TypeScript

•••

Objectives

- Review Resume/Profile Lab
- Set up Cloud9 for TypeScript Development
- What is TypeScript
- Variables - let & const
- Data Types
- Arithmetic Operators
- Comparison Operators
- Logical Operators
- TypeOf

Set Up Cloud9 for TypeScript Development

Step 1 - Create a new BLANK application in cloud9

Step 2 - npm init : we will hit enter all the way until we have to type yes

Step 3 - touch mycode.ts

Step 4 - Inside the package.json file we will change main to the name of or .ts file

Step 5 - Type some typescript code > save > and do tsc mycode.ts to run it

Step 6 - Create a tsconfig file by doing tsc --init in terminal

Step 7 - watch mode tsc mycode.ts --w

What is TypeScript?

TypeScript is a superset of JavaScript which primarily provides optional static typing Classes, and Interfaces. One of the big benefits is to enable IDE's to provide a richer environment for spotting common errors as you type the code. TypeScript compiles to JavaScript.

What is a compiler?
A compiler is a software program that transforms source code written by a developer in a high level programming language into low level object code(binary code) in machine language, which can be understood by the processor

Variables: let & Const

What is a variable?

A variable is a value that can change, depending on conditions or on information passed to the program. It’s like a variable like a box to hold something that can change!

There are two ways of declaring variables in TypeScript.

let - used to hold variables that can change.

const - used to hold variables that do not change

Example of using let to declare a variable

```
let myName: string = "Mikaila";

console.log(myName);
```

How we print something to the console?

Example of using const to declare a variable

```
const myAge: number = 19;

console.log(myAge);
```

If we try to assign another age to this variable the compiler will give us an error

We have several data types in TypeScript such as …

String: used to define text or letters

Number: used to defined variables that must be numbers or decimals

Boolean: used to hold data that will be True or False

Any: used to hold data types that can change to any type such as string or number

Array: used to hold a list of data with indexes starting at 0

Tuples: are like arrays but with mixed types and the order is important

Enums: used to make numbers more expressive

How to declare variables using types

```
let myString: string = “I am a string”;
let myNumber: number = 18;
let myArray: number[] = [1,2,4,5,7];
let myArray: string[] = [‘1’,‘2’,’3’,’4’];
let myFact: boolean = true;

Let canChange: any = 10;
canChange = ‘changed to a string’;
```

Arithmetic Operators

Addition   Plus sign (+)

Subtraction   Minus sign (-)

Multiplication Multiplication sign (*)

Division Division sign (/)

Modulus Remainder(%)

PostFix Increment and Decrement (x++)   and   x(--)

Prefix Increment and Decrement   (++x) & (--x)

Comparison Operators - Assuming X = 5

x > 10	= false	
x < 10	= true	
x >= 5	= true	
x <= 100	= true	
x == "5"	= true	// Type coercion
x === "5"	= false	// No type coercion
x != b	= true	
x !== "5"	= true	

Logical Operators

&&   AND both sides need to be true

|| OR One side needs to be true

!   NOT if something was true it makes it false (vice versa)

Using Operators for calculations

P. E. D. M. A. S rules applies -

The grouping operator ( ) controls the precedence of evaluation of expressions

```
let x : number = 500;
let y : number = 400;
let z : number = 10;
let a : number = 2;
let Answer =  x - y + z  * a ;   // How to fix  - Add brackets before multiplication?
Console.log(Answer);
```

Ternary Condition Operator

It takes three operands such as condition ? val1 : val2

```
let age: number = 20;
let status : number  = ( age >= 21)  ?  ‘can drink’  :  ‘cannot drink’;

console.log(status);
```

Conditional statements { if statements }

We have two main conditional statements in TypeScript.
The if else statement and a switch statement

IF STATEMENT

```
if (true) {
console.log(true);
} else{
console.log(false);
}
```

Using the OR || Comparison Operator

```
let n: string = 'netflix';
let  h: string = 'hulu';
let userInput: string = n;


if( (userInput == n) || (userInput == h) ){
        console.log('I will be streaming movies on netflix OR hulu this weekend' );
}else{
        console.log('I will be studying this weekend');
}
```

Using the AND && Comparison Operator

```
let n: string = 'netflix';
let  h: string = 'hulu';
let userInput1: string = 'netflix';
let userInput2: string = h;

if((userInput1 == n) && (userInput2 == h)){
        console.log(' I will be streaming movies on netflix and hulu this weekend ');
}else{
        console.log('I will be studying this weekend');

}
```

Else If

We use the Else If to test a new condition, if the first condition is false or not met

```
let condition1: number = 1;     let condition2: number = 2;
if(condition1 > condition2){
console.log('One is greater than Two');
}else if(condition1 == condition2){          // change to Not equal
console.log('One is equal to Two');
}else{
console.log('None of the above is true');
}
```

Switch statement

```
switch(value){
        case 0:
        console.log(print something......);
        break;

        case 1:
        console.log(print something else......);
        break;


        default:
        console.log(print something else......);
        break;
}
```

TypeOf

Since static typing is optional in TypeScript you may come across data and need to know the type of data it is. That’s when we use TypeOf as a mechanism to check

```
let myName = “Mikaila”;

console.log(typeof(myName));
```

will print - String

Practice with the other data types such as boolean, array, number etc...

While Loops

Used to loop through a block of code as long as a specified condition is true.

```
        while (condition) {
        Code block to be executed
        }

let j : number = 0;    let counter : number = 10;
        while( j < counter ) {
        j++;
        console.log(j);
}
```

DO/While Loop

A variant of the while loop. It will execute the code at least once before checking if the condition is true, then it will repeat the loop as long as the condition is true.

```
do{

 Code block to be executed

}
while ( condition);
```

See example on the page next page.

Example of Do while loop

```
let i : number = 0;

do {

    console.log( " The number is " + i );

    i++;

} while(i < 5);
```

For Loops

For loops are used to loop through a block of code a number of times.

```
for(statement 1; statement 2; statement 3){
        Code block to be executed
}
```

For example:

```
for( let x = 0 ;  x < 5 ;  x++ ) {
console.log( ' The value of x is ===> ' + x ) ;
}
```

Array Methods

let fruits : string[ ] = ["Banana", "Orange", "Apple", "Mango"];

toString() converts an array to a string of comma separated array values

console.log(fruits.toString());

join() behaves like toString but you can specify a separator

console.log(fruits.join(' * '));

pop() removes the last element from an array

console.log(fruits.pop ());

push() adds a new element to the end of the array and also returns the array length

console.log(fruits.push (‘ kiwi ‘));

Array Methods continued....

shift() removes the first element from an array and returns the element

```
console.log( fruits.shift () );
```

unshift() adds a new element at the beginning of the array

```
console.log( fruits.unshift ( ' lemon ' ) );
```

.length Used to get the length of the array

```
console.log(' The length of the array is  ' +  fruits.length);
```

For further array methods see w3schools for methods such as slice, splice, sort, reverse

Math Methods

Math.round (x)   returns the value of x rounded to the nearest integer   //4 . 7
Math.PI   returns 3.141592653589793
Math.pow(x, y);   returns the value of x to the power of y //8 , 2
Math.sqrt(x) returns the square root of x // 64
Math.abs(-x) returns the absolute positive value // -4 . 7
Math.ceil(x) returns the value of x rounded up to the nearest integer //4 . 4
Math.floor(x) returns the value of x rounded down to its nearest integer //4.7
Math.min(1,2,3)and Math.max(1,2,3) used to find the lowest and highest values
Math.random returns a random number between 0 and 1
For further array methods see w3schools

Functions

A function is a block of code designed to perform a particular task. Functions are executed when called or invoked. For Example: This function multiplies the values of p1 and p2 and returns the answer.

```
function myFunction(p1: number, p2: number) :number{
return   p1 * p2;
}
```

To call or invoke the function we simply do - myFunction(2, 20);

In cloud 9 we log it to the console - console.log(' Ans is ==> ' + myFunction(2,20));

p1 and p2 are known as parameters while 2 and 20 are known as arguments

The return type :number -- means that this function returns a number

Return keyword and function with no parameters

return (expression) ; - The return statement ends function executions and specifies a value to be returned to the caller of the function.
It is affected by the semicolon therefore always end your return statements with a semicolon **;**

```
 function myStringFunction(): string{
        return 'Hello world' ;
}

console.log(myStringFunction());
```

Notice that this function has no parameters and takes no arguments

Function with two parameters and a return type of Number

```
 function modulous(num1: number, num2: number): number{
        return num1 % num2;
}

 console.log(' The answer is  ' , modulous(91 , 2));
```

Notice that this function takes two parameters therefore it needs two arguments

Function with one argument Type and returns a String

```
 function myFavSport(nameOfSport: string): string {
        return ' My favorite sport is  ' + nameOfSport ;
}

console.log(myFavSport('Cheerleading'));
```

Notice that this function takes one argument of type string and returns a string

Function with arguments and no return type & return types

No return type and takes two arguments

```
function add(a: number, b: number):{
 console.log(a + b);
}
add(1, 2);
```

Takes one argument and has a return type

```
function returnMyName(name: string): string{
return 'My name is ' + name;
}
```

Mixing outside variables with parameters

```
let z: number = 100;

function addToZ(x: number, y: number):number {
 return x + y + z;
}

console.log(' The result of adding to z is ' + addToZ(3, 17));
```

Anonymous Function & Function as variables values

An anonymous function has no name. Often used as callback functions. For example You can assign a function with no name to a variable.

```
let xFunc = function (p1 :number, p2 :number): number {
return p1 * p2;
}

console.log(xFunc(20,6));
```

Number of parameters matter

```
function fullName(firstName: string, lastName: string) {
 return firstName + " " + lastName;
}

let result1 = fullName("Bob");                    // error, too few parameters - undefined

let result2 = fullName("Bob", "Adams", "Sr.");  // too many parameters

let result3 = fullName("Bob", "Adams");         // ah, just right


console.log(result1);
console.log(result2);
console.log(result3);
```

Functions without return keyword and outside variables

```
let tasty: boolean = true;

function printCalories(burger: string, amountOfCalories: number): void{
        console.log('Your ' + burger + ' has ' + (amountOfCalories * 2.5) + ' calories and its 
' + tasty + ' that it is tasty' );
}

printCalories('harmburger', 200);
```

Objects literal

An Object is a collection of named values. The named values are called properties.
Property is the key such as name
Value is the value of the key such as " Becky"

To declare an object using an object literal we simply do something like this -

```
let person: { name: string;  age: number;  sex:string;  likesHipHop: boolean; };
person = {
name: 'Becky',
age: 22,
sex: 'Female',
likesHipHop: true
};
```

Printing out the value of our Object

```
console.log("My name is " + person.name + " " + " I am " + person.age + " years old " +
person.sex + " and its " + person.likesHipHop + " That I like HipHop");
```

Try using the person Object to print out something unique. Make it your own!

Object Methods

Methods are actions that can be performed on objects. An object method is a function definition. Let’s add a method to our person object and print it out as seen below!

```
person = {
name: 'Becky',
age: 22,
sex: 'Female',
likesHipHop: true
makeNoise: function(){ return "Arhhhhhhhh"; }
};console.log("My name is " + person.name + " " + " I am " + person.age + " years old " + person.sex + " and its " + person.likesHipHop + " That I like HipHop so" + "everybody make some noise " + person.makeNoise());
```

For in - Used to loop through our object

```
for(let x in person){
console.log(x);  //prints the keys
}

for(let y in person){
console.log(person[y]); //print the values
}
```

For in continued....

To print both the keys and values of an Object we combine what we learned so far..

```
for(let z in person){
 console.log( z + " <==> " + person[z]); //print the values
}
```

To just get the value we leverage the key. For example

```
console.log(person["name"]); //get name
console.log(person.name); // get name
```

To set the value of the Object you use the key and assign

```
person.name = "yonce";
console.log(person.name);

console.log(person);
```

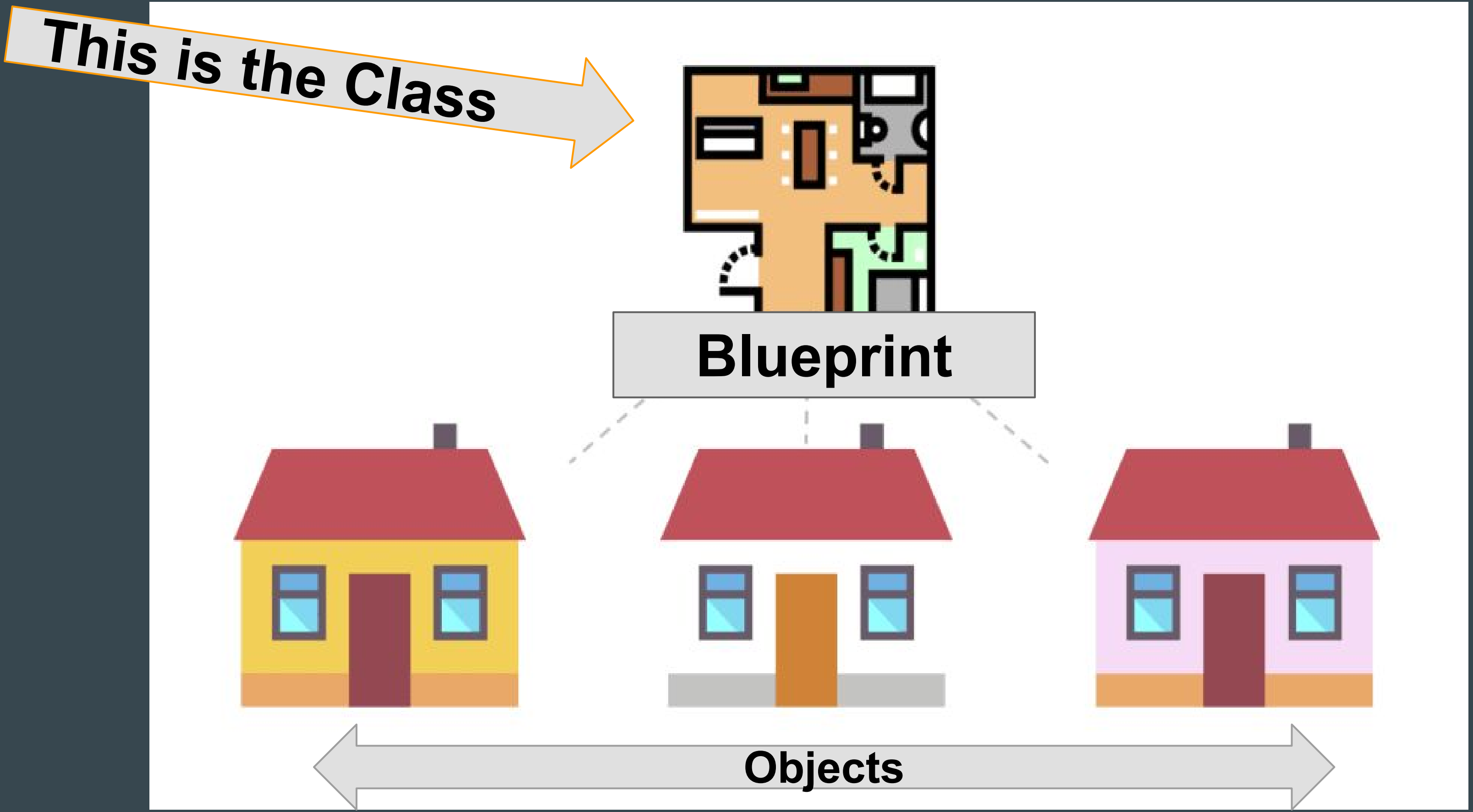
O.O.P Object Oriented Programming

Object Oriented Programming is a style of programming that focuses on using objects to design and build applications. An object is a self contained component that contains properties and methods needed to make a certain type of data useful.

An object's properties are what it knows(attributes) and its methods are what it can do (actions)
A class is a blueprint for creating objects. It is similar to the blueprint of a house.

Classes have a name, define attributes(properties or characteristics) and behavior(methods). An instance of a class is a specific built object for a specific class by a process called instantiation

Class - Blueprint to create objects!



Objects from a Car (Blueprint) class

Every car will have a model number , fuel tank, windows etc..
And will have behaviors like start, brake, accelerate ..etc

SO we need a Blueprint to create these basics things that every car needs then we can create different types of cars and change certain things like color

Class and Constructors

We start defining a class with the class keyword, followed by the class name. For eg:

```
class Person {
}
```

A constructor is a special method used by the new keyword to create instances of our class. (Objects). The syntax for a constructor is as follows:

```
 constructor( ){

    }
```

Access Modifiers

Access modifiers are used to change the visibility of properties and methods in a class.

public - accessible from outside the class. By default all is public

private - can only be accessed within the class or objects created by the class and exposed with getters and setters to objects of the class. Not accessible to subclasses

protected - are accessible by any objects that inherit from the class or created by the class. Inherited classes will have access to protected properties but not private properties

Class plus constructor Syntax

Then we add the class member’s to the class such as attributes and methods.
 Let’s start with the class’s Properties and then add a constructor.

```
class Person {
        public firstName: string;
        public lastName: string;
        public email: string;
        constructor(firstName: string, lastName: string, email: string){
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = email;
        }

}
```

This …what does this mean?

When inside the class and you want to refer to the properties and methods of an object we use the this keyword to do so

```
class Person{
        public firstName: string;
        public lastName: string;
        public email: string;
         constructor(firstName: string, lastName: string, email: string){
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = email;
        }
```

Methods are like functions without the function keyword

```
class Person{
        public firstName: string;
        public lastName: string;
        public email: string;
         constructor(firstName: string, lastName: string, email: string){
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = email;
        }
greet(){
console.log(' Hi ');  }
}
```

```
class Person{
        public firstName: string;
        public lastName: string;
        public email: string;
        private type: string;
        protected age: number = 24;
        constructor(firstName: string, lastName: string, email: string){
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = email; }
        greet(){ console.log ('Hello - ' + this.firstName);}
        printAge(){console.log(this.age);}
        setType(type: string){
        this.type = type;
        console.log(this.type); }
}
```

Instantiate an Object with our Class

```
let me: Person = new Person("Mikaila", "Akeredolu", "Mikaila@zipcode.com");
console.log(me);
console.log(me.firstName);
console.log(me.lastName);
console.log(me.email);
me.greet();
```

Next... when we try to access protected and private properties TypeScript throws error!!

```
console.log(me.type);  -- Property 'type' is private  via compiler see bash
console.log(me.age);  -- Property 'age' is protected via compiler see bash
```

Comment out methods inside class for now

Private Properties and Methods

They are not accessible outside the class. Can only be accessed inside the class

Let's set our setType Method to private and try to set it with our instance = warning

```
 private setType(type: string){
        this.type = type;
        console.log(this.type);     }
```

```
me.setType('private guy');
```
Then let's access it from the printAge() Method

```
// TyPeScript warns you - setType' is private and only accessible within class 'Person'
```

```
printAge(){
        console.log(this.age);
          this.setType('exposed guy');  //exposing
 }
```

Inheritance

Allow you to create subclasses(child class) from a parent class. The subclass inherits all the properties and methods of the parent class but child classes can include additional properties and methods not available in the parent class. To extend the parent class we use the extends keyword as seen below. Let’s add a method only available to Student class and not Person class and use the learn method

```
class Student extends Person{
        learn(){
         return 'I am a student learning how to code';' ;
        }
}
console.log(student)   console.log(student.learn());  console.log(student.firstName);
console.log(me.learn());  //  Property 'learn' does not exist on type 'Person'
```

Adding new Properties to a subclass and the Super keyword

Sometimes we need a subclass (child class) to provide a specific implementation of a method that it inherits from the Parent class. To do so we use the keyword super

Let's add a new property/attribute to the student class and initialize it with the constructor that we declared in the Parent class. Let's create a string array property

```
class Student extends Person{
        public subjects: string[ ];
constructor(firstName: string, lastName: string, email: string, subjects:string[ ] ){
        super(firstName, lastName, email);
        this.subjects = subjects; }
        learn(){ return 'I am here to learn';  }
}

//Create a student instance
let student = new Student('Mike','Jones','mj@gmail.com',['TypeScript', 'Java']);
```

Method Overriding

We can also use the super keyword when we want to extend an existing method. Let’s override the greet method that we inherited from the parent class as seen below

```
//Overriding the greet method inside the student class

    greet(){

        super.greet();  //get access to the parent class greet method

       return 'Hello. Did u hear that ' + this.firstName + ' is learning ' + this.subjects;

   }
```

Protected Access Modifier

protected - Earlier we discussed briefly that protected properties are accessible by any objects that inherit from the parent class or created by the class. Inside the student constructor let’s change the age of the student by adding modifying the constructor

```
  constructor(firstName: string, lastName: string, email: string, subjects:string[] ){
        super(firstName, lastName, email);
        this.subjects = subjects;
        this.age = 20;
}
```

Notice now that when we save and run, the student instance/ object’s age has changed.

```
Student {
age: 20, firstName: 'Mike', lastName: 'Jones', email: 'mj@gmail.com', subjects: [
'TypeScript', 'Java' ]
}
```

Getters and Setters

Allow you to offer controlled access to your properties. We create getters and setters like methods but call them/invoke them like properties in TypeScript.
Let’s say we want to create on species that have at least three or more characters. We can create a setter and check the number of characters as seen below

```
    set species(value: string){
        if(value.length > 3){
        this._species = value;
    }else{
        this._species = 'Default Specie';
    }

  }
```

Getter

To get access to the private property then we would use the getter syntax like this

```
get species(){
      return this._species;
  }
```

Implement a Getter and Setter for a new Class called Plant

```
class Plant{
        private _species: string;
        get species(){
                return this._species;
        }
        set species(value: string){
                if(value.length > 3){
                this._species = value;
                }else{
                this._species = 'Default Specie';
                }
        }

}
```

Instantiate a new Object and use the Getter and Setter

Let’s create a new object with less than three characters

```
let plant = new Plant();
plant.species = 'ox';
console.log(plant.species);   // Notice it logs the default value
```

Now if we add a specie with more than three characters we see the getter in action

```
plant.species = 'Green Plant';
console.log(plant.species);
```

Let’s create another class for practice and call it Menu

We will create a menu class to display lunch specials then extend the class.

```
class Menu{
    public items: string[ ];
    public pages: number;
        constructor(items: string[ ], pages: number){
            this.items = items;
            this.pages = pages;
        }
    lunchSpecials( ): void{
        console.log('Today our lunch specials are: ');
        for(let x = 0; x < this.items.length; x++){
            console.log(this.items[x]); }
    }
}
```

Now let’s create an instance of our Menu Object

```
const lunch:Menu = new Menu(['BBQ','Chicken','Fries', 'Burgers'], 1);

lunch.lunchSpecials();
```

It will print out the following list below in your console

```
BBQ

Chicken

Fries

Burgers
```

Now extend the Menu and create a subclass

```
class HappyMeal extends Menu{
        constructor(items: string[ ], pages: number){
                super(items, pages);
        }
                lunchSpecials( ):void{
                        console.log('The kids menu are: ');
                        for(let x = 0; x < this.items.length; x ++){
                                console.log(this.items[x]); }
                }
}

let happyMeal:Menu = new HappyMeal(['kids Meal', 'Ham meal', 'cheese burger meal'], 2);
happyMeal.lunchSpecials( );
```

Static Properties and Methods

Static properties and Methods belong to a class and do not need to be instantiated

For example Math.PI.. Math is the class and PI is a static property. Let’s create our own

```
class Helpers{
        static PI: number = 3.14;  // static variable

        static circumfrenceOfCircle(radius: number): number{  //static method
        return 2 * this.PI * radius;  // 2 * pi * r

    }

}
console.log(Helpers.PI);
console.log(Helpers.circumfrenceOfCircle(10));
```

Abstract Classes

Abstract classes are marked with the abstract keyword. They cannot be instantiated directly. You have to inherit from them. We do this to create sort of like a basic setup

```
abstract class Project{  //abstract means need to be extended
        projectName: string = 'Default Name';
        budget: number = 1000;
abstract changeName(name: string): void;  //abstarct method no logic implemetation
calcBudget(){
        return this.budget * 2;
        }
}
//Next we need to Implement the abstract class with the extend keyword
```

Implement our abstract class

```
class ITProject extends Project{
	changeName(name: string): void{
		this.projectName = name;
	}
}
let newProject = new ITProject();

console.log(newProject);

newProject.changeName('New IT Project');

console.log(newProject);
```

Singleton and private constructors

A class that you only want to have one instance of.. They have a private constructor. By making the constructor private. You cannot instantiate it from outside anymore.

```
class OnlyOne{
	private static instance: OnlyOne;
	private constructor(public name: string){}
	static getInstance(){
		if(!OnlyOne.instance){
			OnlyOne.instance = new OnlyOne('There is only one of me');
}
	return OnlyOne.instance;
	}
}
let rightway = OnlyOne.getInstance();
```

The Four Pillars of OBject Oriented Programming

Abstraction

Encapsulation

Inheritance

Polymorphism

Abstraction

Abstraction - Let's you focus on what an Object does versus how it does it. It helps you build Independent modules which can interact with each other by some means. Modifying one independent module does not affect another module

For example when you press the brake of your car. The brake system is abstracted while you are provided a pedal to press to stop your car or when you tap the buttons on your phone it does what you want like make a call or send a text without you needing to know how those things are implemented in the background/in the phone

Interface - outside looking in (Interfaces)...for an engineer use it to hide

Encapsulation

Encapsulation binds the data and methods together as a single unit. The data is not accessed directly but can be accessed through exposed functions. It exposes the solution to a problem without the user fully understanding the problem domain. It helps protects the data integrity and users cannot set the internal data directly to an invalid or inconsistent state.It’s about exposing complexity in a fail safe manner.

When something is defined inside a class and u dont need to understand how it works The implementation is encapsulated in a class..the process of defining an implementation of an interface

Inheritance

Inheritance helps organizing classes into an hierarchy. Then classes can inherit attributes(properties) and behaviors(methods) from their parent class. Inheritance describes IS A relationship for example- A parrot is a Bird, The US dollar is a currency

Inheritance allows you to declare a common implementation or behavior and later for specialized classes you can change them. Inheritance does not work backwards. A child cannot inherit from a parent. Inheritance is a mechanism for code reuse

Polymorphism

Polymorphism is the idea that there can be many different implementations of an executable unit and the differences occur all behind the scenes without the caller’s awareness. Inheritance with method overriding and overloading are types of polymorphism

S.O.L.I.D Principles of Object Oriented Programming

Single Responsibility Principle: States that a software component(function or class) should focus on one unique task (Have only one responsibility)

Open/Closed Principle: states that software entities should be designed with application growth(new code) in mind (Open to extension) but the application growth should require fewer possible number of changes to the existing code(closed for modification)

Liskov substitution principle: We should be able to replace a class in a program with another class as long as both classes implement the same interface and should work!

S.O.L.I.D Principles continued

Interface Segregation Principle: This states that we should split Interfaces that are very large (general purpose interfaces) into smaller and more specific ones(Many client-specific interfaces) so that clients will only need to know about the methods that are of interest to them.

Dependency Inversion Principle: This states that entities should depend on abstractions(interfaces) as opposed to depending on concretion(classes)