4

Classes, Interfaces and Enums

Class, Instance objects, Inheritance, Implementation, Generics, Enums

Classes

- Supports
  - Instance members
  - Static members
  - Access modifiers
  - Constructors
  - Interface implementation
  - Class inheritance

Classes

- Class definition
  - Fields
  - Constructors
  - Functions

- Use of ‘**`class`**’ keyword

```
class class_name {
   //class scope
}
```

Classes

Fields

- A field is any variable declared in a class
- No `var`, `let` or `const` keyword

```
class Car {
   //field
   engine: string;
}
```

Classes

Constructor

- Use **constructor** keyword
- We cannot define multiple constructors

```
class Car {
   //field
   engine:string;

   //constructor
   constructor(engine:string) {
      this.engine = engine
   }

}
```

Classes

Functions

- Actions a class can take, also referred as methods
- No use of **function** keyword

```
class Car {

   //function
   buy(owner: IPerson) : void {
      //...
  }

}
```

Classes

- Access modifiers
  - Public, Private and Protected
  - Public by default

```
class Car {
    public price: number;
    private priceWithVAT: number;
}
```

  - Does nothing at runtime, only a compile-time check

Classes

- Readonly modifier
  - Makes a member readonly
  - It needs to be assigned at creation of the object

```
class Car {
    public readonly price: number;
    private priceWithVAT: number;
}
```

Classes

- Parameter property declaration

```
public name: string;

constructor(name: string) {
  this.name = name;
}
```

⇩

```
constructor(public name: string) { }
```

Automatically creates property
and sets value
(public, private, protected, readonly)

Classes

Getters

- Use **get** keyword followed by a function expression
- We cannot pass an argument to the getter method
- A getter must return a value

```
class Shape {

   _color : string

   get color() {
      return this._color
   }
}
```

Classes

Getters

- Although getters are methods, they are not invokable
- They are accessed just like a property

```
console.log(shape.color)
```

Classes

Setters

- Use **set** keyword followed by a function expression
- A setter must have exactly **one** argument
- If a setter return a value, it is ignored

```
class Shape {

   _color : string

   set color(value:string) {
      this._color= value
   }
}
```

Classes

Setters

- Setters are invoked when the property is assigned

```
//Setting color. Runs the setter method
shape.color="red";
```

- The property that is set by the setter method is called the backing property

Classes

Instance objects

- The new keyword is responsible for instantiation
- The right-hand side of the expression invokes the constructor

```
var instance_name = new CLASS_NAME([ arguments ])

var car = new Car("BMW")
```

Classes

Accessing fields and functions

- Accessing class instances is the same for objects

```
//accessing a property
obj.field_name


//invoking a function
obj.function_name()
```

Classes

Class inheritance

- Typescript supports the concept of inheritance
- Inheritance allows to extend a class from another class called base/super class
- Use **extends** keyword

```
class Circle extends Shape {

}
```

```
class Square extends Shape {

}
```

Classes

Class inheritance

- Use `super()` to call the constructor of the base class

```
class Employee extends Person {

   constructor( firstName: string, lastName: string, jobTitle: string) {
       // call the constructor of the Person class:
       super(firstName, lastName);
   }

}
```

Classes

Class inheritance

- A child (derived) class method may or may not use the logic defined in the parent (base) class method
- Redefine the superclass's method by using the **same name and arguments**
- **NO `override`** keyword

```
class Rect extends Shape {
    area(length:number, width:number) {
        return length*width
    }
}
```

```
class Square extends Shape {
    area(length:number, width:number) {
        return length*length
    }
}
```

Generics

- Create reusable components (classes, functions, interfaces, ,,,)
- Works with multiple types rather than a single one
- Use **any** type ?

```
function identity(arg: any): any {
   return arg;
 }
```

- Lose of type information, type-checking, intellisense
- Lose of control over the accepted types

Generics

- Capture type information via **type variable**
- Use **<Type>** syntax

```
function identity<T>(arg: T): T {
   return arg;
 }
```

- On inspection, the return type is the same for the input type

Generics

Functions

- Call generic functions in one of two ways
- Explicitly (Pass in the type argument)

```
let output = identity<string>("myString");
```

- Type inference

```
let output = identity("myString"); //type of output is string
```

Generics

Types

- Generics in type aliases or interfaces allows to creeate reusable types

```
interface Wrapped<T> {
   value: T
};
```

```
type Wrapped<T>  = {
   value: T
};
```

```
var a: Wrapped<number> = {value: 10}   //OK
var b: Wrapped<string> = {value: 20}   //Error
```

Generics

Types

- We can use multiple type arguments

```
interface KeyPair<T, U> {
   key: T;
   value: U;
}

let kv1: KeyPair<number, string> = { key:1, value:"Steve" }; // OK
let kv2: KeyPair<number, number> = { key:1, value:12345 }; // Error
```

Generics

Classes

- Typescript also support generics for classes
- A class can have generic members (fields of functions)

```
class KeyValuePair<T,U>
{
    private key: T;
    private value: U;

    setKeyValue(key: T, value : U): void {
        this.key = key;
        this.value = value;
    }
}
```

Generics

Generic constraints

- Put constraints over the type arguments
- Narrowing down the accepted types
- Use of `extends` statement

```
type Lengthwise = {
   length: number;
 };
 function getLength<T extends Lengthwise>(arg: T): number {
   return arg.length;
 }
```

- Only variables having `length` property are accepted

Interfaces

- Interfaces are a structure that defines a contract
- In Typescript we can use Interfaces to define a types or to implement a class
- Use of **`interface`** keyword

```
interface KeyPair {
   key: number;
   value: string;
}

let kv1: KeyPair = { key:1, value:"Steve" }; // OK
let kv2: KeyPair = { key:1, val:"Steve" }; // Compiler Error
```

Interfaces

- Interfaces can extend one ore more interfaces
- Use **extends** keyword

```
interface IPerson {
   name: string;
   gender: string;
}
```

```
interface IEmployee extends IPerson {
   empCode: number;
}
```

Interfaces

- Interfaces can be implemented with a class
- The class needs to strictly conform to the structure of the interface
- Use **`implements`** keyword

```
interface IEmployee {
   empCode: number;
   name: string;
   getSalary:(empCode: number) => number;
}
```

Interfaces

- Example

```
class Employee implements IEmployee {
   empCode: number;
   name: string;

   constructor(code: number, name: string) {
       this.empCode = code;
       this.name = name;
   }

   getSalary(empCode:number):number {
       return 20000;
   }
}

let emp = new Employee(1, "Steve");
```

Enum

- Give more friendly names to sets of named constants, a collection of related values
  - Numeric enums
  - String enums
  - Heterogeneous enums

- Use **enum** keyword

```
enum Status {
  Active,
  Deactivate,
  Pending
}
```

Enum

Numeric enums

- Store values as numbers
- Srarting from **0** by default
- Incremented for each member

```
enum PrintMedia {
   Newspaper,
   Newsletter,
   Magazine,
   Book
 }
```

```
Newspaper = 0
Newsletter = 1
Magazine = 2
Book = 3
```

Enum

- **Numeric enums**
  - We can explicitly set the numeric value for an option
  - The other options are incremented by 1

```
enum PrintMedia {
   Newspaper = 1,
   Newsletter = 3,
   Magazine,
   Book
 }
```

```
Newspaper = 1
Newsletter = 3
Magazine = 4
Book = 5
```

Enum

String enums

- Store values as string literals
- String values offer better readability
- Must be explicitly set

```
enum PrintMedia {
   Newspaper = "newspaper",
   Newsletter = "newsletter",
   Magazine = "magazine",
   Book = "book"
}
```

Enum

- **Heterogenous enums**
  - Can contain both numeric or string values

```
enum Status {
   Active = 'ACTIVE',
   Deactivate = 1,
   Pending
}
```