Q

When should you use GET and when POST?

idempotent methods

https://developer.mozilla.org/en-US/docs/Glossary/idempotent

An HTTP method is **idempotent** if an identical request can be made once or several times in a row with the same effect while leaving the server in the same state.

An idempotent method should not have any side-effects (except for keeping statistics).

Implemented correctly, the GET, HEAD, PUT, and DELETE method are **idempotent**, but not the POST method.

safe methods

https://developer.mozilla.org/en-US/docs/Glossary/safe

An HTTP method is safe if it doesn't alter the state of the server.

A method is safe if it leads to a read-only operation.

Several common HTTP methods are safe: GET, HEAD, or OPTIONS. All safe methods are also idempotent, but not all idempotent methods are safe.

For example, PUT and DELETE are both idempotent but unsafe.

Safe, unsafe and idempotent methods

- GET /pageX HTTP/1.1 is idempotent. Called several times in a row, the client gets the same results:
  - GET /pageX HTTP/1.1
  - GET /pageX HTTP/1.1
  - GET /pageX HTTP/1.1

- POST /add_row HTTP/1.1 is not idempotent; if it is called several times, it adds several rows:
  - POST /add_row HTTP/1.1
  - POST /add_row HTTP/1.1 -> Adds a 2nd row
  - POST /add_row HTTP/1.1 -> Adds a 3rd row

- DELETE /idX/delete HTTP/1.1 is idempotent, even if the returned status code may change between requests:
  - DELETE /idX/delete HTTP/1.1 -> Returns 200 if idX exists
  - DELETE /idX/delete HTTP/1.1 -> Returns 404 as it just got deleted
  - DELETE /idX/delete HTTP/1.1 -> Returns 404

Q

How should you structure a web app?

KISS! DRY!

Do not pack multiple functionalities into a servlet.

Do not put different functonalities in POST & GET

e.g.

Instead of a single AccessDataServlet read/update with GET
create with POST

create three servlets:

ReadDataServlet implemented with GET

UpdateDataServlet implemented with GET

CreateDataServlet implemented with POST

(exception: if you want to hide params)

Model, View, Controller

The Model defines the business layer of the application,

the Controller manages the flow of the application,

the View defines the presentation layer of the application.

Usually:

model -> Java classes

controller -> servlets

view -> jsps

Example: the model

```
public class Student {
    private int id;
    private String firstName;
    private String lastName;
    // constructors, getters ,setters, toJson() go here
}
public class StudentService {
    public Student getStudent(int id) {
        switch (id) {
            case 1:
                return new Student(1, "John", "Doe");
            case 2:
                return new Student(2, "Jane", "Goodall");
            case 3:
                return new Student(3, "Max", "Born");
            default:
                return null;
        }
    }
}
```

defines our miniworld:
the things
we're talking about
and the actions
we can perform on them

adapted from https://www.baeldung.com/

Version Servlet+JSP

Example: the controller

```
@WebServlet(name = "StudentServlet", urlPatterns = "/student-record")
public class StudentServlet extends HttpServlet {
   private StudentService studentService = new StudentService();
   @Override
   protected void doGet(HttpServletRequest request, HttpServletResponse response)
                                        throws ServletException, IOException {
      String studentID = request.getParameter("id");
      if (studentID != null) {
         int id = Integer.parseInt(studentID);
         Student s=studentService.getStudent(id);
         if (s!=null) request.setAttribute("studentRecord", s));
      }
      RequestDispatcher dispatcher = request.getRequestDispatcher(
        "/WEB-INF/jsp/student-record.jsp");
      dispatcher.forward(request, response);
   }
}
```

get the invocations
from the user, maps them
into actions onto the miniworld,
obtains results,
gives them to the view

Example: the view (as a JSP)

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@page import="it.unitn.ronchet.Student" %>
<!DOCTYPE html>
<html> <head> <title>Student Record</title> </head>
   <body>
   <%
      if (request.getAttribute("studentRecord") != null) {
         Student student = (Student) request.getAttribute("studentRecord");
   %>
   <h1>Student Record</h1>
   <div>ID: <%= student.getId()%></div>
   <div>First Name: <%= student.getFirstName()%></div>
   <div>Last Name: <%= student.getLastName()%></div>
   <%
      } else {
   %>
   <h1>No student record found.</h1>
   <% } %>
    <i>Thanks to JSP technology</i>
   </body>
</html>
```

Shows the retrieved data

Presentation logic is done by JSP

Version Servlet+JSP+JS

Example: the controller

```
@WebServlet(name = "ControllerForJS", urlPatterns = "/ControllerForJS")
public class ControllerForJS extends HttpServlet {
   private StudentService studentService = new StudentService();
   @Override
   protected void doGet(HttpServletRequest request, HttpServletResponse response)
                                        throws ServletException, IOException {
      String studentID = request.getParameter("id");
      if (studentID != null) {
         int id = Integer.parseInt(studentID);
         Student s=studentService.getStudent(id);
         if (s!=null) request.setAttribute("studentJson", s.toJson());
      }
      RequestDispatcher dispatcher = request.getRequestDispatcher(
       "/WEB-INF/jsp/view_js.jsp");
      dispatcher.forward(request, response);
   }
}
```

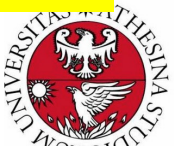
get the invocations
from the user, maps them
into actions onto the miniworld,
obtains results,
gives them to the view

Example: the view (as a JSP+JS)

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@page import="it.unitn.ronchet.Student" %>
<!DOCTYPE html>
<html>
   <head>
      <title>Javascript student viewer</title>
      <meta charset="UTF-8">
      <meta name="viewport" content="width=device-width, initial-scale=1.0">
      <script src="viewer.js"></script>
   </head>
   <body>
      <script>
         prepareData2(<%=request.getAttribute("studentJson")%>);
      </script>
      <i>Thanks to JSP technology</i>
   </body>
</html>
```

Shows the retrieved data

Presentation logic is delegated to JS

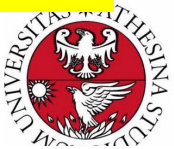


Example: the view (as a JSP+JS)

```
function prepareData(student){
   var div=document.createElement("div");
   if (student) {
      div.innerHTML="<h1>Student Record</h1>"+
      "<div>ID:"+student.id+"</div>"+
      "<div>First Name: "+student.firstName+"</div>"+
      "<div>Last Name: "+student.lastName+"</div>";
   } else {
      div.innerHTML="<h1>No student record found.</h1>";
   }
   div.innerHTML=div.innerHTML+"<i>Thanks to JS technology</i>";
   document.body.appendChild(div);
}
```

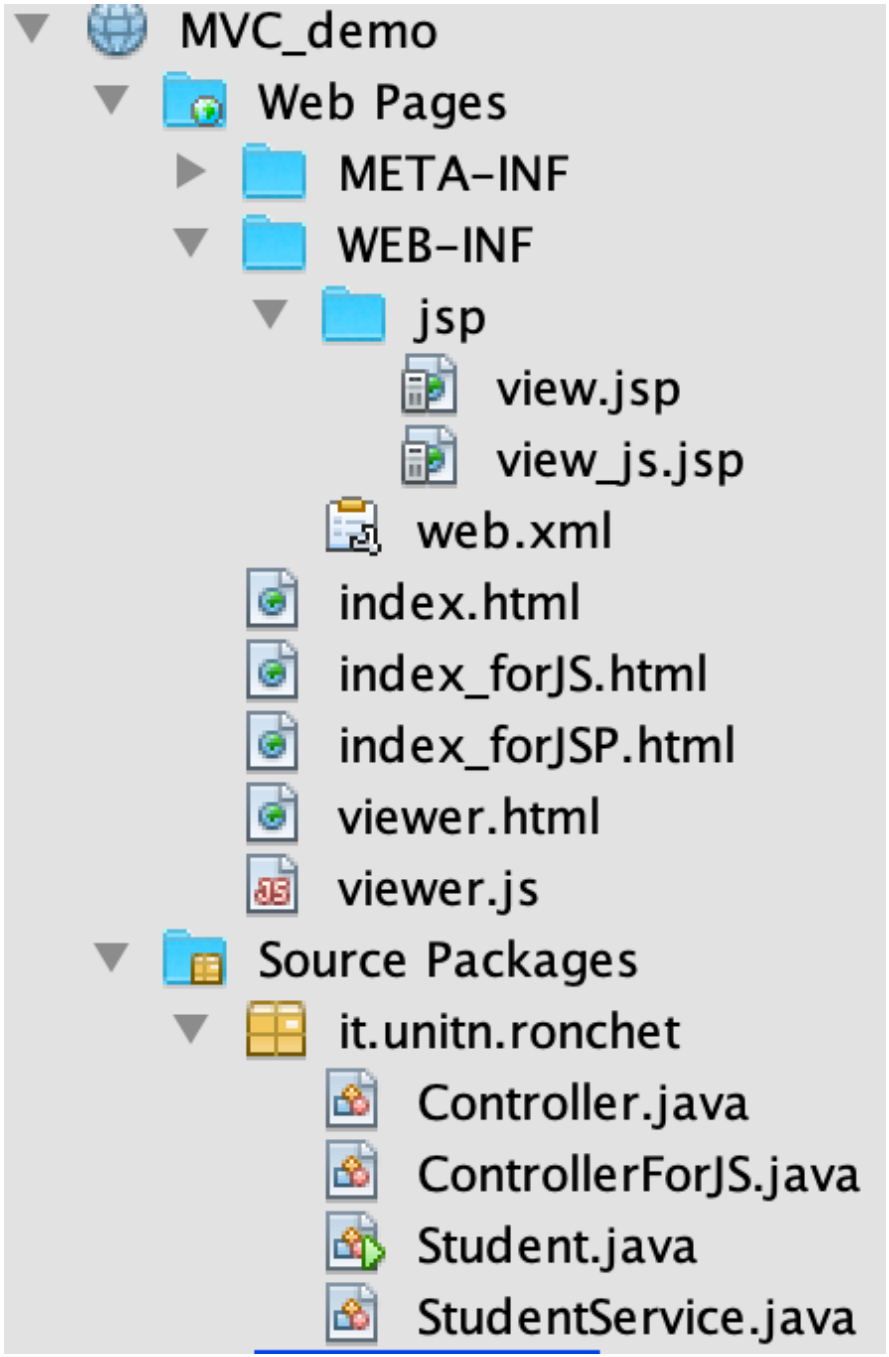
Shows the retrieved data

Presentation logic is delegated to JS

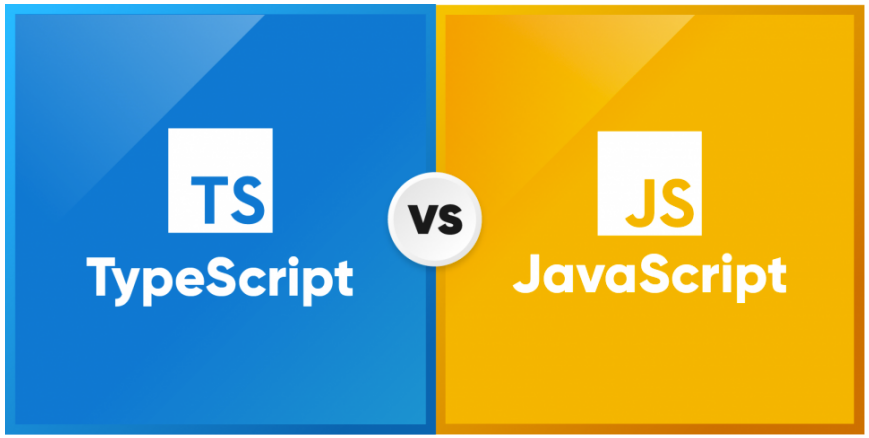


Q

Why jsp in web-inf?



Making life easier with Typescript



Some slides adapted from LifeMichael.com

Polyfilling and transpiling

- Polyfiling is one of the methodologies that can be used as a sort of backward compatibility measurement.
- “A polyfill, or polyfiller, is a piece of code (or plugin) that provides the technology that you, the developer, expect the browser to provide natively. (Remy Sharp).”

- a “Transpiler” is a tool that transforms code with newer syntax into older code equivalents. This process is called “Transpiling”.

TypeScript

- The TypeScript programming language was developed by Microsoft. It is an open source programming language.
- The code we write in TypeScript is compiled into JavaScript (transpiled)
- TypeScript gives us the capabilities, which are required to develop large scale applications using JavaScript.

TypeScript

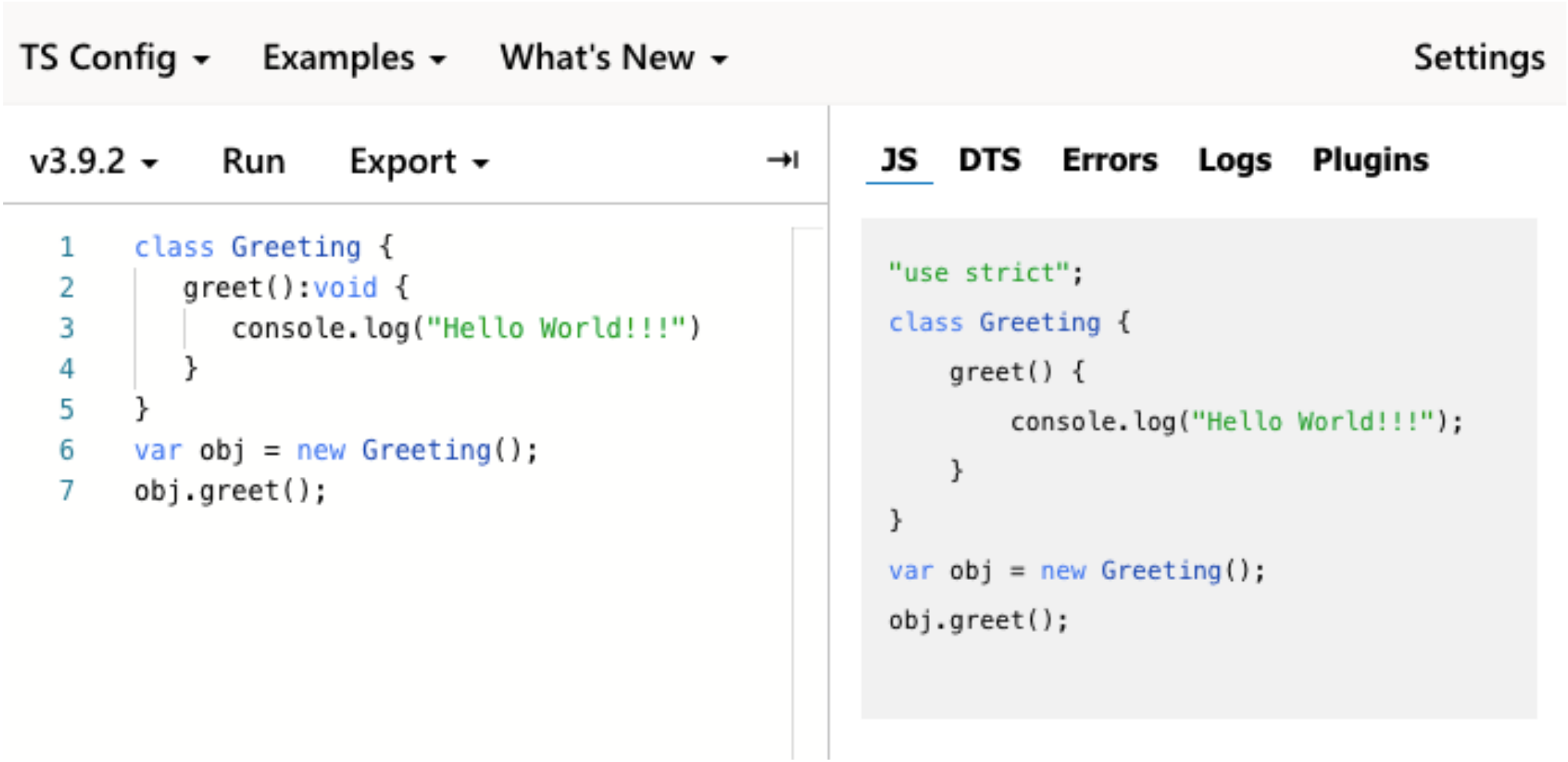
- TypeScript is a superset of JavaScript. It includes the entire JavaScript programming language together with additional capabilities.
- TypeScript allows us to use JavaScript as if it was a strictly type programming language.
  - TypeScript allows us to specify the type of the variables.
  - TypeScript allows us to define classes and interfaces.

TypeScript

- In general, nearly every code we can write in JavaScript can be included in code we write in TypeScript.
- Compiling TypeScript into JavaScript we get a clean simple ES3 compliant code we can run in any web browser

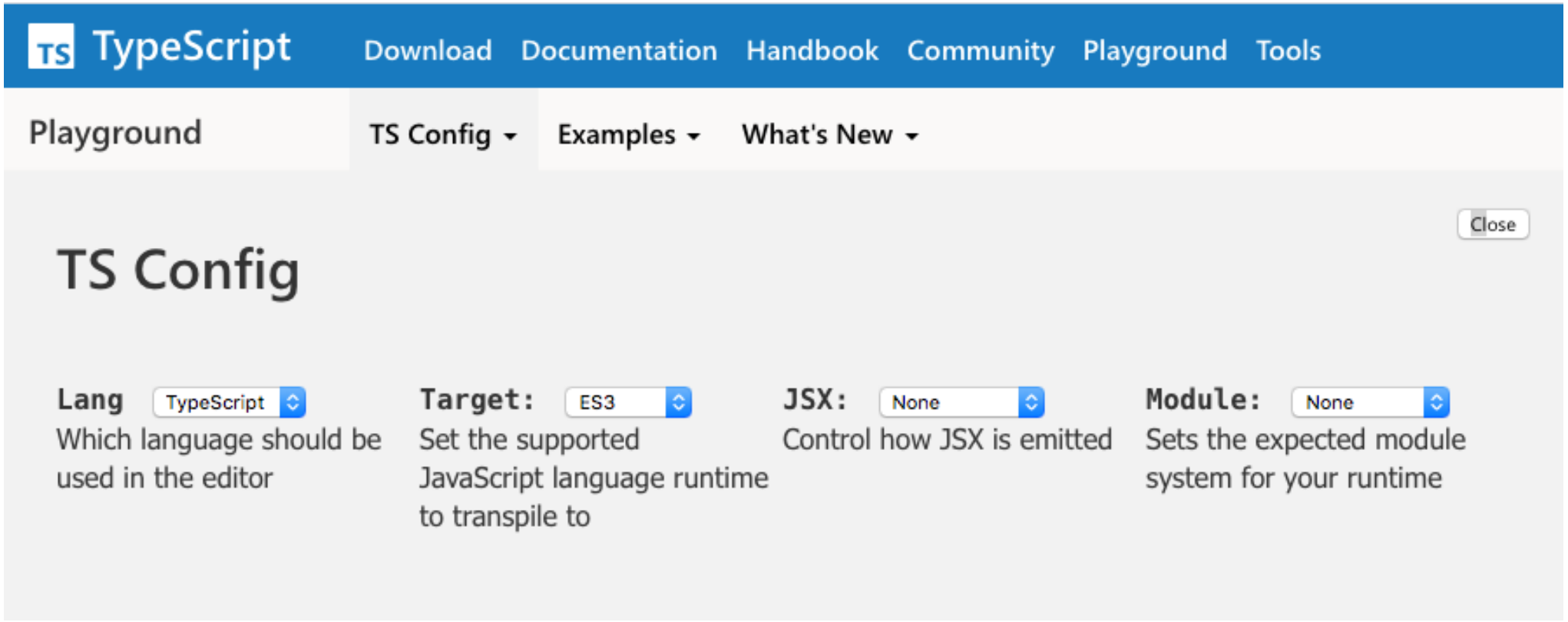
The TypeScript playground

https://www.typescriptlang.org/play/



EcmaScript 2017

Configuring TypeScript playground



Configuring TypeScript playground

TS Config ▾ Examples ▾ What's New ▾ Settings

v3.9.2 ▾ Run Export ▾ ⇥

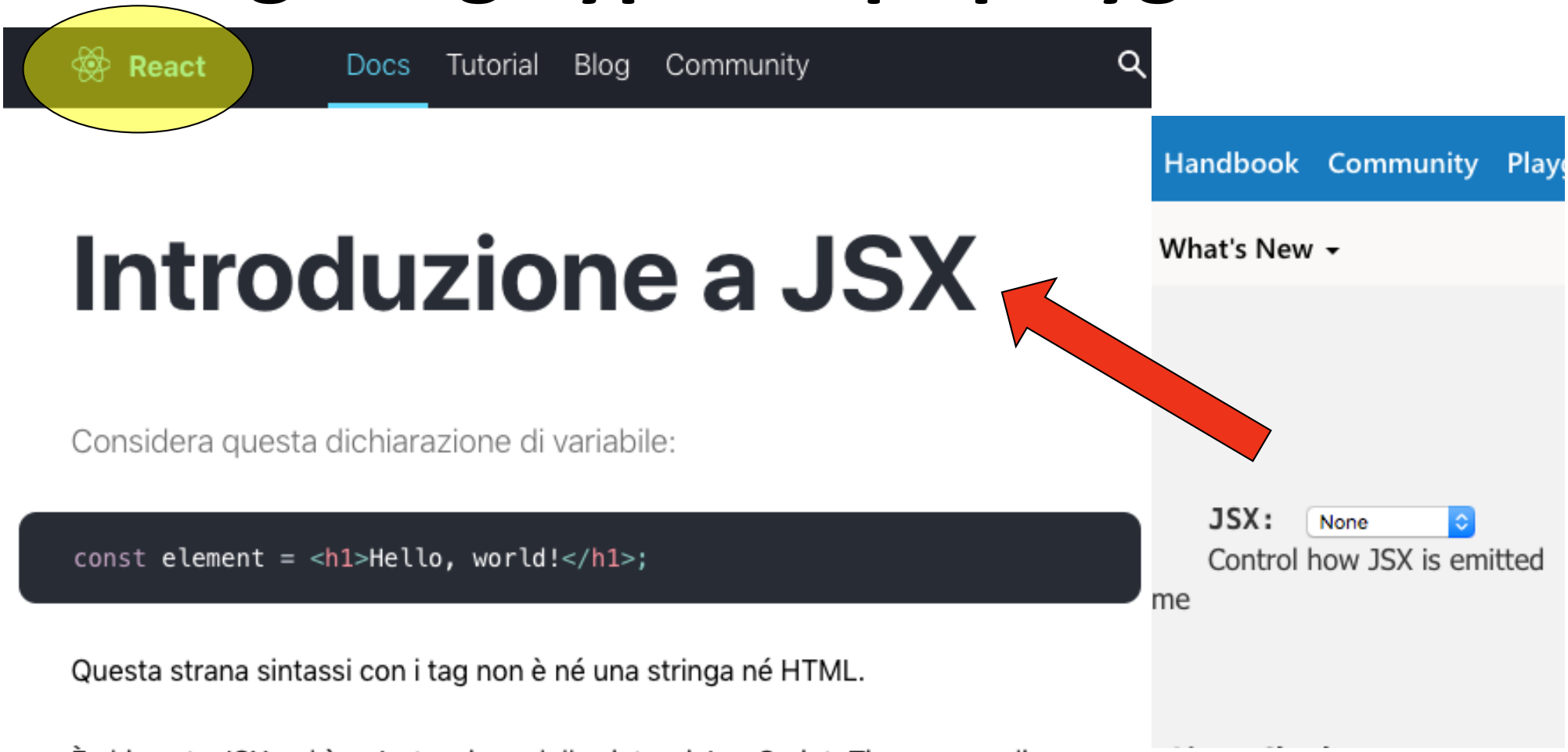
```
1   class Greeting {
2       greet():void {
3           console.log("Hello World!!!")
4       }
5   }
6   var obj = new Greeting();
7   obj.greet();
```

JS DTS Errors Logs Plugins

```
"use strict";
var Greeting = /** @class */ (function () {
    function Greeting() {
    }
    Greeting.prototype.greet = function ()
        console.log("Hello World!!!");
    };
    return Greeting;
}());
var obj = new Greeting();
obj.greet();
```

ES 3

Configuring TypeScript playground



Variable typing

We define the type of the variables:

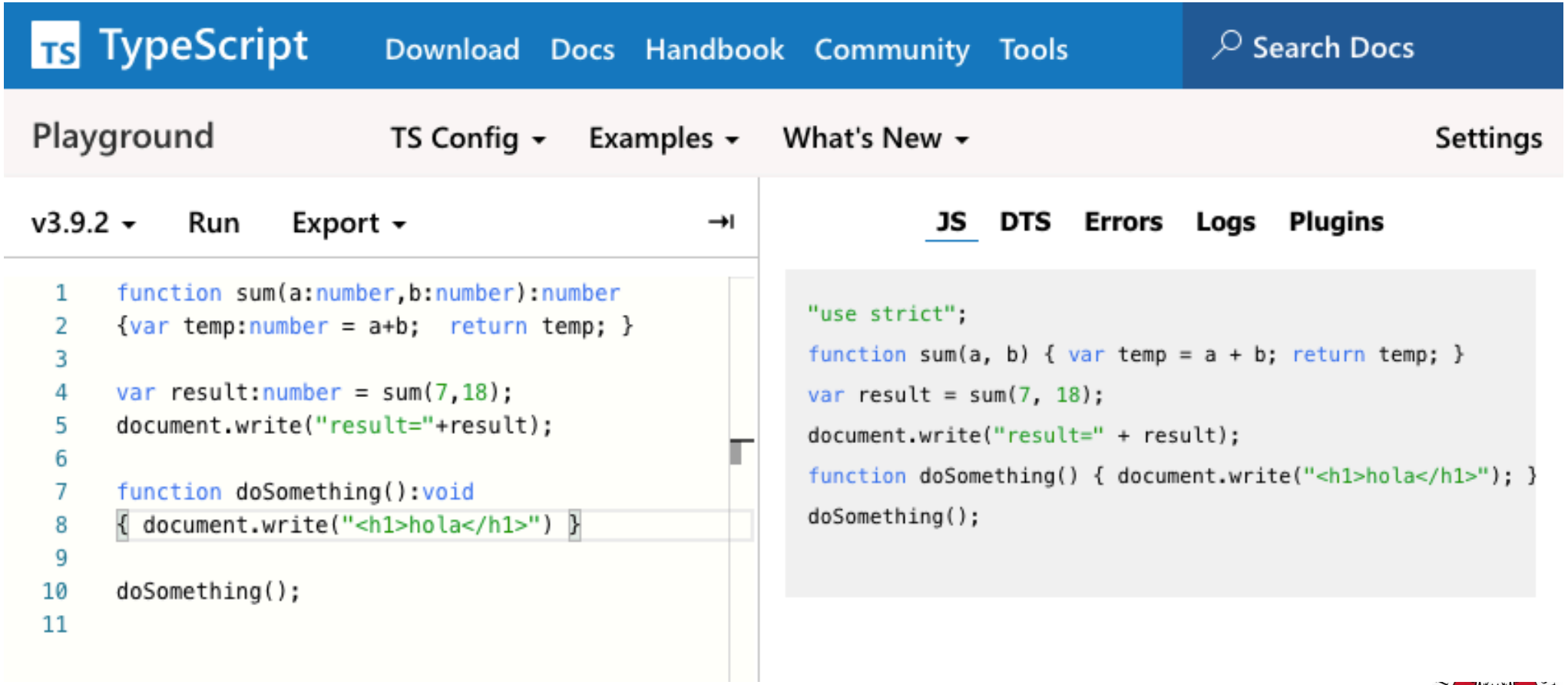
name:type

```
var id:number = 221255;
var aname:string = "Dorothea";
var tall:boolean = true;
var names:string[] = ['pippo','pluto','minnie'];
```


Function typing

We define the type of the variables:

name:type



Number of params

- Unlike JavaScript, when calling a function passing over arguments the number of arguments must match the number of the parameters, otherwise then we get a compilation error.

Result: NaN

Optional params

- Adding the question mark to the name of a parameter will turn that parameter into an optional one.
- The optional parameters should be after any other required one. They should be the last ones.

```
function sum(a:number,b:number,c?:number):number
{
var total = 0;
if(c!==undefined) { total += c; }
total += (a+b); return total;
}

var temp = sum(7,8);
document.write("temp="+temp);
```

Default params

- When defining a function we can specify default values for any of its parameters. Doing so, if an argument is not passed over to the parameter then the default value we specified will be set instead.

Playground TS Config Examples What's New Settings

v3.9.2 Run Export

```
1  function dosomething(a:number,b:number,step:number=((b-a)%5)) {
2    var i=a;
3    while(a<=b) {
4     document.write("<br/>"+a);  a+=step;
5    }
6  }
7
8  dosomething(1,20);
9
```

JS DTS Errors Logs Plugins

```
"use strict";
function dosomething(a, b, step) {
    if (step === void 0) { step = ((b - a) % 5); }
    var i = a;
    while (a <= b) {
        document.write("<br/>" + a);
        a += step;
    }
}
dosomething(1, 20);
```


Rest params

- We can define a function with an arbitrary number of params (like the "main" in Java).

Playground TS Config Examples What's New Settings

v3.9.2 Run Export

```
1   function sum(...numbers: number[]):number
2   {
3   var total:number = 0;
4   for(var i=0; i<numbers.length; i++)
5   {
6   total += numbers[i];
7   }
8   return total;
9   }
10
11  document.write("<br/>"+sum(2,5,3));
12
13
```

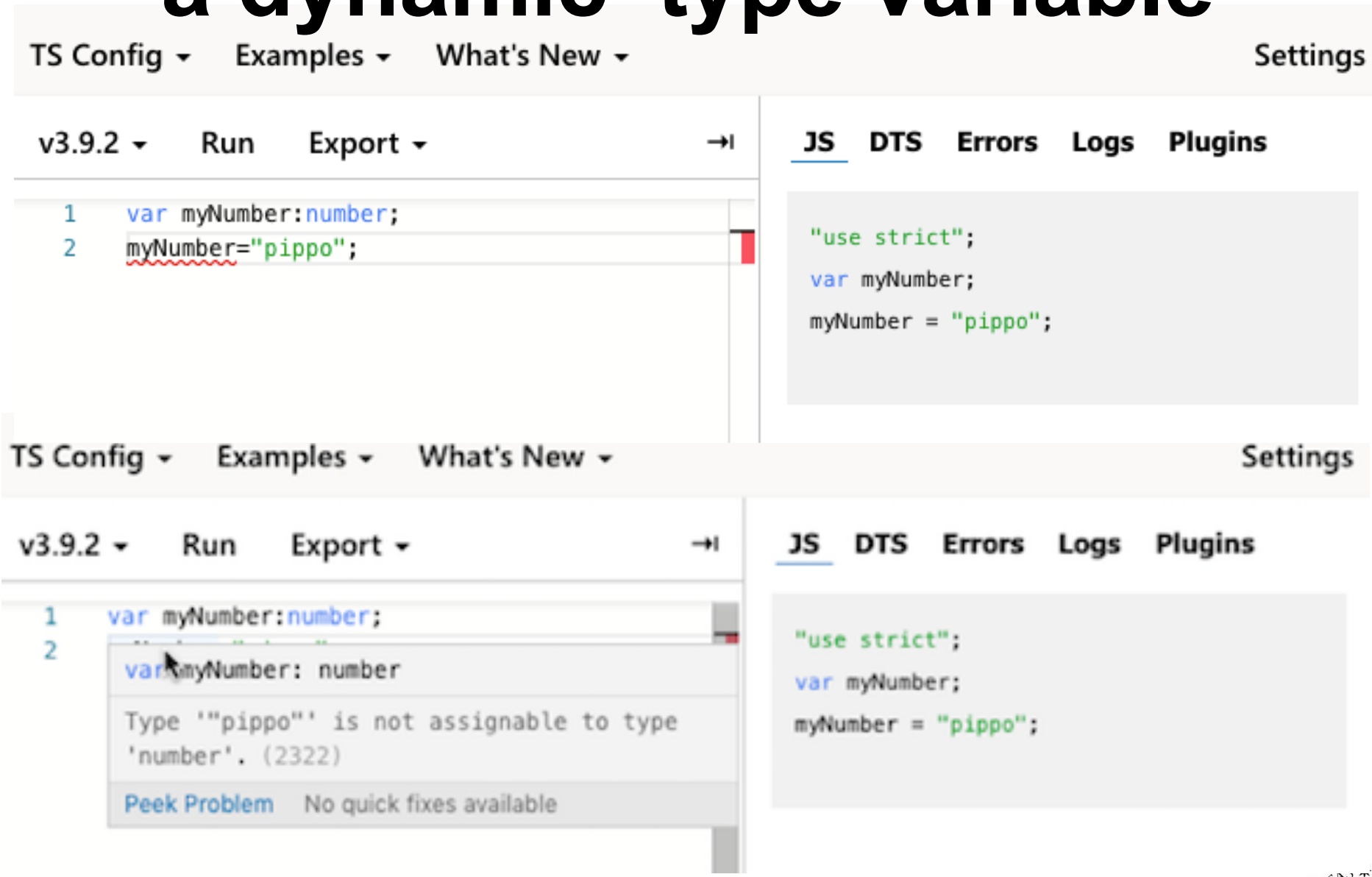
JS DTS Errors Logs Plugins

```
"use strict";
function sum() {
    var numbers = [];
    for (var _i = 0; _i < arguments.length; _i++)
        numbers[_i] = arguments[_i];
    }
    var total = 0;
    for (var i = 0; i < numbers.length; i++) {
        total += numbers[i];
    }
    return total;
}
document.write("<br/>" + sum(2, 5, 3));
```

TypeScript

- When starting from standard JavaScript, you might get some errors due to the differences between JavaScript and TypeScript.
- For instance, you get an error if you treat a variable in our code as if it was a dynamic type variable (as in JavaScript).
- Unlike other programming languages, when getting error messages from the TypeScript compiler it will still try to execute the code.

treating a variable as if it was a dynamic type variable



Dynamic type variables

We can create a variable with a dynamic type if we specify its type to be `any`.

```
var temp:any = 3;
temp = 'a';
temp = [23,5,23];
temp = true;
temp = new Object();
```


Classes

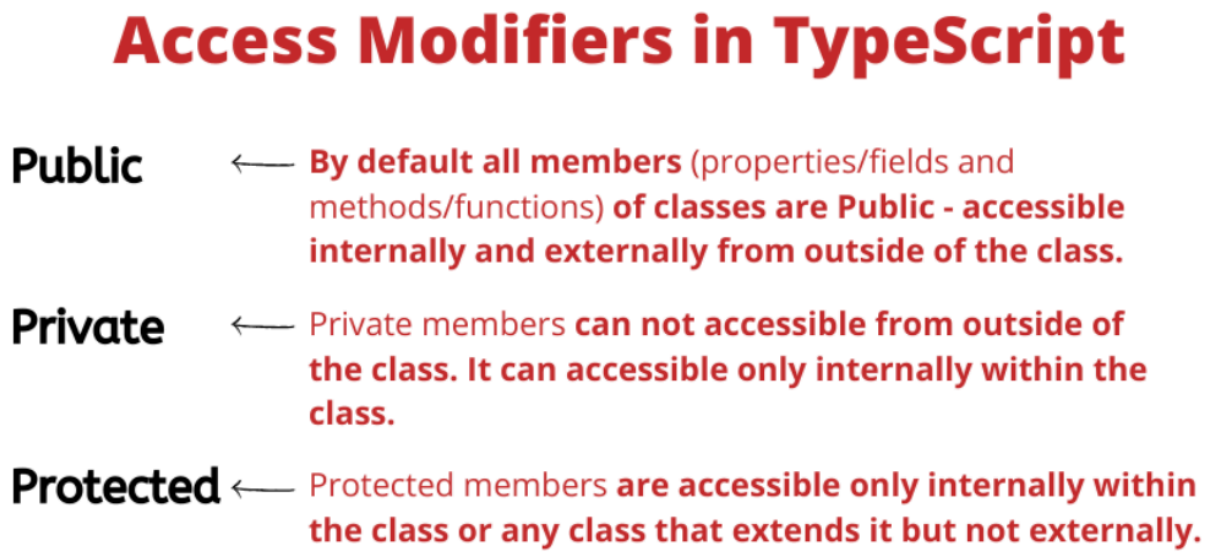
```
class Car {
   //field
   engine:string;

   //constructor
   constructor(engine:string) {
      this.engine = engine
   }

   //function
   disp():void {
      console.log("Engine is   :   "+this.engine)
   }
}
```

Constructor

- When we define a new class it automatically has a constructor, the default one.
- We can define a new constructor. When doing so, the default one will be deleted.
- There is no constructor polymorfism.
- When we define a new constructor we can specify each one of its parameters with an access modifier and by doing so indirectly define those parameters as instance variables

Access modifiers

- The available access modifiers are `private`, `public` and `protected`. The `public` access modifier is the default one. If we don't specify an access modifier then it is `public`.



Instance vars and methods

- The variables are usually declared before the constructor. Each variable definition includes three parts. The optional access modifier, the identifier and the type annotation.
- The methods are declared without using the function keyword. We can precede the function name with an access modifier and we can append the function declaration with the type of its returned value.

Instance vars and methods

```
class Rectangle
{
    private width:number;
    private height:number;
    constructor(width:number,height:number)
    {
        this.width = width;
        this.height = height;
    }

    protected area():number
    {
        return this.width*this.height;
    }

}
```

Static vars and methods

- We can define static variables and static methods by adding the `static` keyword. Accessing `static` variables and methods is done using the name of the class.

Static vars and methods

```
class FinanceUtils
{
    public static VAT = 0.18;
    public static calculateVAT(sum:number):number
    {
        return FinanceUtils.VAT*sum;
    }
}

var price:number = 1020;
document.write("<br/>"+FinanceUtils.calculateVAT(price));
```


Class inheritance

```
class Shape {
   Area:number

   constructor(a:number) {
      this.Area = a
   }
}

class Circle extends Shape {
   disp():void {
      console.log("Area of the circle:  "+this.Area)
   }
}

var obj = new Circle(223);
obj.disp()
```

Type assertion

- *Type assertions* are a way to tell the compiler "trust me, I know what I'm doing." A type assertion is like a type cast in other languages, but performs no special checking. It has no runtime impact, and is used purely by the compiler.

```
class Person {
  id:number;
  name:string;
}

class Student extends Person
{
  average:number;
}

var a:Person = new Student();
var b:Student = <Student>a;
```


Generics

function identity<T>(arg: T): T { return arg; }

Usages:

explicit form:

let output = identity<string>("myString"); // ^ = let output: string

implicit form:

let output = identity("myString"); // ^ = let output: string

see https://www.typescriptlang.org/docs/handbook/generics.html

Other class related issues

- TypeScript doesn't support multiple inheritance.
- super

- Classes implement interfaces

```
class Student implements Iprintable {…}
```


The next big thing: interfaces

- We can use Interfaces as data type definition
- They fully disappear in JavaScript!

Playground TS Config Examples What's New Settings

v3.9.2 Run Export

```
1   interface IPerson {
2       firstName:string,
3       lastName:string,
4       sayHi: ()=>string
5   }
6
7   var customer:IPerson = {
8       firstName:"Tom",
9       lastName:"Hanks",
10      sayHi: ():string =>{return "Hi there"}
11  }
12
13  console.log("Customer Object ")
14  console.log(customer.firstName)
15  console.log(customer.lastName)
16  console.log(customer.sayHi())
```

JS DTS Errors Logs Plugins

```
"use strict";
var customer = {
    firstName: "Tom",
    lastName: "Hanks",
    sayHi: function () { return "Hi there"; }
};
console.log("Customer Object ");
console.log(customer.firstName);
console.log(customer.lastName);
console.log(customer.sayHi());
```

interfaces multiple inheritance

```
interface IParent1 {
    v1:number
}

interface IParent2 {
    v2:number
}

interface Child extends IParent1, IParent2 { }
var Iobj:Child = { v1:12, v2:23}
console.log("value 1: "+this.v1+" value 2: "+this.v2)
```


duck inheritance

```
class Vehicle {
    public run(): void { console.log('Vehicle.run'); }
}

class Task {
    public run(): void { console.log('Task.run'); }
}

function runTask(t: Task) {
    t.run();
}

runTask(new Task());
runTask(new Vehicle());
```


Duck Typing

- Duck typing in computer programming is an application of the duck test—"If it walks like a duck and it quacks like a duck, then it must be a duck"—to determine if an object can be used for a particular purpose. With normal typing, suitability is determined by an object's type.

avoiding duck inheritance - 1

```
class Vehicle {
    private x: string="A";
    public run(): void { console.log('Vehicle.run'); }
}

class Task {
    private x: string="A";
    public run(): void { console.log('Task.run'); }
}

function runTask(t: Task) {
    t.run();
}

runTask(new Task());
runTask(new Vehicle()); // Will be a compile time error
```

Argument of type 'Vehicle' is not assignable to parameter of type 'Task'.
Types have separate declarations of a private property 'x'.(2345)

avoiding duck inheritance - 2

```
class Vehicle {
    private x: string="A";
    public run(): void { console.log('Vehicle.run'); }
}

class Task {
    private s: string="A";
    public run(): void { console.log('Task.run'); }
}

function runTask(t: Task) {
    t.run();
}

runTask(new Task());
runTask(new Vehicle()); // Will be a compile time error
```

Argument of type 'Vehicle' is not assignable to parameter of type 'Task'.
Property 's' is missing in type 'Vehicle' but required in type 'Task'.(2345)

References for TypeScript

- http://www.typescriptlang.org
- https://www.tutorialspoint.com/typescript/index.htm