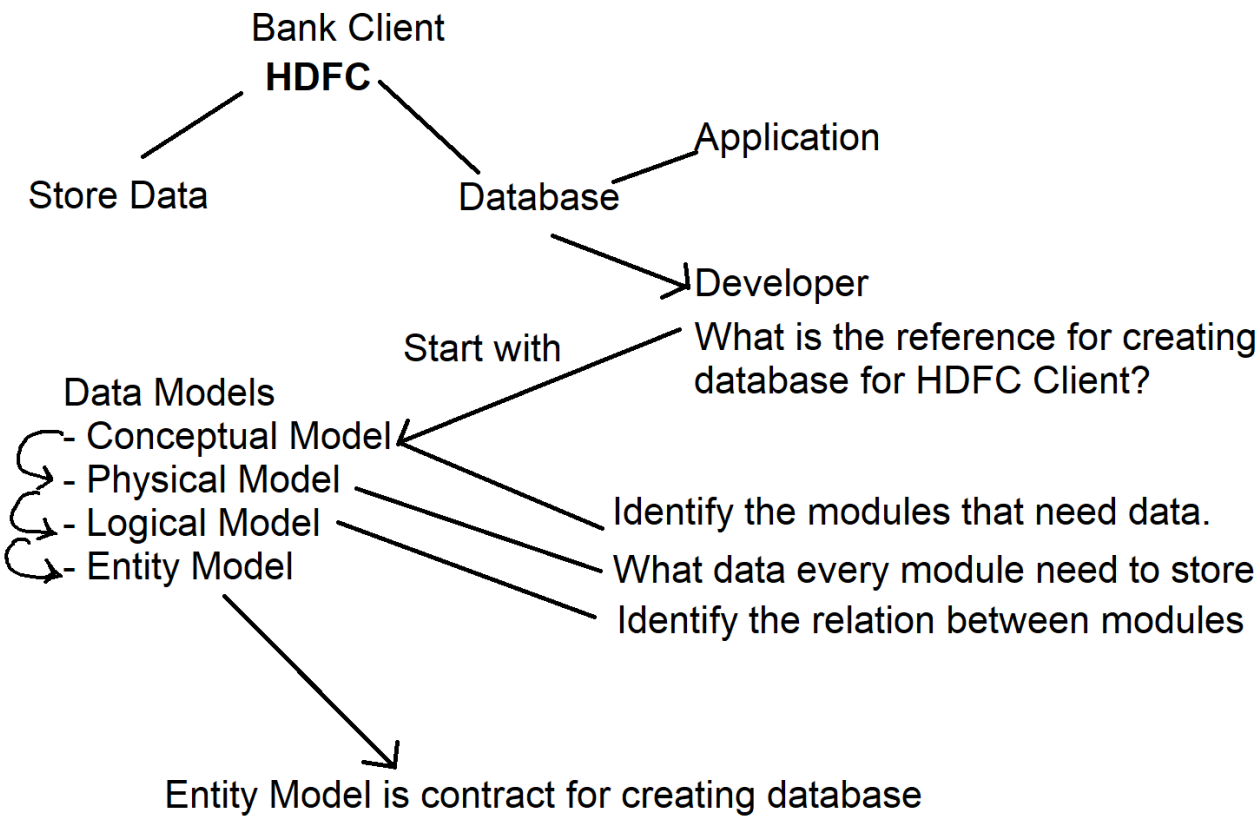
# Contracts in OOP

- In OOP every component is designed as per the specified contract.
- A contract defines set of rules for designing component.




- Technically contracts are defined as “Interfaces”.
- Interface is used to create a contract in OOP.
- Contract comprises of a set of rules.

- Interface defines a set of rules for designing components in OOP.
- The keyword “interface” is used to configure a contract.

  **Syntax:**

  interface ContractName
  {
  // set of rules
  }
- Every rule defined in a contract is mandatory to implement.

  *[You can also define optional rules. But by-default every rule is mandatory]*
- A contract can contain only declaration of rules not any implementation.

```
interface ContractName
{
  Name:string = “TV”;       // invalid
  Print():void                  // invalid
  {
   // some functionality;
  }
}
```

```
interface ContractName
{
  Name:string;      // valid
  Print():void;     // valid
}
```

- The contract member can’t have any access restriction or modification.
- You can’t use any access modifier for contract member.
- TypeScript supports the following access modifiers
    - public
    - private
    - protected
- The contract members will not have any access restriction.
- A contract can be implemented as Type, as Template, as Component directly.

Ex:

```
interface IProduct
{
   Name:string;
```

```
    Price:number;
    InStock:boolean;
    Qty:number;
    Total():number;
    Print():void;
}
let product:IProduct = {
    Name: "Samsung TV",
    Price: 45000.55,
    InStock:true,
    Qty:2,
    Total():number {
        return this.Qty * this.Price;
    },
    Print():void {
console.log(`Name=${this.Name}\nPrice=${this.Pri
ce}\nQty=${this.Qty}\nInStock=${this.InStock}\nT
otal=${this.Total()}`);
    }
```

```
}
product.Print();
```

**Optional Rules**

- Every rule defined in contract is mandatory to implement.
- However, we can configure optional rules in a contract.
- Every component will have an objective and goal.
- Objective must be achieved and time bound.
- Objective is non-nullable.
- Every member in a contract is by default non-nullable.
- Goal is not mandatory to achieve.
- It is optional.
- Contract can be defined with optional rules by using a null-reference character “?”.
- “?” is used to define a non-nullable rule into nullable.

Ex:

```
interface IProduct
```

```
{
    Name:string;
    Price:number;
    InStock:boolean;
    Qty:number;
    Mfd?:any;                    //optional
    Total():number;
    Print():void;
    Expiry?():any;               //optional
}
let product:IProduct = {
    Name: "Samsung TV",
    Price: 45000.55,
    InStock:true,
    Qty:2,
    Total():number {
        return this.Qty * this.Price;
    },
    Print():void {
```

```
console.log(`Name=${this.Name}\nPrice=${this.Price}\nQty=${this.Qty}\nInStock=${this.InStock}\nTotal=${this.Total()}`);
    }
}
product.Print();
```

**Note: Your component can't can have implementation for any property or method, which is not defined in contract.**

Ex:

```
interface IProduct
{
    Name:string;
    Price:number;
    InStock:boolean;
    Qty:number;
    Total():number;
    Print():void;
```

```
}
let product:IProduct = {
    Name: "Samsung TV",
    Price: 45000.55,
    InStock:true,
    Qty:2,
    Mfd: new Date("2020-02-10"),            // Invalid
– no Mfd in contract.
    Total():number {
        return this.Qty * this.Price;
    },
    Print():void {
console.log(`Name=${this.Name}\nPrice=${this.Pri
ce}\nQty=${this.Qty}\nInStock=${this.InStock}\nT
otal=${this.Total()}`);
    }
}
product.Print();
```

**Read Only Members in a Contract**

- You can re-define a value for every member in a contract.
- After defining a value into contract member, we don’t want to re-define.
- Once they are initialized with a value you can restrict re-defining value by using “readonly” as access specifier.

Ex:

```
interface IProduct
{
    Name:string;
    readonly Price:number;
    InStock:boolean;
    Qty:number;
    Total():number;
    Print():void;
}
let product:IProduct = {
    Name: "Samsung TV",
```

```
    Price: 45000.55,
    InStock:true,
    Qty:2,
    Total():number {
        return this.Qty * this.Price;
    },
    Print():void {
console.log(`Name=${this.Name}\nPrice=${this.Pri
ce}\nQty=${this.Qty}\nInStock=${this.InStock}\nT
otal=${this.Total()}`);
    }
}
product.Price = 65000.55;     // not allowed –
invalid – it is marked as readonly
product.Print();
```

## Extending Contracts

- A contract can be extended with rules.

- You can define new rules for the contract by extending the contract, without disturbing the existing contract.
- The keyword “extends” configures the relation between contracts and allows to extend a contract.
- The newly created and extended contract is known as “Derived Contract”.
- The existing contract is known as “Super Contract”.
- Finally, we can implement Derived contract to access the rules of both derived and super.

Syntax:

```
 interface Super
{
  // super rules
}
interface Derived extends Super
{
  // derived rules
}
```

Ex:

```
//Super Contract
interface IProduct
{
    Name:string;
    Price:number;
}
//Derived Contract
interface ICategory extends IProduct
{
    CategoryName:string;
}
let product:ICategory = {
    CategoryName: "Electronics",
    Name: "Samsung TV",
    Price: 34000.44
}
```

```
console.log(`Name=${product.Name}\nPrice=${product.Price}\nCategory=${product.CategoryName}`);
```

FAQ: Can we define multiple contracts for single object as Type?

A. No

Ex:

```
interface IProduct
{
    Name:string;
    Price:number;
}
interface ICategory
{
    CategoryName:string;
}
let product:ICategory | IProduct = {            // Invalid Declaration
    CategoryName: "Electronics",
```

```
    Name: "Samsung TV",
    Price: 34000.44
}
console.log(`Name=${product.Name}\nPrice=${product.Price}\nCategory=${product.CategoryName}`);
```

**Note: Every object can use only one contract as "Type". Hence, we have to configure relation between contracts. The relation for contract can be defined as:**

- Single
- Multiple
- Multi-Level

**Single:**

- There will be only one Super Contract.
- A Super Contract can be extended by multiple Derived Contracts.
- All Derived contracts are individual, they are not having any relation with each other.
- Your component can implement only the extended contract that it needs.

Ex:

```
interface Isuv
{
    Power:string;
}
interface IModel2019 extends Isuv
{
    Name:string;
    Features:string;
}
interface IModel2020 extends Isuv
{
    Name:string;
    Features:string;
}
let Xuv300: IModel2020 = {
    Power: "1450cc",
    Name: "XUV 300",
    Features: "18KMPL"
```

```
}
let Xuv100: IModel2019 = {
    Power: "1300cc",
    Name: "XUV 100",
    Features: "20KMPL"
}
console.log(`---- Model 2020 ------`);
for(var property in Xuv300)
{
    console.log(`${property} :
${Xuv300[property]}`);
}
console.log(`---- Model 2019 --------`);
for(var property in Xuv100)
{
    console.log(`${property} :
${Xuv100[property]}`);
}
```

**Multi-level**

- A derived contract is extended by another derived contract.
- It represents a multi-level hierarchy of relation between components.
- We generally use this type of extensibility when we are implementing an Incremental model of designing application.
- **The model should support backward compatibility**.

Ex:

```
interface Suv2019
{
   FogLamp:boolean;
   SunRoof:boolean;
}
interface Suv2020 extends Suv2019
{
   PowerStreeing:boolean;
}
interface Suv2021 extends Suv2020
```

```
{
    AlloyWheels:boolean;
}
let year:number = 2021;

switch(year)
{
    case 2019:
        var Xuv3002019:Suv2019 = {
            FogLamp:false,
            SunRoof:false,
        }
        console.log(`XUV 300 2019 Model: \n
FogLamp=${Xuv3002019.FogLamp}\nSunRoof=${X
uv3002019.SunRoof}`);
    break;
    case 2020:
        var Xuv3002020:Suv2020 = {
            FogLamp:true,
```

```
        SunRoof:true,
        PowerStreeing:true
    }
    console.log(`XUV 300 2020 Model:
\nFogLamp=${Xuv3002020.FogLamp}\nSunRoof=
${Xuv3002020.SunRoof}\nPowerSteering=${Xuv3
002020.PowerStreeing}`);
  break;
  case 2021:
    var Xuv3002021:Suv2021 = {
        FogLamp:true,
        SunRoof:true,
        PowerStreeing:true,
        AlloyWheels:true,
    }
    console.log(`XUV 300 2021 Model:
\nFogLamp=${Xuv3002021.FogLamp}\nSunRoof=
${Xuv3002021.SunRoof}\nPowerSteering=${Xuv3
002021.PowerStreeing}\nAlloyWheel=${Xuv30020
21.AlloyWheels}`);
```

```
    break;
}
```

Ex:

```
interface Suv2019
{
    FogLamp?:boolean;
    SunRoof?:boolean;
}
interface Suv2020 extends Suv2019
{
    PowerStreeing?:boolean;
}
interface Suv2021 extends Suv2020
{
    AlloyWheels?:boolean;
}
let car2019:Suv2019 = {FogLamp:true,
SunRoof:false };
```

```
let car2020:Suv2020 = {FogLamp:true,
SunRoof:true, PowerStreeing:true };
let car2021:Suv2021 = {FogLamp:true,
SunRoof:true, PowerStreeing:true,
AlloyWheels:true};


console.log(`--Car Features in 2019--`);
for(var property in car2019) {
   console.log(property);
}
console.log(`--Car Features in 2020--`);
for(var property in car2020) {
   console.log(property);
}
console.log(`--Car Features in 2021--`);
for(var property in car2021) {
   console.log(property);
}
```

## Multiple

- Single derived contract can extend multiple super contracts.
- If you don’t what to maintain any relation between existing super contracts and you want to create a new contract that extends all super contract then you can use “Multiple” approach.
- All super contracts are separated with “,”

Syntax:

interface derived extends super1, super2, ..

{

}

Ex:

```
interface Suv
{
   Seats:number;
   PowerStreeing:boolean;
}
interface Muv
{
```

```
    AlloyWheels:boolean;
}
interface Vehical extends Suv, Muv
{
    Name:string;
}
let car:Vehical = {
    Name: "Some Car",
    AlloyWheels:true,
    PowerStreeing:true,
    Seats:7
}
for(var property in car)
{
    console.log(`${property} : ${car[property]}`);
}
```

**FAQ: How a contract handles same rules defined while extending?**

- Extension of contract will not have any effect with same rules defined.
- TypeScript uses the rules from the first contract defined in Multiple approach.
- Extension of contract will be incorrect if same rule is used with different type of configuration.
- Rules must be identical.
- If rules are not identical can it will not support extending contract.
- It can have different rules but can’t have same rule with different configuration.

Ex:

```
interface Suv
{
 Name:string;
}
Interface Muv
{
 Name:string;
```

*}*

*Interface vehicle extends Suv, Muv*

*{*

*Name is taken from Suv*

*}*

*Interface vehicle extends Muv, Suv*

*{*

*Name is taken from Muv*

*}*

**FAQ: Can any object implement multiple contracts?**

**A.**No

**Can we use a contract directly for component?**

*Contracts are implemented by templates and these templates are used for components.*