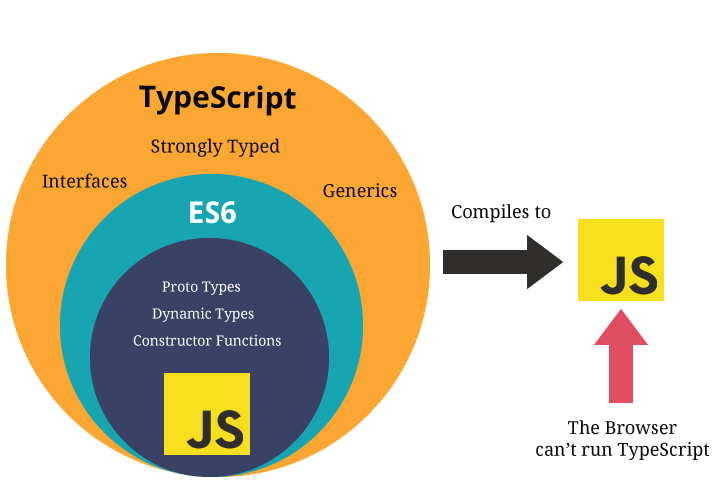
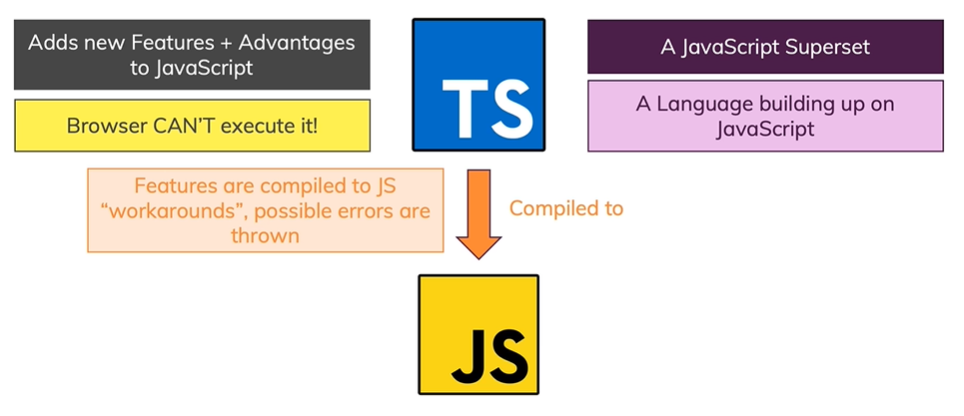
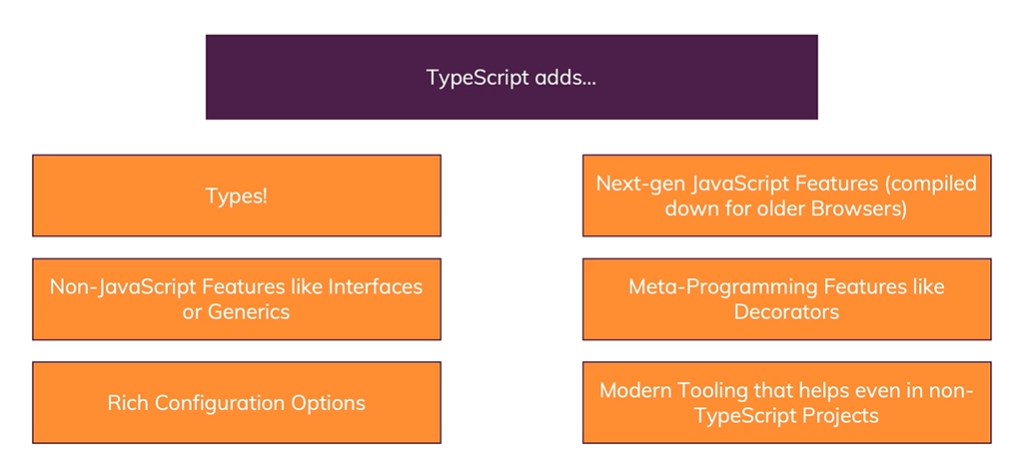
**TypeScript**

**What is TypeScript?**

****

****

* TypeScript is wrapper around the JavaScript.
* TypeScript is Strongly Typed Language which means you have to be specific with Variable type.
* TypeScript doesn’t run in the browser. We need to compile it to JS to run.
* TypeScript is a typed superset of JavaScript and pure Object oriented.

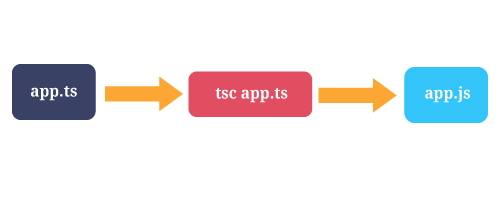


**Features of TypeScript:**

* Due to the static typing, code written in TypeScript is more predictable, and is generally easier to debug.
* Makes it easier to organize the code for very large and complicated apps.
* You can use TypeScript for other JS libraries, Because TS compiles plain JS.
* TS is platform Independent.

**TypeScript Compilation**

* The file extension of TypeScript is “**.ts**”.
* The TSC (TypeScript Compiler) is a source-to-source compiler (transcompiler / transpiler).
* The TSC generates the equivalent JS code from the source TS code. This process is termed as transpilation.



* ***tsc first.ts*** This will create a new file named first.js in the same location. Keep in mind that if you already had a file named **first.js**, it would be overwritten.
* You can run multiple files at the time by using ***tsc first.ts second.ts third.ts***
* Initializes a watcher process that will keep main.js up to date.  
  ***tsc main.ts --watch***

**To know the Node and NPM versions**

**node -v and npm -v**

**Execute the TS files directly from Node cmd**

1. Install ts-node npm globally by using this command

*npm install typescript -g*

*npm install ts-node -g*

1. Run the below command

*ts-node fileName.ts*

**How to Identify Installed Package version**

*npm info* ***packageName*** *version*

**Variable declaration in TypeScript**

We can declare a variable in TS by using the following ways

i. var variableName : dataType = value;

ii. var variableName : dataType;

iii. var variableName = value;

iv. var variableName; (you can store any values here)

**Difference between *var* and *let:***

| **let** | **var** |
| --- | --- |
| let allows you to declare variables that are limited in scope to the block, statement, or expression on which it is used. | This is unlike the let/keyword, which defines a variable globally, or locally to an entire function regardless of block scope. |
| Variables declared by let have their scope in the block for which they are defined, as well as in any contained sub-blocks. | In this way, let works very much like var. The main difference is that the scope of a var variable is the entire enclosing function |

**Example:**

| function varTest() {  var x = 1;  if (true) {  var x = 2; // same variable!  console.log(x);   }  console.log(x);  } |
| --- |

| function letTest() {  let x = 1;  if (true) {  let x = 2; // different variable  console.log(x);   }  console.log(x);  } |
| --- |

**String:**

let name : string = “gopi”;

You can change the variable value later but not datatype.

**Template String:**

Template strings are surrounded by the backtick/backquote (`) character, and embedded expressions are of the form *${ expr }*.

**Example:**

| let g:string = "gopi"; let age:number = 30; let msg = `Name is ${g}  age is ${age}`; console.log(msg); |
| --- |

**Result:**

Name is gopi

age is 30

**Number:**

var num : number = 20;

**Boolean:**

var boo : boolean = true;

**Array:**

var arr = [“string”, “string”]; => this is a string Array, you can store only string values not other data types.

var arr = [“string”, number]; => now the array only accept string and numbers only not other data types.

var arr : string[]; => this is a string Array, you can store only string values not other data types.

var arr : any[]; => now this array accept any kind of data types.

var xyz : string | number; // union method

var arr : (string|number|boolean)[];

**Object:**

*var obj: {name:string, phone:number} = {name:”gopi”, phone:123};*

In the above code you can only use 2 properties which is name and phone. You can’t dynamically add more properties.

If you want to add more properties dynamically means just use,

*var obj : {[k:string]:string} = {};*

Now you can add more properties.

**Enum:**

Enum allows us to define a set of named numeric **constant**. By default, enum is zero-based but we can change its value according to the requirement.

***enum*** *Animals {cat, lion, dog, cow, monkey};*

*let c: Animals = Animals.cat; // 0*

By default, the numbering of enums starts at 0, but you can also set a different value for the first or any other members manually. This will change the value of all the members following them by increasing their value by 1. You can also set all the values manually in an enum.

**any:**

The any type is a powerful way to work with existing JavaScript, allowing you to gradually opt-in and opt-out of type-checking during compilation.

*var x:any = “test”;*

**Custom Types:**

TypeScript allows us to create custom types.

*type Websites = 'www.google.com' | 'reddit.com';*

*let mySite: Websites = 'www.google.com' //pass*

*//or*

*let mySite: Websites = 'www.yahoo.com' //error*

// the above line will show error because Websites type will only accept 2 strings either 'www.google.com' or 'reddit.com'.

// another example.

*type Details = { id: number, name: string, age: number };*

*let student: Details = { id: 803, name: 'Max', age: 13 }; // pass*

//or

*let student: Details = { id: 803, name: 'Max', age: 13, address: 'Delhi' } // error*

// the above line will show error because 'address' property is not assignable for Details type variables.

//or

*let student: Details = { id: 803, name: 'Max', age: '13' }; // error*

// the above line will show error because string value can't be assign to the age value, only numbers.

**void:**

void is a little like the opposite of any: the absence of having any type at all. You may commonly see this as the return type of functions that do not return a value:

*function warnUser(): void {  
 alert("This is my warning message");  
}*

Declaring variables of type void is not useful because you can only assign undefined to them:

*let unusable: void = undefined;*

**Null and Undefined:**

*let u: undefined = undefined;  
let n: null = null;*

**Using Types in functions**

We can use the data types in functions also.

**Functions:**

Functions are the fundamental building block of any applications.

In typescript also we can write functions as named functions or anonymous function.

**Passing Parameter to the Functions:**

* In TypeScript, every parameter is assumed to be required by the function.
* When the function is called the compiler will check that the user has provided a value for each parameter.

*function fullName(fname:string, lname:string):string {*

*return fname+" "+lname;*

*}*

*let fn = fullName("gopi", "kris"); // accept*

*let fn = fullName("gopi"); // Error*

*let fn = fullName("gopi", "kris", “E”); // Error*

*console.log(fn);*

**Passing Optional Parameter:**

* We can use optional parameter when we do not pass any argument to the function by simply adding “?” in the parameter name.
* The optional parameter should be set as the last argument in a function.

*function* ***buildName****(firstName: string, lastName?: string) {  
 if (lastName)  
 return firstName + " " + lastName;  
 else  
 return firstName;  
}  
  
let result1 = fullName("Gopi"); // works correctly now  
let result2 = fullName("Gopi", "Kris", "Sr."); // error, too many parameters  
let result3 = fullName("Gopi", "Kris"); // ah, just right*

**Default-initialized parameters:**

* In TypeScript, we can also set a value that a parameter will be assigned if the user does not provide one, or if the user passes **undefined** in its place.
* Default-initialized parameters that come after all required parameters are treated as optional, and just like optional parameters.

*function fullName(firstName: string, lastName = "Kris") {  
 return firstName + " " + lastName;  
}  
  
let result1 = fullName("Gopi"); // works correctly now, returns "Gopi Kris"  
let result2 = fullName("Gopi", undefined); // still works, also returns "Gopi Kris"  
let result3 = fullName("Gopi", "Adams", "Sr."); // error, too many parameters  
let result4 = fullName("Gopi", "Arun"); // Gopi Arun*

**How to set default parameter before required Parameter?**

*function fullName(firstName = "Mr", lastName: string) {  
 return firstName + " " + lastName;  
}  
  
let result1 = fullName("Gopi"); // error, too few parameters  
let result2 = fullName("Gopi", "Kris", "Sr."); // error, too many parameters  
let result3 = fullName("Gopi", "Kris"); // okay and returns "Gopi Kris"  
let result4 = fullName(undefined, "Kris"); // okay and returns "Mr Kris"*

**Rest Parameter:**

* Required, optional, and default parameters all have one thing in common.
* Rest parameters don’t restrict the number of values that you can pass to a function.
* Sometimes, you want to work with multiple parameters as a group, or you may not know how many parameters a function will ultimately take.

*function dorest(...arg:(string|number)[]){*

*return arg.join(“ “);*

*}*

*console.log(dorest("hi", "hello", "how", "r", "u?", 44, 58, 96, 12));*

### **Arrow Functions**

Arrow function also called as lambda function, it avoids the reserved keyword “**function**”.

If your function has only one line means you can write the arrow function as,

*var doAdd = (x:number, y:number) => x+y;*

*console.log(doAdd(3, 4)); // 7*

Or

*var doAdd = (x:number, y:number) => {*

*return x+y;*

*}*

*console.log(doAdd(3, 4)); // 7*

Without parameter

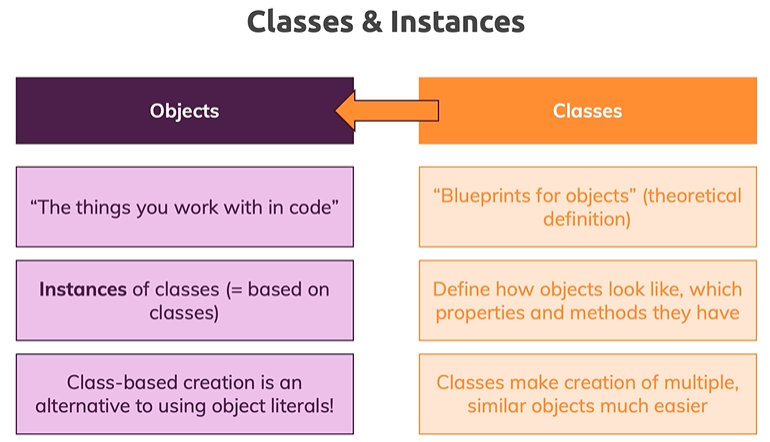
*var greet = () => {*

*console.log("Hello");}*

### **Class**

**What is Class?**

* A class is a template or blueprint that is used to create objects.
* Class representation of objects and the sets of operations that can be applied to such objects.
* A class consists of Data members and methods.
* When building large scale apps, the object oriented style of programming is preferred.



**Create a Class:**

*class class\_name {   
 //class definition*

*}*

The class Definition has the following,

1. **Propetry**

Property is any variable declared in a class.

Optional property ***name!:string;***

**2. Constructor**

Refers to a special type of function which will be called automatically whenever there is an object creation.

**3. Methods**

These are the function defined inside a class and are used to access object data.

**Example:**

*class person {*

*fname:string;*

*lname:string; // do not use var keyword for class properties*

*constructor(a:string, b:string){*

*this.fname = a; //* ***this*** *keyword represents the instance of current class*

*this.lname = b;*

*}*

*fullName():string {*

*return this.fname+" "+this.lname;*

*}*

*}*

*var obj = new person("gopi", "kris");*

*console.log(obj.fullName());*

*Note: If you want to access the constructor function parameter values, make it as public or private.*

# **Inheritance:**

Inheritance is a mechanism in which one class acquires the property of another class. Inheritance allows your Program is being able to extend existing classes to create new ones.

**Syntax:**

*class child\_class\_name* ***extends*** *parent\_class\_name*

**Inheritance can be classified as**

* **Single** − Every class can at the most extend from one parent class
* **Multiple** − A class can inherit from multiple classes. TypeScript doesn’t support multiple inheritance.
* **Multi-level** − The following example shows how multi-level inheritance works.

*class* ***Root*** *{   
 str:string;   
}   
  
class* ***Child*** *extends* ***Root*** *{}   
class* ***Leaf*** *extends* ***Child*** *{} //indirectly inherits from Root by virtue of inheritance   
  
var obj = new Leaf();   
obj.str ="hello"   
console.log(obj.str) // hello*

**Single Inheritance Example:**

*class* ***Wishes*** *{*

*dailyWish(msg:string)*

*{*

*console.log(msg);*

*}*

*}*

*class* ***festival*** *extends* ***Wishes*** *{*

*diwaliWish(msg:string)*

*{*

*console.log(msg);*

*}*

*}*

*var wish = new festival();*

*wish.dailyWish("Good Morning");*

*wish.diwaliWish("Happy Diwali");*

From the above code classes inherit properties and methods from base classes.

* Here, ***festival*** is a derived class that derives from the ***Wishes*** base class using the extends keyword.
* Derived classes are often called *subclasses*, and base classes are often called *superclasses*.
* Each derived class that contains a constructor function must call ***super()*** which will execute the constructor of the base class.
* The ***super*** keyword can be used in expressions to reference base class methods and the base class constructor.
* Before we ever access a property on this in a constructor body, we have to call ***super()***.
* The **super()** methods is not available in outside of the constructor.
* We can’t use **super** keyword for access the base class properties in child class, just use **this** keyword to access the property.

# **Public, private, and protected modifiers:**

1. **Public by default**

* We’ve been able to freely access the members.
* In TypeScript, each member is public by default.

1. **Private**

* When a member is marked ***private***, it cannot be accessed from outside of its containing class.

1. **Protected**

* The protected modifier acts much like the private modifier.
* The exception that members declared protected can also be accessed within deriving classes.
* A constructor may also be marked **protected**. This means that the class cannot be instantiated outside of its containing class, but can be extended.

# **Readonly modifier:**

* You can make properties ***readonly*** by using the readonly keyword.
* Readonly properties must be initialized at their declaration and can update in constructor only.
* You can’t use constants in class member functions, so just use readonly or static readonly.

*class read {*

*readonly a:string;*

*readonly b:number = 10;*

*constructor()*

*{*

*this.a = "hi";*

*this.b = 30;*

*}*

*}*

*var r = new read();*

*//r.a = "ddfd";*

*console.log(r);*

**Interface**

* Interface is basically contract signed by an object which is having guaranteed properties and methods.
* An interface is a **contract**: The person writing the interface says, "hey, I accept things looking that way", and the person using the interface says "OK, the class I write looks that way".
* There are only the signatures of the methods, which implies that the methods do not have a body.
* The interface can't do anything. It's just a pattern.
* Interface just have guaranteed properties and methods.

**Rewrite this code in ts**

function Car(name) {

this.name = name;

this.acceleration = 0;

this.honk = function() {

console.log("Toooooooooot!"); };

this.accelerate = function(speed) {

this.acceleration = this.acceleration + speed;

}

}

var car = new Car("BMW");

car.honk();

console.log(car.acceleration);

car.accelerate(10);

console.log(car.acceleration);