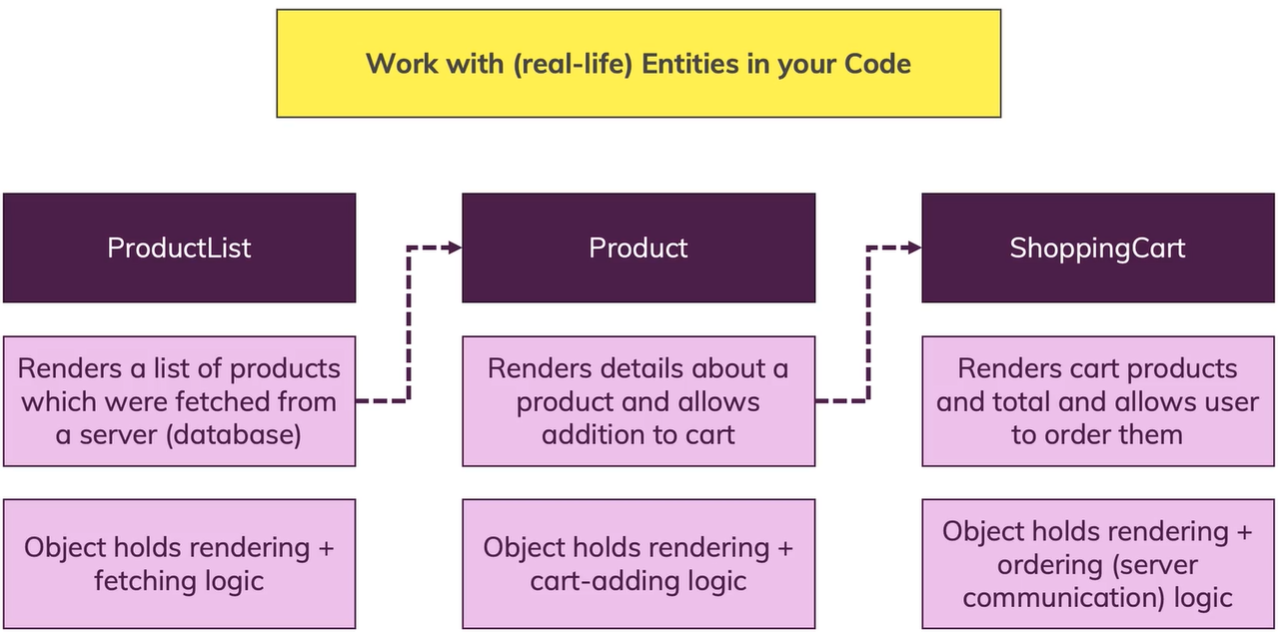
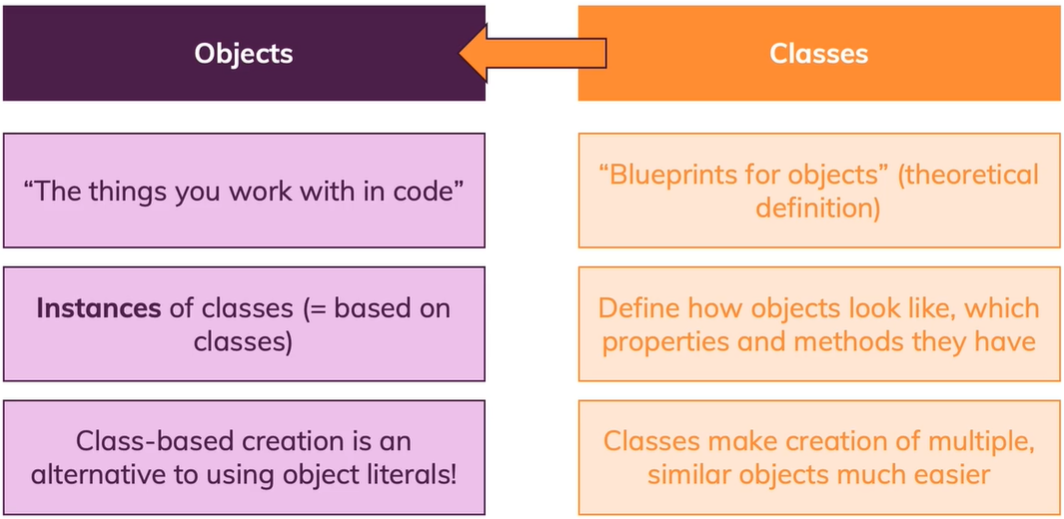
**Classes and Interfaces (Classes\_Interfaces\_Modifiers\_Inheritance\_Getters/Setters)**

1. **Classes and Interfaces:**
   1. What is [Object Oriented Programming](https://www.educative.io/blog/object-oriented-programming)?
      1. It is a programming paradigm, in which the application is structured around simple, reusable pieces of code (classes). Class templates are used as blueprints to create individual instances of objects.



* 1. How are classes different from objects (class instances)?



1. **Classes**:
   1. **Property** - A variable in a class.
   2. **Behavior -** A method in a class.
   3. **constructor** - Used to initialize instance variables.
   4. **Modifiers** (public, private, protected) – Used for data [encapsulation](https://www.educative.io/blog/object-oriented-programming).
      1. **Encapsulation** – Restricting access to class properties and methods from outside of the class (limits direct access, via an object instance).
      2. **private modifier** - Only accessible through public class methods.
      3. **public modifier** - Is the default. Accessible from outside the class.
      4. **protected modifier** - Makes variables accessible to subclasses.
      5. **readonly** - Properties that should not change after creation
   5. **Shorthand Initialization** - Gets rid of instance variable declaration, and constructor initialization. Add the modifiers directly to the params of a constructor.
      1. constructor(private id: string, public name: string) {}
2. **Inheritance:** It is possible to create a new class that inherits the basic structure from another class, by using the **keyword extends**. The new class may also contain new properties and behaviors. It allows you to share some common functionality and create more specialized blueprints.
   1. **Syntax:**

class Department {

constructor(private readonly id: string, public name: string){

}

class ITDepartment extends Department {

constructor(id: string, public admins: string[]) {

*super*(id,’IT’)

}

}

* 1. **Class Inheritance Rules:**
     1. **A class can only inherit (extend) from one class.**
     2. It is possible to call the base class constructor inside the constructor of the child class, using the keyword **super**. Super must be called first in the constructor of the child class.
  2. When to use inheritance?
     1. When you have a base set of properties that will be identical in other classes.

1. **Overriding and Protected Modifier:**
   1. **Override** - Creating a method that exists in the base (parent) class, in the subclass. However, the method can’t access private variables in the base class. The subclass methods can access protected and public properties only from its super class with this.propertyName.
   2. **Protected variables** - variables in the base class that can be inherited by subclasses.
   3. **Getters and setters** – A method to access and update private instance variables, via class properties.
      1. **Getters** – A public function that returns a value when executed. Although it is defined as a method, it is called by accessing the property on a class instance, using dot notation (without parenthesis).
         1. **Declaration Syntax:** 
            1. get mostRecentReport() { return this.lastReport}
         2. **Call Syntax:**
            1. console.log(“Last report:”, accounting.mostRecentReport);
      2. **Setters** - A public function that takes a parameter, which is used to set the value of a class instance variable. Although it is defined like a method, it is called by accessing it as a property on a class instance and assigning it a value.
         1. **Declaration Syntax:**

set mostRecentReport(value: string) {

this.reports.push(value);

}

* + - 1. **Call Syntax:**
         1. accounting.mostRecentReport = ‘Year End Report’;
  1. **Static Methods and Properties** - One variable/method that is available directly on the base class. They are not accessible through an instance, but directly off the class name (like Math.trunc).
     1. Static functions are often used as helper functions, and static properties are often used as global constants, stored in a class.
        1. **Declaration Static Property:**
           1. static fiscalYear = 2023;
           2. console.log(“Year:”, Department.fiscalYear);
        2. **Declare Static Method:**

static createEmployee(name: string) {

return {name: name};

}

* + 1. NOTE1: Since static members are not properties on the instance of the class, they cannot be invoked on an instance of a class.
       1. ~~Example: it.fiscalYear;~~
    2. NOTE2: Since static members are not properties on the instance of the class, they cannot be invoked inside the method of a non-static method with the “this” keyword. However, they can be invoked inside a class’s non-static method with the class name (className.propertyName).
       1. **Access in non-static methods:**
          1. className.propertyName
       2. **Access in Static Methods:**
          1. this.propertyName

1. **Abstract Classes:** Classes that will never be instantiated but define a basic blueprint that can be inherited and defined in subclasses. This will **enforce** the implementation of common methods and/or properties in the subclass. The base (parent) class will only contain the signature of the abstract method.
   1. **NOTE1: Abstract classes** may be a mixture of signatures and implemented methods.
   2. **NOTE2**: Classes with an abstract method signature, must be declared abstract
   3. Signature:

abstract class Department {

abstract describe(this: Department): void;

}

* 1. **NOTE2**: We still use the keyword **extends** when a subclass inherits from an abstract class.
     1. **Implementation in an inherited class:**

class AccountingDepartment extends Department {

discribe() {

console.log(‘Accounting Department – ID:’ + this.id);

}

}

* 1. **NOTE3: ABSTRACT CLASSES CAN NOT BE INSTANTIATED!** They are a blueprint for the subclasses that will inherit from them. Only the subclasses with concrete method implementations can be instantiated.
  2. **NOTE4:** Any instance variables that should be available to the subclass must be public or protected, not private.

1. **Singletons & Private Constructors: Singletons ensure there is only one instance of a certain class. It does not allow multiple instances of a class.**
   1. **Private Constructor** - Making a private constructor ensures that only one instance of a certain class is available.
   2. **How to access the single instance?** Create a static **getInstance** method, which is never instantiated, in the singleton class that will return one instance of the class.
      1. NOTE: “this” inside of a static method refers to the class name, since the method will never be accessed from an instantiated class.
   3. **Example:**

abstract class AccountingDepartment {

// Step 1:Create an instance variable of the class type

private static instance: AccountingDepartment;

constructor(protected readonly id: string, public name: string) {

}

// Step 2: Create a method that returns the class instance

static getInstance() {

if (this.instance) {

return this.instance;

}

this.instance = new AccountingDepartment('d2', []);

return this.instance;

}

}

// Step 3: Get the singleton instance

const accounting = AccountingDepartment.getInstance();

1. **Interface**: Describes the structure of an object that must be implemented in a concrete class. An interface is a blueprint for a class. It is a contract with the classes that will either use it as a type definition or inherit it with the implements or extends keyword. Any property and method signatures must be implemented in its subclasses.
   1. **Class Example:**

interface Person {

name: string;

age: number;

greet(phrase: string): void;

}

let user1: Person;

* 1. **Interface Requirements:**
     1. **All properties are understood to be public**, so **private, protected access modifiers are not allowed**.
     2. **Interfaces and Types** Can use the readonly modifier.
     3. Interfaces **cannot be created using union types, although types can:**
        1. ~~interface dog: Playable & Greetable ;~~
        2. type Combinable = number | string;
  2. **Interfaces and Types serve two purposes**:
     1. They are used to create custom definitions for class and functions.
     2. They are used to type check objects that must have a certain structure.
  3. **Why use an interface over a type?** 
     1. Although Types and Interfaces can be used interchangeably, there are some differences.
        1. **Interfaces** can only be used to describe the structure of an object.
        2. **Types** can store the structure of an **object and union types**. Types are more flexible.
     2. **Abstract classes and Interfaces define the structure of a class:**
        1. A class can implement one or more **interfaces**, by separating them with a comma.
        2. An interface is a contract that an inherited subclass must adhere to. Subclasses can inherit more than one interface.
     3. **Difference between abstract classes and interfaces:**
        1. **Interfaces** have no implementation, only signatures. Any classes that implement an interface must declare any method signatures, to be a concrete class (class that can be instantiated).
        2. **Abstract classes** may be a mixture of signatures and implemented methods.
  4. **Why use interfaces?**
     1. **Interfaces** ensure the existence of common methods in inherited classes. Interfaces **enforce a certain structure in inherited classes**.
     2. **Readonly modifier** - Interface instance variables cannot be declared public or private, however, they can be made readonly. Readonly variables can be set only once and can’t be changed after the object has been initialized.
  5. **Interface Inheritance:**
     1. **Classes that Extend and Implement from Super classes and Interfaces, respectively:**
        1. When extending and inheriting from classes, you must use the extend keyword first:
           1. **Example:**

class Pet extends Feedable implements Playable{}

* + 1. **Extends** -
       1. Interfaces can extend other interfaces, using the **extends** keyword.
       2. Interfaces can inherit from multiple interfaces, using the extends keyword, but **classes can only inherit from one class (using extend keyword).**
       3. Interfaces can inherit (**using extend keyword**) one or more interfaces; however, **classes can implement multiple interface.**
    2. **Implements** -
       1. Multiple interfaces can be inherited by subclasses, using the **implements** keyword.
       2. The **class implementing the interface**, may contain more properties, but they must contain the properties defined in the interface.
  1. **Interfaces and type Function definitions**: Interfaces can be used to define a function type instead of type.

interface AddFn {

(a: number, b: number): number;

}

type AddFn = (param1: number, param2: number) => number;

* 1. **Optional Properties in interfaces, Classes, and Function parameters:**
     1. Use a ? when declaring a property or method optional, in the class or interface.

interface Named {

readonly name?: string;

outputName?: string;

myMethod?(param1: string, param2: string): string;

}

class Person implements Greetable {

name?: string;

age = 30;

constructor(n?: string) {

if(n) {

this.name = n;

}

}

}

* + 1. In an optional function parameter, the variable will be undefined, so there needs to be some type checks in the function, before using it.

1. **Useful Resources and Links:**
   1. **More on (JS) Classes:** [**https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes)
   2. **More on TS Interfaces:** [**https://www.typescriptlang.org/docs/handbook/interfaces.html**](https://www.typescriptlang.org/docs/handbook/interfaces.html)