TYPESCRİPT

-Using type predicates -

-Using Class Types in Generics-

-classes>this

parameters,this

Types,this

-based type guards-

Javascript super set.Typescript writen on the javascript.It give us extra features like Type validation,easily writing code etc...

Installation :We must install Nodejs for npm(node package manager).

<https://nodejs.org/en/download/>

-npm install -g typescript (-g for install globally .after now on we don't need to install typescript for each project)

-Typescript doesn't run in browsers.We have to compile typescript files for transforming to vanilla javascript files.

tsc tscFileName.ts like this tsc index.ts this command compile typescript and give us a vanilla javascript file like index.js

Note:the meaning of the "Vanilla javascipt" is simple javascript.When we don't use javascript in React,vue,jquery etc..

**TypeScript advantages;**

**1-Static type-checking:**when we called a function with variable but if actually the function can't executed by the variable.

On vanilla javascript: const name="didem"; didem() ; we show err at runtime

on typescript : const name="didem"; didem(); we show err at devepment.

**2-Non-exception Failures:**

-when we called a key of object but if actually the key of object doesn't exist

vanilla javascript: return undefined

typescript: throw an error

-when we typed a wrong name of functions.

name.tolowercase()

vanilla javascript we show err at runtime

typescript: we show err at development

**3-Typescript tsc:** typescript compiler.

usage: tsc filename.ts

this command give us a vanilla javascript file in the same directory.This file won't have typscript custom codes like types...

Note:If we would rather typescript acted strict.use --noEmitOnError.

After now on if we have an error(or errors) ts file won't transform to js file.

usage:tsc fileName.ts

**-**Typescript's type system only help us during development.

**TypeScript types:**

**: string : number**  : **boolean**

declaring a type:

let age: number , name: string , isAdmin: boolean

(age: number,name: string,isAdmin: boolean)=>....

example:

const num1=document.getElementById("num1")! as HTMLInputElement;

const num2=document.getElementById("num2")! as HTMLInputElement;

function addNumbers(num1:number,num2:number){ alert(Number(num1)+Number(num2))}

document.getElementById("addButton").addEventListener("click",()=>{

addNumbers(num1.value,num2.value)

})

expression:

-const num1=document.getElementById("num1")! as HTMLInputElement; get element with id "numb1" as HTMLInputElement.

-This code throw an error.The error is "Argument of type 'string' is not assignable to parameter of type 'number' ".Function addNumbers wants number values but type of num1.value and num2.value are string.

For this error we have to type addNumbers(+num1.value,+num2.value)

inference:Typescript help us to catching error on development time like this example.

**: any** : variable can be any type.

let name=21;name="didem";typescript throw a error

let name: any=21;name='didem';typescripy doesn't throw a error.

: any[] : Elements can be any type.let x: any[], x=["hey",2,true]

-If we use : any ,typescript can't control this variable.So use

**--noImplicitAny** on cli to if we haven't typed a type at any variable or parameter ,typescript will throw an error.

**: void:**Only undefined type assignable to void type.

if we don't use **--strictNullChecks** on cli ,null type assignable to void type.

-When we use :void type for function,the function can not return any statement.like return true ,return "sad".but can just return or return undefined

**: object:**Value can be object or array.Can not be primate or null or undefined.

**:unkown**:Similar to any but a little bit safely from any.It not legal to do anything with :unkown type.

const x=(a:any)=>{

a.toLowerCase() // ok

}

const x=(a:unknown)=>{

a.toLowerCase() //err

}

**: never** : Some functions never return a value.like err functions..

Never provide that Function can't return anything.

const x=(): void=>{

return // ok

}

const x=(): never=>{

return //err

}

-contuined on after control flow analysis

**: function :**When we use ":Function" type ,function return type become ": any".If we use "()=>void" function return type become ":void"

**Union types:** string | number or string | boolean ....

let name: string | number;

name="ahmet" ; name=21 both expression is right.

We can change type of variable through union types.

-When we use union types in functions, we must use narrow or we will get an error.

function write(a: string | number){

a.toLowerCase(); // ts will throw an error.

}

write("ahmet")

we must use narrow.

-function write(a: string | number){

if(type of a==="string"){a.toLowerCase()}} that method is write.

-function write(a: string | string[]){

if(Array.isArray(a))console.log(a.length)}

**Javascript typeof:**typeof method give us basic imformation about variable.

typeof "23" typeof 2 typeof {a:"d"} typeof [1,2] typeof true typeof fn

typeof can return these statements: "string" "number" "function" "object" "symbol" "bigint" "boolean" "undefined"

-typeof null and array variables are essentially object.This is one of those unfortunate accident of history.

-typeof returns string statements like"string" not String

-0 NaN null undefined "" is equal to false for conditions.if...

- !!"mehmet" : value:true type:true but Boolean("mehmet") type:boolean value:true

- these are looser equalities : == , != .

x==null equal to x==undefined equal to x==undefined ||x==null

**-in Operator:** if an object has a property with this name."x" in user

**Control Flow Analysis**:Typescript uses this strategy to split and combine types.

let username: string | number | boolean ;

username=21; // number

username=Math.random() > 0.3 ? "hakan" : 2; // string | number

username=Math.random()>23; // boolean

**: never** : "never" type can assignable every type but no type can assignable to "never" type( except never itself).So we can use : never type to be sure all types are used.

const getUser=(id: string | number | boolean)=>{

if(typeof id === "string")console.log("string")

else if(typeof id==="number")console.log("number")

else {let \_allowTypeUsed: never=id} //throw an error.

}

const getUser=(id: string | number | boolean)=>{

if(typeof id === "string")console.log("stribg")

else if(typeof id==="number")console.log("number")

else if(typeof id==="boolean") console.log("hi")

else {let \_allowTypeUsed: never=id} //all types used.we can delete.

}

**Object Types:**

const user={name:"ahmet",surname:21} is equal to

const user: {

name:string,

surname:number

}={

name:"ahmet",

surname:21

} both declarations are right and equal.

**Array types:**

: any[] :elements can be any type.

: string[] : elements can just be string. equivalent to ["ahmet"]

: number[]: elements can be just number .equivalent to [1,2,3,5,21]

:boolean[] , : object[]

**type typeName=type or union types** ;This operator allows us to type the type of expressions before expressions.type myname=string

type user={

name:string,

age:number

}

const user: user={

name:"ahmet",

age:21

}

type username= "username" | number

type surname= string

**interface** : like a type but a little bit different from it.

interface User{

username:string,

age?: number

}

const getUser=(user:User)=>{}

getUser({username:'1""'})

**interface vs type:**

-extending: Both methods can extending.

extending a type:

type Customer={

username:string,

}

type Customer2=Customer & {

surname:string

}

extending a interface:

interface User{

username:string,

}

interface User2 extends User{

surname:string

}

-Adding a new file : we can't add new file to type alias.

interface User{

username:string,

}

interface User{

surname:number

}

-combine interfaces:

interface username{

username:string

}

interface id{

id:number

}

interface User extends username,id{}

const user:User={

username:"asd",id:2

}

or with type

type ColorfulCircle = username & id;

**Tuple Types:** this type say how many elements in the array and which elements have which types.

let users: [string,number,boolean] or let users=["ahmet",2,"derya"]

These operations will throw error.

user[1]="ahmet" ; user[2]=21 ; user[5]="whatever"

users=["ahmet"] ; users=["ali",21,true,"ali"]

Important annotation:When we declared a array with typescript without initial variable and after we push an element.Typescript doesn't throw an error because everything is ok for it.But js will throw an error.

Example: let users : string[]; users.push("ahmet");

for typescript is ok but js will throw this err.

TypeError: Cannot read properties of undefined (reading 'push')

**İmportant onnotation 2:** tuple types can not catching error for push method.

-We can use optional tuple type but It have to be last element.

const users=(users:[string,number,string?])=>{}

**Enum Types**:Enums generates unique values for the property.

enum Ware { BOOK,TABLE,PENCİL,CAR} //book=0,table=1,pencil....

const user={

name:"ahmet",

ware:Ware.PENCİL

}

switch(user.ware){

case Ware.BOOK: console.log("this is a book") ; break;

case Ware.CAR : console.log("this is a car"); break;

case Ware.PENCİL : console.log("this is a pencil"); break;

default: console.log("this is a car")

}

-We can use custom numbers;

**Logic:**Each member value of enum +=previous member value of enum

enum Wares{ CAR=32,TABLE,BOOK} CAR=32,TABLE=33...

- enums without initializers either need to be first, or have to come after numeric enums initialized with numeric constants or other constant enum members.

-\*Enums are read-only.We can't change enums

**String enums**: enum Wares {PENCIL="PENCIL",CAR="CAR"}

enum Wares {PENCİL="PENCIL",CAR=2}

**-keyof typeof :**

enum Ware { BOOK,TABLE,PENCİL,CAR}

const addss=(username: keyof typeof Ware)=>console.log(username)

equivalent to const addss=(username: "BOOK" | "TABLE" | "PENCIL" | "CAR")=>console.log(username)

**Reverse mapping:**enum Ware { BOOK,TABLE,PENCİL,CAR}

const a=Ware.BOOK

console.log(a) //0

console.log(Ware[0]) //BOOK

console.log(Ware[a]) //BOOK

-String enum members can't reverse mapping.

**Literal Types:** We can use certain values for a variable.

let username: "ahmet" | "mehmet"; just can be "ahmet " or "mehmet".

fonction change(color: "red" | "blue" | "yellow"){}

function change(color: "red"){}

let combine: string | 1 | 2; can be 1 or 2 or any string.

const getUser=(username:"mehmet"): 'yes'| 'no'=>{

return 'yes'

}

**Const vs let-var variables:**

When we declare a variable with let or var.We can change variable whenever we want.

let user="ahmet" ; user="kada";

but when we use const variable .we can't change value of variable because it is a constant variable.

const username="mehmet" equivalent to let username: "mehmet"

**-If** we want to keys value be literal type values.we must use as const

const user={username:"mehmet",surname:"ilhan"} as const;

user.username="asd" ;ts throw an error.

**-example:**

const getUser=(username: 'ahmet' | 'mehmet')=>{} this function want to 'ahmet' or 'mehmet' to work.

const user={username:"mehmet",age:21};

username key is equal to username: string="mehmet";

-getUser(user.username) in this condition, we send user parameter type as 'string' but parameter type want to 'ahmet' or 'mehmet' from us.So ts throw an error.

-solutions:

1-const user={username:"mehmet" as "mehmet",age:21};

2-const user={username:"mehmet",age:21} as const;

3-const user:{username:"mehmet", age:number}={

username:"mehmet", age:2}

**Functions:** we can would prefer function returned a specific type.For this use types for functions.

types: string number boolean any object never void unkown ...

const combine=(): number=>{}

const getUsername=(id: number): string=>{}

Annotation:For example,we use forEach() method for an array.

const users=["ahmet","mehemt"]; users.forEach(item=>...)

typescript will inspect each item of array and tell us what we methods we can use.

const getUser=( py:{username: string,id:number})=>{}

type Py={username: string,id:number};

const getUser=(pt:Py)=>{}

**-x instanceof Foo** checks whether the prototype chain of x contains Foo.prototype

**Optional operator ?** :this operator say this parameter not necessary.

function hey(op:{name:string,surname:number}){}; hey({name:"ahmet"})

typescript throw an error.

function hey(op:{name:string,surname?:number}){] hey({name:"ahmet")

everything is good.

obj.last?.toUpperCase() :if obj.last is exist use toUpperCase() Moden js

-When we use a optional parameter,callers can always pass "undefined".

**Type Assersions:**We can tell typescript that this element is ...

const div=<HTMLInputElement>document.getElementById("asd")

or

const div=document.getElementById("asd") as HTMLInputElement

const user={username:"mehmet"] as const is equivalent to

const user: {username:"mehmet"}={username:"mehmet"}

**Null and Undefined:**Typescript supports these types and provide an option for checking.

strictNullChecks at cli --strictNullChecks

default this command's value is off.When we changed value to on.typescript checks them for errors.like:

const getUser=(username: string | null)=>{

username.toUpperCase() } //throw an error because can be null

const getUser(username: string |null){

if(username!==null)username.toUpperCase()} //ok.

-Typescript special operator for null and undefined: !

username!.toUpperCase() equal to if(username!==null || username!==undefined)

**Function type Expressions:**

(x:string)=>void :one function with parameter x .x must be string.The function can't return any value withoud undefined.

const isIncludeX=(x:string): boolean=>x.indexOf("x")>-1

const controlles=(x:(c:string)=>void)=>console.log(x("hey"))

controlles(isIncludeX)

-also ,We can declare fn type in type;

type Fn={

(a: string):void

}

const x=(a:Fn)=>{

a("hakanım")

}

const s=(a:string)=>console.log(a)

x(s)

**Generic Types:**

-Sometimes we want to function parameter type be equal to function return data type.Like above example,function can return any type.Use **<Type>** for certain type.

const getUser=<Type>(users: Type[]): Type | undefined=>{

return users[0]}

-there are two ways to call Generic type:

getUser(['ahmet','mehmet']) // must return same Type or undefined.not number,string or boolean.Must return same Type or undefined.Like this example getUser(['ahmet','mehmet']) string method become active.

getUser([1,2]) number method become active.

example2: const combine=<Type>(n1:Type,n2:Type)=>{

console.log(typeof n1===typeof n2) //always be true

}

-<Type> value is parameter.We can change parameter name.<Type1> or <Hey>. We can use multiple Type like <Num1,Num2>

-***why we use "Type" instead of "any"***

***-the write types of generic types:***

***1-function doit<Type>(name:Type):Type{}***

***2-const doit=<Type>(name:Type):Type=>{}***

***3-interface Doit{***

***<Type>(name:Type):Type***

***}***

***const doit:Doit=<Type>(name:Type):Type=>name***

***4-interface Doit<Type>{***

***(name:Type):Type***

***}***

***const doit:Doit<number>=<Type>(name:Type):Type=>name,***

***5-Generic Type with class:Similar to interface.***

***class GenericNumber<NumType> {***

***zeroValue: NumType;***

***add: (x: NumType, y: NumType) => NumType;***

***}***

***let myGenericNumber = new GenericNumber<number>();***

***myGenericNumber.zeroValue = 0;***

***myGenericNumber.add = function (x, y) {***

***return x + y;***

***};***

**Constraint:**If you prefer to contrait the types, use this method.

<Type extends {property:return Type}>

example:const smallest=<Type extends {length:number}>(num1:Type,num2:Type): any=>void

smallest(["sd"],[21,21]) // ok

smallest("asd","asdd") // ok

smallest(1,2) // // Error! Numbers don't have a 'length' property

<Type extends {length:number}> is equal to

interface LengthWise{length:number}

<Type extends LengthWise>

-Advanced example:We can pass the length value to execute.

const arrLength=<Type extends {length:number}>(arr:Type): number=>arr.length

console.log(arrLength("asd"))

console.log(arrLength([21,23]))

console.log(arrLength({value:21,length:23}))

**Keyof with generic constraints:**

const isKeyInObj=<Obj,Key extends keyof Obj>(obj:Obj,key:Key)=>obj[key]

const x={a:2,b:2,c:21,s:21,ahme:"dasd"}

isKeyInObj(x,"a")

isKeyInObj(x,as) //can not find name "as"

**Specifying Type arguments:**function combine<Type>(arr1: Type[], arr2: Type[]): Type[] {return arr1.concat(arr2);}

combine([1,2],["12",2]) //Type 'string' is not assignable to type 'number'.

combine<string | number>([1,2],["12",2]) // ok.

-type parameters should appear twice.No need use type parameter to one call.

**Function overloads:**We can type multiple functions.After we must combine them.Combine function must be compatible for all overload signatures.

function makeDate(timestamp: number): Date;

function makeDate(m: number, d: number, y: number): Date;

function makeDate(mOrTimestamp: number, d?: number, y?: number): Date {

if (d !== undefined && y !== undefined) {

return new Date(y, mOrTimestamp, d);

} else {

return new Date(mOrTimestamp);

}

}

makeDate(321312) // ok

makeDate(1,2,3) // ok

makeDate(1,2) // err

-If all overload signatures return the same type,we don't need write type on implementation signature(last function).

function x(a:string): number;

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

function x(a:number | string,b?:number){return 2} //must return number

console.log(x(1,2))

console.log(x("2"))

console.log(x(1))

-We can call single overload for each call.

function x(a:string): number;

function x(a:number): number;

function x(a:number | string){return 2} //must return number

console.log(x("2"))

console.log(x(1))

console.log(x(Math.random()> 0.5 ? 1 : "2"))

--Always prefer parameters with union types instead of overloads as possible as

**Declaring this in a function** .

const user = {

id: 123,

admin: false,

becomeAdmin: function () {

this.admin = true;

},

};

-typescript let you declare the type for this in the function body;

interface User {

id: number;

admin: boolean;

}

const admins =function (this: User) {

return this.admin;

};

**Parameters and Arguments:**

**Rest Parameters:**Allows infinite parameters to be used in a function(...)

const x=(...a:number[])=>a.forEach((item)=>console.log(item))

x(1,2,3,1,312,3,23,12,321,312,32,13,21)

-use string[] or use number[] or use any[]..

**Rest arguments:**

const arr1 = [1, 2, 3];

const arr2 = [4, 5, 6];

arr1.push(...arr2);//ok

---

const args = [8, 5];

const angle = Math.atan2(...args); // err

---

const args = [8, 5] as const;

const angle = Math.atan2(...args); // ok

**Parameter Destructing:**

type ABC = { a: number; b: number; c: number };

function sum({ a, b, c }: ABC) {

console.log(a + b + c);

}

or

function sum({ a, b, c }: { a: number; b: number; c: number }) {

console.log(a + b + c);

}

**readonly property:**This feature is invalid for runtime, active only for development time. Thanks to this feature, the key cannot be changed later.

type User={

readonly username:string

}

const getUser=(user:User)=>{

user.username=21 //err

}

-readonly property can remove with alias.

type X={name:string}

type Y={readonly name:string}

const x:Y={name:"ahmet"}

// x.name="sasd" //err

const y:X=x;

y.name="asd" //ok

**Index signature:{[index\_name: index\_type]:element\_type}**

type testType = {

[<index\_name>: <index\_type>]: <element\_type>

}

// index\_name: could be any string value.

// index\_type: string or number.

// element\_type: it could be any premitive type (string, number etc) or a custom type you want your elements to conform to.

type User={[username: string]:number} //all values must be string. // obj.key can use

type User2={[index: number]: string | number} //all values must be string or number. // obj.key can't used

**Generic Object Types:**

interface User{

username:any // not good idea

}

interface Box<Type> {

contents: Type;

}

type Box<Type> = {

contents: Type;

};

example:

interface User<Type>{

username:Type

}

let user:User<string>={username:"mehmet"}

or with Type alias:

type Box<Type> = {

contents: Type;

};

type alias with primate types:

type User2<Type>=Type | number;

let mUser:User2<string>="asd";

**The Array Type:**Array<Type> === Type[]

const user:Array<string>=["asd","sad"] is equal to const user:string[]=["asd","sad"],

**ReadonlyArray Type** :It ensures that the Array elements do not change.

ReadonlyArray<Type> === readonly Type[]

const users: ReadonlyArray<string>=["ahmet","mehmet"]

const users2: readonly string[]=["ahmet","mehmet"]

users[2]=1

users2[2]=1

**Rest parameter on Arrays:**

[string,...number[],string] [...number[],string] ....

**Readonly Tuple Types:** readonly [string,string]

const x=[1,2] as const; = const xy: readonly [number,number]=[1,2]

**Creating Types from types:**This section will teach to create new type from existing type or value.

**"keyof" Type Operator:**take object key types.

type Object={username:string,age:number};

type P=keyof Object; is eqaul p="username" | "age"

example 2:

type Obj={ [username: number] : string}

type key=keyof Obj;

const getItem=(obj:Obj,key:key)=>{

obj.key // error

console.log(obj[key])

}

example 3:

type Mapish = { [k: string]: boolean };

type M = keyof Mapish; equal to type M = string | number

**Typeof Operator:**Already we use "typeof" for basic types .In typescript we can also use for function return types.

type Predicate = (x: unknown) => boolean;

type K = ReturnType<Predicate>; // K=boolean

function f() {return { x: 10, y: 3 };}

type P = ReturnType<typeof f>; // P== {x: number;y: number;}

**Indexed Access types:** We use "indexed access types" to take key type from another type

type User={

username:string,

surnamge:string,

age:number,

isAdmin:boolean

}

type Username=User["username"] //string

type UserInfos=User["username" | "age"] //string number

type X=User[keyof User] // string | number | boolean

-Indexed access type is a type.So we can use union type or keyof with it.

type S=User["username"] | boolean // string | boolean

-we can use it on arrays.Use keyof for capture to element types.

const Users=[

{username:1,age:23},

{username:21,age:12}

]

type User=typeof Users[number] // User={username:number,age:number}

type Age=typeof Users[number]["age"] // Age=number

**Conditional Types:**condition ? trueType : falseType

type IdObj={

id:number,

key:string

}

type UserObj={

username:string,

surname:string

}

type FnReturnType<Type extends number | string>=Type extends number ? IdObj : UserObj;

const getUser=<Type extends number |string>(peyload:Type): FnReturnType<Type>=>{

throw "asd"

}

let idObj=getUser(21) // IdObj

let userObj=getUser("asd") // UserObj

let whetherObj=getUser(Math.random()>0.5 ? 2 : "asd") //UserObj | IdObj

**Mapped Types:**Mapped types use to generate a type from anyone type.

type User={

username:string,

age:number,

isAdmin:boolean

}

type UserBooleans<T>={ // change each value type to boolean.

[index in keyof T]:boolean

}

const user:UserBooleans<User>={

username:true, // username:"ahmet" is false

isAdmin:false,

age:false,

}

-We can remove readonly and ?( optional operator) with mapped modifier.Use - to remove and + to assume.

type User={

readonly username:string,

readonly age:number,

isAdmin:boolean

}

type RemoveReadonly<T>={

-readonly [index in keyof T]:T[index]

}

let x:RemoveReadonly<User>={

username:"asd",

age:21,

isAdmin:false

}

x.age=12; // ok.

-[Property in keyof Type]-?: Type[Property]; //to remove optional operator:

-We change key name with mapped types.

type addMeToEachKey<T>={

[index in keyof T as `Hi${string & index}`]:T[index]

}

type User={

username:string,

age:number

}

let user:addMeToEachKey<User>={

Hiusername:"asd",

Hiage:21

}

-We can filter key:

type RemoveKindField<Type> = {

[Property in keyof Type as Exclude<Property, "kind">]: Type[Property]

}; // {

radius: number;

}

-advanced:with conditional types.

type ExtractPII<Type> = {

[Property in keyof Type]: Type[Property] extends { pii: true } ? true : false;

};

**Template Literal Types**:"Template literal types" build on "string literal types" and use to expand string types.

type Users="Ahmet" | "Mehmet";

type Admins="Ali" | "Alex"

type AllMembers=Users | Admins // "Ahmet" | "Mehmet" "Ali" | "Alex"

type OneAdminOneUser=`${Admins} | ${Users}` // "Ali | Ahmet" | "Ali | Mehmet" | "Alex | Ahmet" | "Alex | Mehmet"

**-To string manipulation:**

-Uppercase:convert string to uppercase version

-Lowercase:convert string to lowercase version

-Capitalize:convert first letter of string to uppercase version

-Uncapitalize:convert first letter of string to lowercase version

way 1:on expressions.

type UppType<T extends string>=Uppercase<T>

type lowType<T extends string>=Lowercase<T>

type CapitalType<T extends string>=Capitalize<T>

type UnCapitalType<T extends string>=Uncapitalize<T>

let user1:UppType<"hakanim">="HAKANIM"

let user2:lowType<"Hakanim">="hakanim"

let user3:CapitalType<"hakanim">="Hakanim"

let user4:UnCapitalType<"Hakanim">="hakanim"

way 2:on types.

type Users="Ahmet" | "Mehmet";

type UppUsers=Uppercase<Users> // "AHMET" | "MEHMET"

type LowUsers=Lowercase<Users> // "ahmet" | "mehmet"

type capitalUsers=Capitalize<Users> //"Ahmet" | "Mehmet"

type UnCapitalUsers=Uncapitalize<Users> // "Ahmet" | "Mehmet"

**Classes**

-Classes serve as template to create new objects.

-Classes are actually functions.We can use class properties class without "class" syntax.

-Classes use to create inherit and reproducate complex objects .

-Fundamental purpose is useing ES6 properties.

**x in Object:**It looks all prototypes of object and take all keys.

**x.hasOwnProperty("prop"):** It looks just own object.It doesn't look the prototypes of object.

**for(const key in object) :**It takes all keys.It looks all prototype of object.

**Object.keys(object):**It just takes own keys of object.It doesn't look prototypes of object.

-constructor function is copied in memory and when want to generate a new object from it ,the constructor function in memory generate new object.

-function User(){} ; const user1=new User()

user1.\_\_proto\_\_ = User.prototype

-there are prototypes that contain all properties of that type under each type

-It can delete or add new properties to affect all members created by a particular prototype.

delete Array.prototype.forEach

const users=[1,23,23,123]

users.forEach(item=>console.log(item)) //TypeError: users.forEach is not a function

**APPLY,CALL AND BIND**

Apply:This method calls a function with a given "this" value,parameters type as array. fn.apply(objectName,[params])

Call:this method calls a function with a given "this" value,parameters type as parameter :). fn.call(object,param1,param2....)

Bind:It generate new function with called function and a given value

const x={

y:"ahmet",a:21,c:function(){

console.log(this.y,this.a)

}

}

x.c() // ahmet 21

const b=x.c

b() //undefined undefined

const z=x.c.bind(x)

z() // ahmet 21

**class-Constructor Function:**

function User(username,surname){

this.username=username,

this.surname=surname,

this.writeConsol=()=>console.log("hi")

}

User.prototype.writeUser=function(){

console.log(this.username,this.surname)

}

-Differences:

1-"Functions" don't hide properties in prototype

for(let item in new User()){

console.log(item)

}

username

surname

writeConsol

writeUser

-this function is not eqaual to class syntax.Differences will be explained

class User{

constructor(username,surname){

this.username=username,

this.surname=surname,

this.writeConsol=()=>console.log("hi")

}

writeUser(){

console.log(this.username,this.surname)

}

}

-Differences:

1-"Classes" hide methods in prototype

for(let item in new User()){

console.log(item)

}

username

surname

writeConsol

2-Classes always use strict

annotation:

console.log(typeof User==="function") //true

console.log(User===new User.\_\_proto\_\_.constructor) //true

**-Class syntax:**

class className{

constructor(params){

this.prop1=param1 ...

}

method1(){} // this field will type object prototype.

}

**-Getters and setters:** they are written to class Name.prototype

class User{

constructor(username){

this.username=username

}

get username(){ return this.\_username}

set username(val){

return val.length<3 ? console.error("invalid input") : this.\_username=val

}}

**Computed Names:**

class User{constructor(){}

[`Noluyoqnq${'x'}`](){}}

**Fields:** class User{

username="mehmet"

}

**Inheritance: class childClass extends ParentClass**

class Animal{

constructor(name){

this.speed=0

this.name=name

}

run(speed){

this.speed=speed

console.log(`${this.name} is runnig with speed ${this.speed} h/s`)

}

stop(){

this.speed=0;

console.log(`${this.name} stopped,ok by why ?`)

}

}

class Rabbit extends Animal{

hide(){

console.log(this.name+"hided")

}

}

const animal=new Animal("rabbit")

const rabbit=new Rabbit("rabbit")