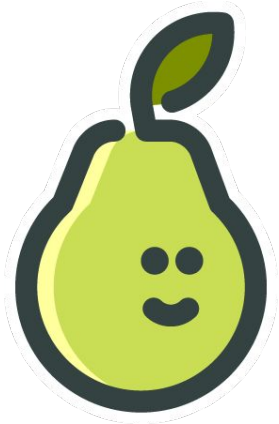
Classes, Interfaces and Generics

Session-2



Did you finish pre-class material?

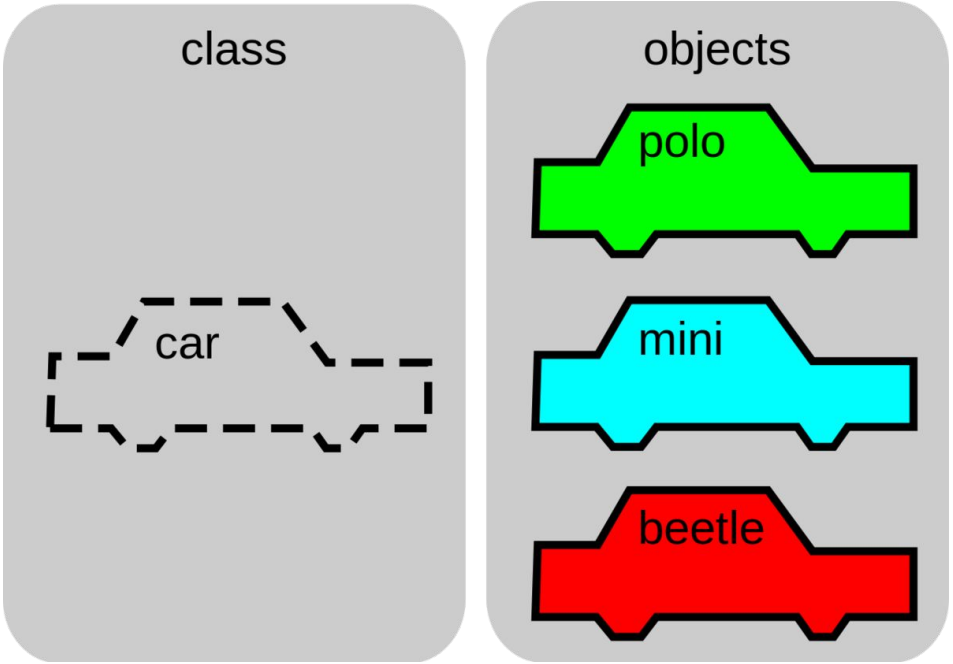
Session Topics

- Classes
- Interfaces
- Generics

Classes

Classes

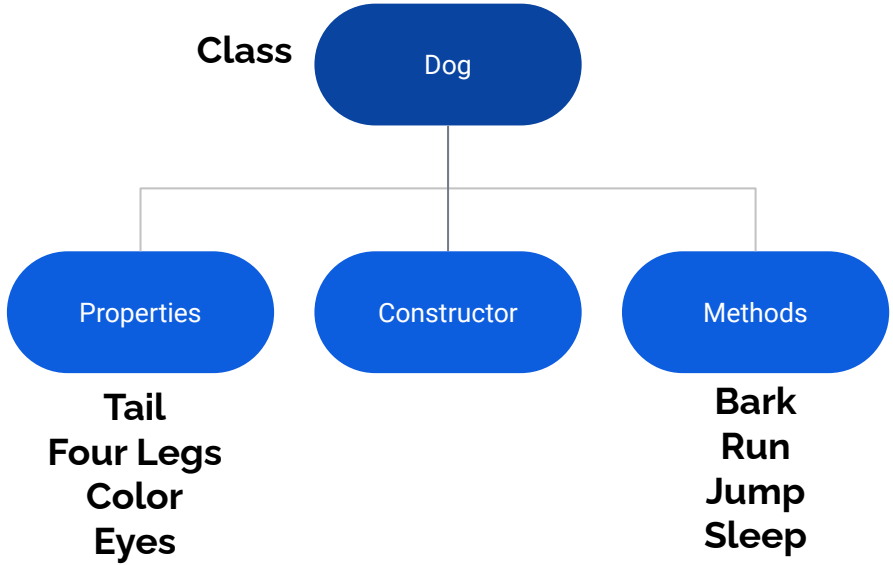
- Classes are the fundamental entities used to create reusable components.
- Functionalities are passed down to classes.
- Objects are created from classes.



Classes

A class can include the following:

- Constructor
- Properties
- Methods



Classes

- An object of the class can be created using the new keyword.
- While instantiating a new object, the class constructor is called with the values passed.

```
class Employee {
  empCode: number;
  empName: string;

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

let emp = new Employee(100, "John")
```

Classes - Inheritance

- Classes can inherit using the keyword **extends.**
- The employee class now includes all the members of the person class.
- The constructor of the employee class is using a special keyword super, and initialises its own properties.
- We must call super() method first before assigning values to properties in the constructor of the child class.

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

class Employee extends Person {
  empCode: number;
  constructor(code: number, name:string) {
    super(name);
    this.empCode = code
  }
  displayName():void {
    console.log(this.name, this.empCode)
  }
}
```

Abstract Classes

- Typescript allows us to define an abstract class using keyword abstract.
- Abstract classes are mainly for defining structure of class, where there are no implementation code. Therefore, they cannot be instantiated.
- An abstract class typically includes one or more abstract methods or property declarations. **The child class must define all the abstract methods.**
- We can think as abstract methods/properties are a placeholder that will be defined when it is inherited.

Abstract Classes

- Child of an abstract class must call super() in the constructor.
- Person is an abstract class has one property and two methods. displayCode() method is an abstract method and so must be defined in the derived class.
- The Employee class should define implementation code for the displayCode() method.
- The Employee class should implement all the abstract methods of the Person class, otherwise the compiler will give an error.

```
abstract class Person {
  name: string;
  constructor(name: string) {
    this.name = name
  }
  displayName():void {
    console.log(this.name)
  }
  abstract displayCode():void;
}

class Employee extends Person {
  empCode: number;
  constructor(code: number, name:string) {
    super(name);
    this.empCode = code
  }
  displayCode(): void {
    console.log(this.empCode)
  }
}
```

Class Data Modifiers

- In object-oriented programming, the concept of 'encapsulation' is used to make class members public or private i.e.
- A class can control the visibility of its data members. This is done using access modifiers.
- There are three types of access modifiers in typescript:
  - **public**
  - **private**
  - **protected**
- By default, all members of a class in Typescript are public. All the public members can be accessed anywhere without any restrictions.

Class Data Modifiers - Private

- The private access modifier ensures that class members are visible only to that class and are not accessible outside the containing class.
- When we create an object emp and try to access the emp.empCode member, it will give an error.

```
class Employee {
  private empCode: number;
  empName: string;

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

let emp = new Employee(100, "John")
console.log(emp.empCode) // Compiler Error
```

Class Data Modifiers - Protected

- Child class can't access **private** methods/ properties.
- However, child class can access **protected** members.
- In this example, `DevelopmentEmployee` class can't access `id` in the parent class. However, it can reach public and protected properties.
- Outside the class, we can reach `empName`, but we can't reach `department`.

```
class Employee {
  public empName: string;
  protected empCode: number;
  private id = Math.random();
  constructor(empCode: number, name: string) {
    this.empCode = empCode;
    this.empName = name
  }
}
class DevelopmentEmployee extends Employee {
  private department: string;

  constructor(code: number, name: string, dep: string) {
    super(code, name);
    this.department = dep;
    this.id = 100 // Compiler Error. Private to parent
  }
}

let emp = new DevelopmentEmployee(100, "John", "Backend")
console.log(emp.department) // Compiler Error
```

Readonly Modifier

- In addition to the access modifiers, typescript provides two more keywords: read-only and static.
- Prefix read-only is used to make a property as read-only.
- Read-only members can be accessed outside the class, but their value cannot be changed after initialization.
- If we try to change the value of empcode after the object has been initialized, we get compiler error.

```
class Employee {
  empName: string;
  readonly empCode: number;

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

let emp = new Employee(100, "John")
emp.empCode = 20 // Compiler Error
```

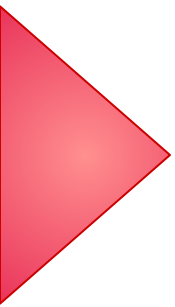
Static Modifier

- The static members of a class are accessed using the class name and dot notation, **without creating an object.**
- The static members can be defined by using the keyword static.
- Circle class includes a static property and a static method.
- We can call these methods and properties directly from the class.

```
class Circle {
  static pi: number = 3.14;

  static calcArea(radius:number) {
    return this.pi * radius * radius
  }
}

Circle.pi; // returns 3.14
Circle.calcArea(5); // return 78.5
```



Interfaces

Interfaces

- Interface is a structure that defines the **contract**. It defines the syntax for classes to follow. Classes must follow the derived interface structure.
- The typescript compiler does not convert interface to JavaScript. Just for type checking. "duck typing" or "structural subtyping".
- An interface is defined with the keyword interface
- It can only include properties and method declarations.
- No implementation

Interfaces

- We use interfaces to define the shape of objects and classes.

```
interface Calendar {
  events: string[];
  addEvent(event: string): void;
}

class LmsCalendar implements Calendar {
  events: string[];
  constructor() {
    this.events = [];
  }
  addEvent(event: string): void {
    this.events.push(event)
  }
}
```

Interfaces vs Type Aliases

- Interfaces and type aliases can be used interchangeably. Both can be used to describe the shape of an object.
- Interface can only be used for object and classes.

```
interface Person {
  name: string;
}

let person: Person = {
  name: "John"
}
```

```
type Person = {
  name: string;
}

let person: Person = {
  name: "John"
}
```

Interfaces vs Type Aliases

- We can add more properties to interfaces later inside our code, whereas in type aliases can't.
- In type aliases we need to union previous type alias into new alias.

```
interface Point {
  x: number;
}

interface Point {
  y: number
}

const point: Point = {x: 1, y: 2}
```

```
type PointX = {
  x: number;
}

type PointY = {
  y: number
}

type Point = PointX & PointY

const point: Point = {x: 1, y: 2}
```

Interfaces

- Interface can only contain declaration of properties and methods, no value or no implementation details are allowed.
- A class can implement more than one interface, but can only implement one super class.

```
interface Color {
  color: { r: number, g: number, b: number }
}

interface Shape {
  area: number;
}

class Square implements Color, Shape {
  color: { r: number, g: number, b: number }
  area: number;
  constructor(c: { r: number, g: number, b: number }, a: number) {
    this.color = c;
    this.area = a;
  }
}
```

Interfaces as Function Type

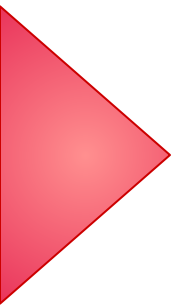
- We can define function type using interface, to assign a function a variable, typescript can enforce specific function.

```
interface NumKey {
  (key: number, value: string): void
}

function addKeyVal(k: number, v: string): void {
  console.log("adding key value", k, v)
}

function update(index: number, newVal: string): void {
  console.log("updating key value", index, newVal)
}

let kvp: NumKey = addKeyVal;
kvp(1, 'John')
kvp = update
kvp(2, 'Jane')
```



Generics

Generics

- TypeScript Generics is a tool which provides a way to create reusable components (functions, interfaces and classes)
- Generics in typescript is almost similar to C# generics.
- A generic type has one or more generic type parameters in angle brackets. e.g.: <T> or <T,U> using uppercase single letter is a convention
- When using generic types, we should supply arguments for generic type parameters or let the compiler infer them (if possible).

Generics

Problem

```
function getArray(items: any[]): any[] {
  return new Array().concat(items)
}
let numArr = getArray([1, 2, 3])
let strArr = getArray(["John", "Jane"])
numArr.push(4); // OK
strArr.push("Jake"); // OK
numArr.push("Tim"); // OK
strArr.push(5); // OK
console.log(numArr); // [ 1, 2, 3, 4, 'Tim' ]
console.log(strArr); // [ 'John', 'Jane', 'Jake', 5 ]
```

Solution

```
function getArray<T>(items: T[]): T[] {
  return new Array<T>().concat(items)
}
let numArr = getArray([1, 2, 3])
let strArr = getArray(["John", "Jane"])
numArr.push(4); // OK
strArr.push("Jake"); // OK
numArr.push("Tim"); // Compiler Error
strArr.push(5); // Compiler Error
```

Generics

Generic Interface

```
interface Result<T> {
  data: T | null;
}
```

Generic Function

```
function wrapInArr<T>(value: T) {
  return [value]
}

let arr = wrapInArr(1)
```

Generic Classes

```
class KeyValuePair<K,V> {
  constructor(public key: K, public value: V) {}
}

let kvp = new KeyValuePair<number, string>(1,'a')
let shorter = new KeyValuePair(1,'a')
```

Generics

- Multiple generic parameter example

```
function displayType<T, U>(param1: T, param2: U) {
  console.log(`param1: ${typeof(param1)}, param2: ${typeof(param2)}`);

}
displayType<number, string>(34, "Istanbul");
displayType<string, number>("Price", 250);
displayType(console.log, 5 > 8);
```

- Single parameter with non-generic

```
function displayType<T>(param1: T, param2: string) {
  console.log(`param1: ${typeof(param1)}, param2: ${typeof(param2)}`);

}
displayType<number>(34, "Istanbul");
displayType<string>("Price", 250);
displayType(console.log, "5 > 8");
```

Generics Constraints

- We can constrain generic type arguments by using the **extends** keyword after generic type parameters. T extends Person or T

```
function echo<T extends number | string>(value: T) {}
// Restrict using a shape object
function echo<T extends { name: string }>(value: T) {}
// Restrict using an interface or a class
function echo<T extends Person>(value: T) {}
```

```
// Passing on generic type parameters
class CompressibleStore<T> extends Store<T> { }
// Constraining generic type parameters
class SearchableStore<T extends { name: string }> extends Store<T> { }
// Fixing generic type parameters
class ProductStore extends Store<Product> { }
```

Generics Constraints

- The **keyof** operator helps by producing a union of the keys of the given object. We can constrain parameters of a function to be only in given list, no new property.

```ts
interface Product {
 name: string;
 price: number;
}
let property: keyof Product;
// Same as
let property: 'name' | 'price';
property = 'name';
property = 'price';
property = 'otherValue'; // Invalid
```

```ts
interface Product {
 name: string;
 price: number;
}
function update<T extends object, K extends keyof T>
            (obj: T, prop: K, newValue: T[K]) {
  //
}
const product1: Product = {name: "Headphones", price: 50}
update(product1, "price", "Powerbank");  🤔
update(product1, "count", 5); 🤦‍♂️
```

Generics Utility Types

- Using type mapping we can create new types based off of existing types.
- For example, a new type with all the properties of another type where these properties are readonly, optional, etc.
- TypeScript comes with several utility types that perform type mapping for us.

  Examples are: Partial<T>, Required<T>, Readonly<T>, etc.
- See the complete list of utility types:

  https://www.typescriptlang.org/docs/handbook/utility-types.html

Generics Utility Types

Type mapping

```
type ReadOnly<T> = {
 readonly [K in keyof T]: T[K];
};
type Optional<T> = {
 [K in keyof T]?: T[K];
};
type Nullable<T> = {
 [K in keyof T]: T[K] | null;
};
```

Utility types

```
interface Product {
 id: number;
 name: string;
 price: number;
}
// A Product where all properties are optional
let product: Partial<Product>;
// A Product where all properties are required
let product: Required<Product>;
// A Product where all properties are read-only
let product: Readonly<Product>;
// A Product with two properties only (id and price)
let product: Pick<Product, 'id' | 'price'>;
// A Product without a name
let product: Omit<Product, 'name'>;
```

Generics Advantages

- **Type-safety**: Only a single type of objects in generics. It doesn't allow to store other objects.
- **Typecasting is not required**: no need to typecast the object.
- **Compile-Time Checking**: checked at compile time so prevents the problem at runtime.

THANKS!