6 Working With Classes

Objectives

After completing this lesson, you should be able to do the following:

- Classes in TypeScript
- Readonly Property
- Public, Private & Protected Access Modifiers
- Constructor Method
- Inheritance
- Getters & Setters
- Structural Typing & Duck Typing
- Declaring Static Members in Typescript
- Abstract Classes in Typescript

Typescript Objects

Typescript Objects

- Objects are the collection of key/value pairs that can be modified throughout the lifecycle of an object as similar to hash map or dictionary. Objects allow us to define custom data types in JavaScript.
- Unlike primitive data types, we can represent complex or multiple values using objects. The values can be the array of objects or the scalar values or can be the functions. Data inside objects are unordered, and the values can be of any type.
- An object can be created by using curly braces **{...}** along with an optional properties list. The property is a **"key:value"** pair, where the key is a string or a property name, and the value can be anything.

Creating Objects

- **Creating an empty object:**

```
var obj_1 = {};
var obj_2 = new Object();
var obj_3 = Object.create(null);
```

- **Creating an object:**

```
var person = {
    "full-name" : "John Doe",
     age: 35,
     address: {
       address_line1: "Clear Trace, Glaslyn, Arkansas",
       "postal code": "76588-89"
     }
};
```

Creating Objects

- **Creating an object with `new`:**

```
function Tree(type1, height1, age1) {
    this.type = type1;
    this.height = height1;
    this.age = age1;
}

var mapleTree = new Tree("Big Leaf Maple", 80, 50);
```

Creating an object with prototypes (syntax):

```
var obj = Object.create(Object.prototype [,properties]);
```

Examples:

```
var myChild = Object.create(Object.prototype); //Object {}
myChild = Object.create({a:10, b:30}); // Object {a=10, b=30}

var myObj = {
    a : 10,
    b : 30
};
myChild = Object.create(myObj);
myChild.a;        //10
myChild.b;        //30
```

Accessing Object Properties

```
var myObject = {
    name: "luggage",
    length: 75,
    specs: {
      material: "leather",
      waterProof: true
    }
}
```

- **Getting and setting properties:**

```
myObject["name"];                    // "luggage"
myObject.name;                       // "luggage"
myObject.specs.material;             // "leather"
myObject["specs"]["material"];       // "leather"
myObject.width;                      // undefined
myObject.tags.number;                // TypeError thrown
```

```
myObject["name"] = "suitcase";       // name : "suitcase"
myObject.name = "suitcase";          // name : "suitcase"
myObject.width = 40;                 // creates a new property
myObject.tags.number = 6;            // TypeError thrown
```

Working with Object Properties

- Deleting properties:

```
delete myObject.length;
delete myObject.height;
```

true, even though it doesn't exist in the object

- Testing and enumerating properties:

```
var myObject = {
    one: "property value one",
    two: "property value two"
}
```

```
for (var myVar in myObject) {
    if (myObject.hasOwnProperty(myVar)) {
      console.log(myVar);
    }
}
```

Accessing Object Properties

- **Property `get` and `set`:**

```
var obj = {
    a : 45,
   get double_a() {return this.a * 2},
   set modify_a(x) {this.a -= x;}
};
```

```
obj.a;              // 45
obj.double_a;       // 90
obj.modify_a = 40;
obj.a;              // 5
obj.double_a;       // 10
```

- **Object method example:**

```
var myObj = {
    print: function() {
      console.log("Hello World!");
    }
};

myObj.print();
```


TS
Typescript Classes

Classes

- Classes are an essential part of object-oriented programming (OOP). Classes are used to define the blueprint for real-world object modeling and organize the code into reusable and logical parts.
- Before ES6, it was hard to create a class in JavaScript. But in ES6, we can create the class by using the **class** keyword. We can include classes in our code either by class expression or by using a class declaration.
- A class definition can only include **constructors** and **functions**. These components are together called as the data members of a class. The classes contain **constructors** that allocates the memory to the objects of a class. Classes contain **functions** that are responsible for performing the actions to the objects.

Syntax and Illustration

Syntax: Class Declaration

```
class Class_name{
}
```

```
class Student{
    constructor(name, age){
    this.name = name;
    this.age = age;
    }

}
```

Instantiating an Object from class

Syntax

```
var obj_name = new class_name([arguements])
```

Example

```
var stu = new Student('Peter', 22)
```

Accessing functions

- The object can access the attributes and functions of a class. We use the **'.' dot notation (**or **period)** for accessing the data members of the class.

Syntax

```
obj.function_name();
```

Example

```
'use strict'
class Student {
  constructor(name, age) {
    this.n = name;
    this.a = age;
  }
  stu() {
    console.log("The Name of the student is: ", this.n)
    console.log("The Age of the student is: ",this. a)
  }
}

var stuObj = new Student('Peter',20);
stuObj.stu();
```

Classes

- When building large scale apps, the object oriented style of programming is preferred by many developers, most notably in languages such as Java or C#.
- TypeScript offers a class system that is very similar to the one in these languages, including inheritance, abstract classes, interface implementations, setters/getters, and more.

- Classes are used to define blueprint of objects. They are used extensively in OOPs based languages.
- Though JavaScript didn’t have direct support for classes till ES6, we were still able to create functionality similar to classes using the *prototype* property of the objects.
- TypeScript had support for classes since its inception.
- Classes in TypeScript can have properties and members as instance variables. Their access to the outside world can be controlled using access specifiers.
- The class can have a constructor to initialize its members.

```
class Employee{
  private bonus: number;

  constructor(private empNo: string, private name: string, private salary: number){
    this.bonus = this.salary * 0.1;
  }

  getDetails(){
    return `Employee number is ${this.empNo} and name is ${this.name}`;
  }

  get Name() {
   return this.name;
  }

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

- As the arguments passed to the constructor are marked as private, they are made members of the class and they receive the values that are passed to the constructor.
- The function *getDetails* returns a string containing details of the employee.
- The default access specifier in TypeScript is public.
- The field bonus shouldn’t be accessed directly using the object, so it is explicitly made private.

Accessors

- TypeScript supports getters/setters as a way of intercepting accesses to a member of an object. This gives you a way of having finer-grained control over how a member is accessed on each object.

Code Snippet …

```
let passcode = "secret passcode";

class Employee {
    private _fullName: string;

    get fullName(): string {
        return this._fullName;
    }

    set fullName(newName: string) {
        if (passcode && passcode == "secret passcode") {
            this._fullName = newName;
        }
        else {
            console.log("Error: Unauthorized update of employee!");
        }
    }
}

let employee = new Employee();
employee.fullName = "Bob Smith";
if (employee.fullName) {
    console.log(employee.fullName);
}
```

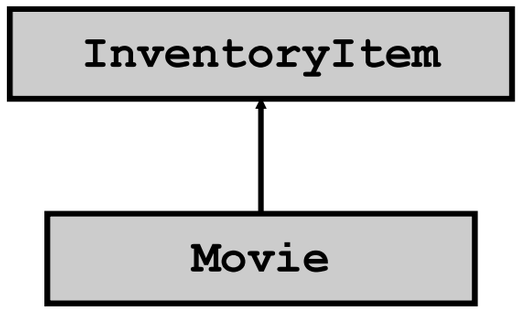
```
class Menu {
  // Our properties:
  // By default they are public, but can also be private or protected.
  items: Array<string>;  // The items in the menu, an array of strings.
  pages: number;         // How many pages will the menu be, a number.

  // A straightforward constructor.
  constructor(item_list: Array<string>, total_pages: number) {
    // The this keyword is mandatory.
    this.items = item_list;
    this.pages = total_pages;
  }

  // Methods
  list(): void {
    console.log("Our menu for today:");
    for(var i=0; i<this.items.length; i++) {
      console.log(this.items[i]);
    }
  }

}

// Create a new instance of the Menu class.
var sundayMenu = new Menu(["pancakes","waffles","orange juice"], 1);

// Call the list method.
sundayMenu.list();
```

Public, private, and protected modifiers

- Public is Default
- Private Accessible Only Within the Class
- Protected modifier acts much like the private modifier with the exception that members declared protected can also be accessed within deriving classes.
- Readonly modifier can make properties readonly by using the readonly keyword. Readonly properties must be initialized at their declaration or in the constructor.

Example of Inheritance

- The `InventoryItem` class defines methods and variables.



- `Movie` extends `InventoryItem` and can:
  - Add new variables
  - Add new methods
  - Override methods in the `InventoryItem` class

Specifying Inheritance in Java

- Inheritance is achieved by specifying which superclass the subclass extends.

```
public class InventoryItem {
   …
}
```

```
public class Movie extends InventoryItem {
   …
}
```

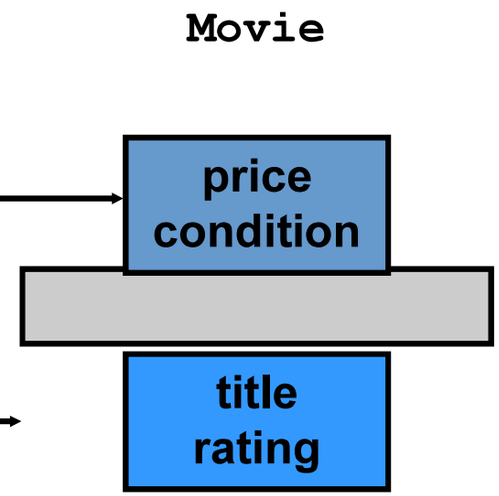
- `Movie` inherits all the variables and methods of `InventoryItem`.
- If the `extends` keyword is missing, `Object` is the implicit superclass.

Subclass and Superclass Variables

- A subclass inherits all the instance variables of its
- superclass.

```
public class InventoryItem {
  private price:number;
  private condition: string; …
}
```

```
public class
 Movie extends InventoryItem {
  private title: string ;
  private rating: string; …
}
```

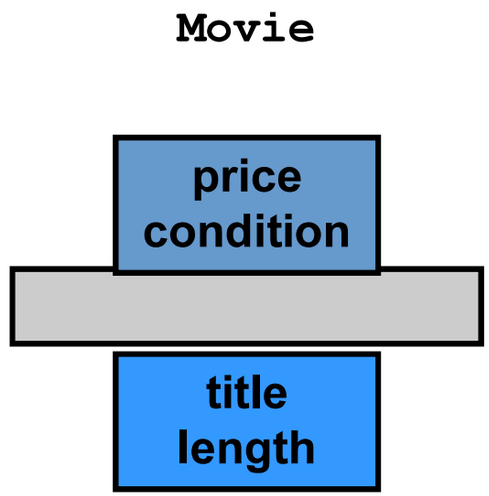


Default Initialization

- What happens when a subclass object is created?

```
Movie movie1 = new Movie();
```

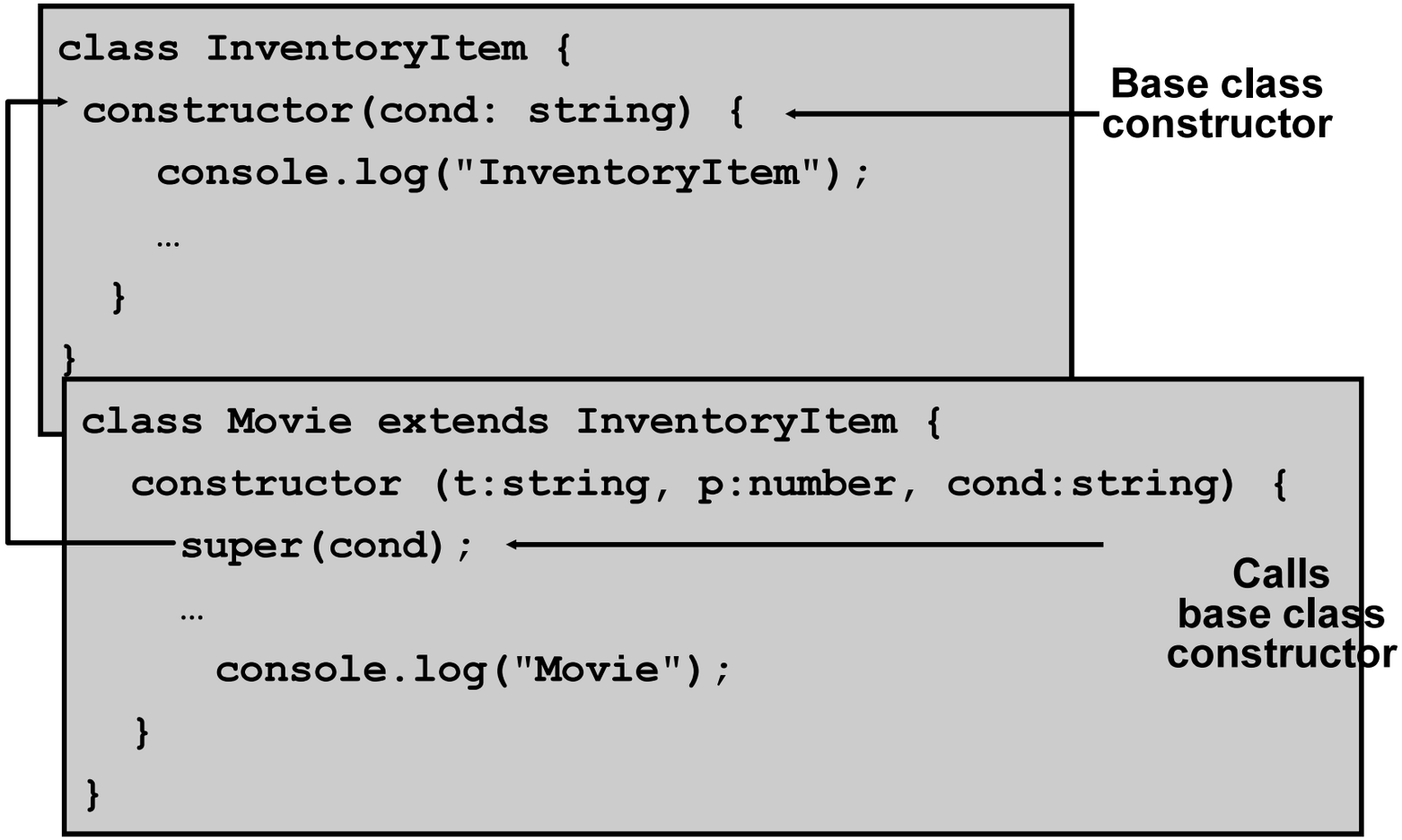
- If no constructors are defined:
  1. The default `no-arg` constructor is called in the superclass.
  2. The default `no-arg` constructor is then called in the subclass.



Super() Reference

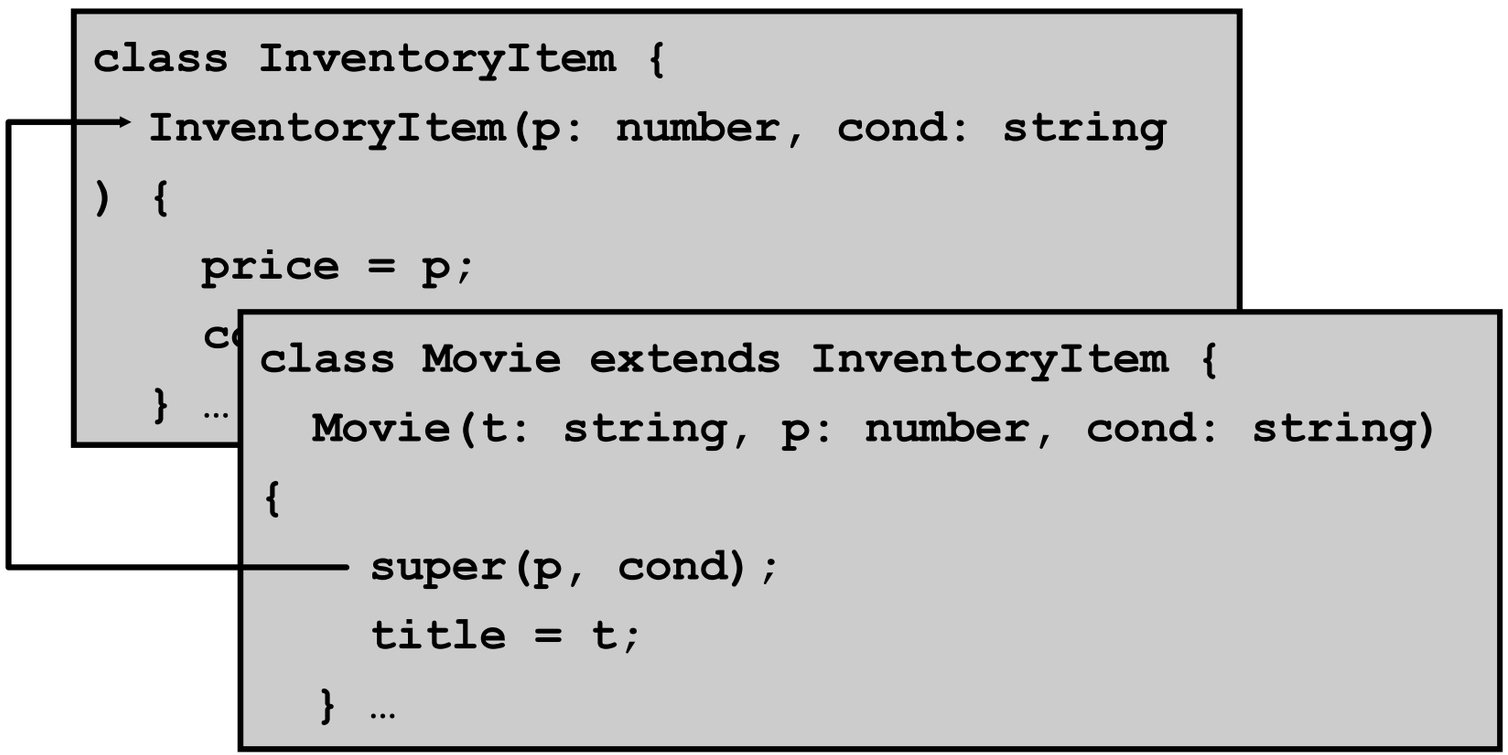
- Refers to the base superclass
- Is useful for calling base class constructors
- Must be the first line in the derived class constructor

Super() Reference: Example



Using Superclass Constructors

- Use `super()` to call a superclass constructor:



Specifying Additional Methods

- The superclass defines methods for all types of `InventoryItem`.
- The subclass can specify additional methods that are specific to `Movie`.

```
class InventoryItem {
calcDeposit():number …
calcDateDue():string…
  …
```

```
class Movie extends InventoryItem {
getTitle(): void…
getLength(): string…
```

Inheritance

```
class HappyMeal extends Menu {
  // Properties are inherited

  // A new constructor has to be defined.
  constructor(item_list: Array<string>, total_pages: number) {
    // In this case we want the exact same constructor as the parent class (Menu),
    // To automatically copy it we can call super() - a reference to the parent's constructor.
    super(item_list, total_pages);
  }

  // Just like the properties, methods are inherited from the parent.
  // However, we want to override the list() function so we redefine it.
  list(): void{
    console.log("Our special menu for children:");
    for(var i=0; i<this.items.length; i++) {
      console.log(this.items[i]);
    }

  }
}

// Create a new instance of the HappyMeal class.
var menu_for_children = new HappyMeal(["candy","drink","toy"], 1);

// This time the log message will begin with the special introduction.
menu_for_children.list();
```

```
class Employee{
  private bonus: number;

  constructor(protected empNo: string, protected name: string, protected salary: number){
    this.bonus = this.salary * 0.1;
  }

  getDetails(){
    return `Employee number is ${this.empNo} and name is ${this.name}`;
  }

  get Name() {
   return this.name;
  }

  set Name(name: string){
   this.name = name;
  }

  get Bonus(){
    return this.bonus;
  }
}
```

```
class Manager extends Employee {
  constructor(empNo: string, name: string, salary: number, private noOfReportees: number) {
    super(empNo, name, salary);
  }

  getDetails(){
    var details = super.getDetails();
    return `${details} and has ${this.noOfReportees} reportees.`;
  }
}
```

Invoking Superclass Methods

- If a subclass overrides a method, it can still call the original superclass method.
- Use `super.method()` to call a superclass method from the subclass.

```
class InventoryItem {
calcDeposit(int custId): Number {
    if
    ret
  }
```

```
class Vcr extends InventoryItem {
calcDeposit(int custId) : Number {
    itemDeposit = super.calcDeposit(custId);
    return (itemDeposit + vcrDeposit);
}
```

Treating a Subclass As Its Superclass

- A object instance of a subclass is assignable to its superclass definition.
  - You can assign a subclass object to a reference that is declared with the superclass.

```
InventoryItem item = new Movie();
  double deposit = item.calcDeposit();
```

  - The compiler treats the object via its reference (that is, in terms of its superclass definition).
  - The run-time environment creates a subclass object, executing subclass methods, if overridden.

Static Properties

- We can also create *static* members of a class, those that are visible on the class itself rather than on the instances.

```
class Grid {
    static origin = {x: 0, y: 0};
    calculateDistanceFromOrigin(point: {x: number; y: number;}) {
        let xDist = (point.x - Grid.origin.x);
        let yDist = (point.y - Grid.origin.y);
        return Math.sqrt(xDist * xDist + yDist * yDist) / this.scale;
    }
    constructor (public scale: number) { }
}

let grid1 = new Grid(1.0);  // 1x scale
let grid2 = new Grid(5.0);  // 5x scale

console.log(grid1.calculateDistanceFromOrigin({x: 10, y: 10}));
console.log(grid2.calculateDistanceFromOrigin({x: 10, y: 10}));
```

Abstract Classes

- Abstract classes are base classes from which other classes may be derived.
- They may not be instantiated directly. Unlike an interface, an abstract class may contain implementation details for its members.
- The abstract keyword is used to define abstract classes as well as abstract methods within an abstract class.

```
abstract class Animal {
    abstract makeSound(): void;
    move(): void {
        console.log("roaming the earth...");
    }
}
```

- Methods within an abstract class that are marked as abstract do not contain an implementation and must be implemented in derived classes.
- Abstract methods share a similar syntax to interface methods. Both define the signature of a method without including a method body.
- However, abstract methods must include the abstract keyword and may optionally include access modifiers.

```typescript
abstract class Department {

    constructor(public name: string) {
    }

    printName(): void {
        console.log("Department name: " + this.name);
    }

    abstract printMeeting(): void; // must be implemented in derived classes
}

class AccountingDepartment extends Department {

    constructor() {
        super("Accounting and Auditing"); // constructors in derived classes must call super()
    }

    printMeeting(): void {
        console.log("The Accounting Department meets each Monday at 10am.");
    }

    generateReports(): void {
        console.log("Generating accounting reports...");
    }
}
```

```
// ok to create a reference to an abstract type
let department: Department;
// error: cannot create an instance of an abstract class
department = new Department();
// ok to create and assign a non-abstract subclass
department = new AccountingDepartment();
department.printName();
department.printMeeting();
// error: method doesn't exist on declared abstract type
department.generateReports();
```

Summary

In this lesson, you should have learned how to:

- Classes in TypeScript
- Readonly Property
- Public, Private & Protected Access Modifiers
- Constructor Method
- Inheritance
- Getters & Setters
- Structural Typing & Duck Typing
- Declaring Static Members in Typescript
- Abstract Classes in Typescript