parse(json);
};
```

Generic type

You can use any name for the generic type, but the most common convention is **T**.

```
const cloneBook = clone(novel);
```

```
console.log(cloneBook.);
```

author
pages
price
title

```
interface Book {
  title: string;
  author: string;
  pages: number;
  price: number;
}

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

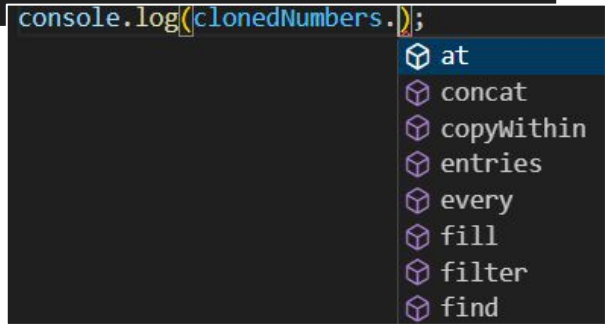
Generics in TypeScript

With generics, we tell TypeScript that the type of the parameter is also the type returned by the function.

```ts
const clone = <T>(value: T): T => {
  const json = JSON.stringify(value);
  return JSON.parse(json);
};
```

Generic type

```ts
const numbers = [1, 2, 3];
const clonedNumbers = clone(numbers);
console.log(clonedNumbers.);
```




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

const books: Book[] = [
  { title: "1984", author: "George Orwell" },
  { title: "The Hobbit", author: "J.R.R. Tolkien" },
];

const clonedBooks = clone(books);
console.log(clonedBooks[0].title); // Output: "1984"
```

Practice 1: Interface

Practice 2: Generic Type