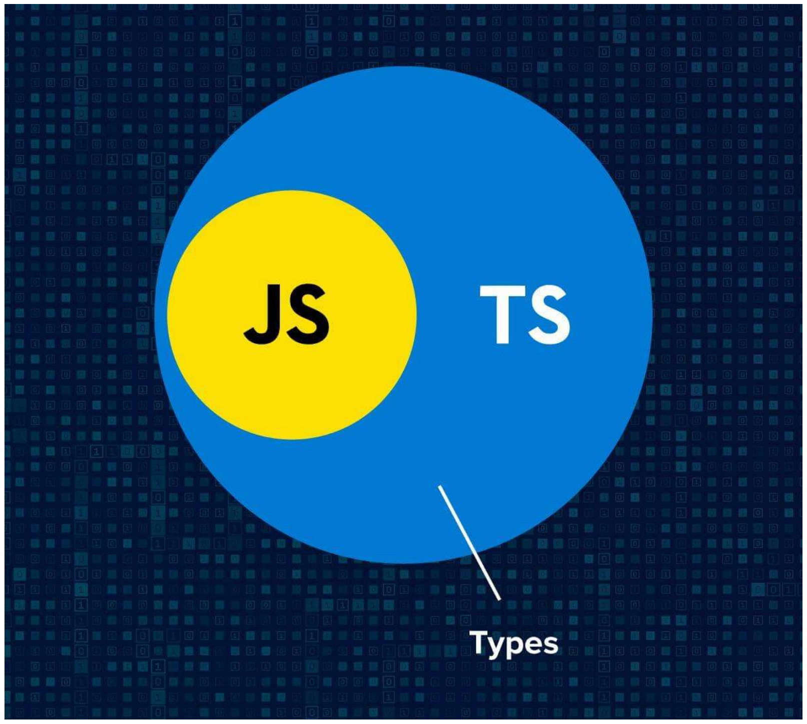
TypeScript

Pre-requisites

- HTML
- CSS
- JavaScript

What is TypeScript?

- **TypeScript is a language that is a typed superset of JavaScript meaning that it adds rules about how different kinds of values can be used**
- **TypeScript is also a programming language that preserves the runtime behavior of JavaScript.**
  - For example, dividing by zero in JavaScript produces Infinity instead of throwing a runtime exception. As a principle, TypeScript never changes the runtime behavior of JavaScript code.
  - This means that if you move code from JavaScript to TypeScript, it is **guaranteed** to run the same way, even if TypeScript thinks that the code has type errors.



The Problem

- Let’s understand the problem with a simple experiment

Should I Learn JavaScript or TypeScript ?

- The answer is that you can’t learn TypeScript without learning JavaScript! TypeScript shares syntax and runtime behavior with JavaScript, so anything you learn about JavaScript is helping you learn TypeScript at the same time.

Installing and Using TS

- Install Node.js
- Install TypeScript using npm

TS Advantages

- TS adds Types
  - Avoids unnecessary and unexpected errors
  - Clarify the intentions of using variables
- Support for non-Javascript features such as Interfaces and Generics
- Support for next generation JS features(compiled for older browsers)
- Rich configuration options
- Meta-programming features such as Decorators
- Modern Tooling that helps even in non-TS projects
  - Babel

TS Project Setup

- Install TSC, TSLint, and type declarations for NodeJS

```
npm install --save-dev typescript tslint @types/node
npm install --g typescript tslint @types/node
```

- Initialize a Node project

```
npm init
```

TS Project Setup

- Every TS project should have a **tsconfig.json** in the root directory
- This is where TypeScript projects define things like which files should be compiled, which directory to compile them to, and which version of JavaScript to emit.
- Issue the following command:

```
tsc --init
```

```
{
    "compilerOptions": {
    "lib": ["es2015"],
    "module": "commonjs",
    "outDir": "dist",
    "sourceMap": true,
    "strict": true,
    "target": "es2015"
    },
    "include": [
    "src"
    ]
}
```

TS Project Setup

- Your project should also have a **tslint.json** file containing your TSLint configuration, codifying whatever stylistic conventions you want for your code (tabs versus spaces, etc.).
- The command will create the tslint.json file

  tslint --init

- TSLint Documentation specifies the full set of rules

```
{
"defaultSeverity":
"error",
"extends": [
"tslint:recommended"
],
"rules": {
"semicolon": false,
"trailing-comma": false
}
}
```

TS Project Setup

- Create a HTML file
- Source **app.ts** in the HTML file
- Write some code in **app.ts**
- Compile using the **tsc** command

```
tsc app.ts
```

TS Project Setup

- Serving Live

Exercise

- Setup a project folder
- Become familiar with TS work-flow

Using Types

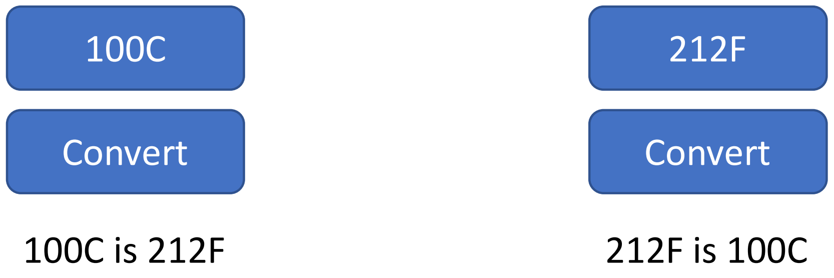
number	5, 5.67, -7.56	All numbers, no differentiation between integers and floats
string	‘type’, “script”, `ts`	All text values
boolean	true, false	Truth values

Remember:

TS is statically typed
JS only knows about a few types, TS knows more types
TS only helps during development, but does not change the run time logic
TS also does a type inference with what ever value a variable is initialized with, you will not be able to further assign a new value of a different type

Creative Lab #1.1

- Create a temperature converter application
- Write a single function to convert temperature from Celsius to Fahrenheit and vice versa
- Add a parameter, say verbosity, that specifies whether to print the result in a detailed or short manner



```
100C is 212F
212F is 100C
```

Arrays, Tuples, Enums

array	[1,2,3]	Any JS array, type can be flexible or strict (regarding element types)
tuples	[1,2]	Fixed length array with fixed types
enum	enum{cat, dog}	Assigns labels to numbers, custom type(first)

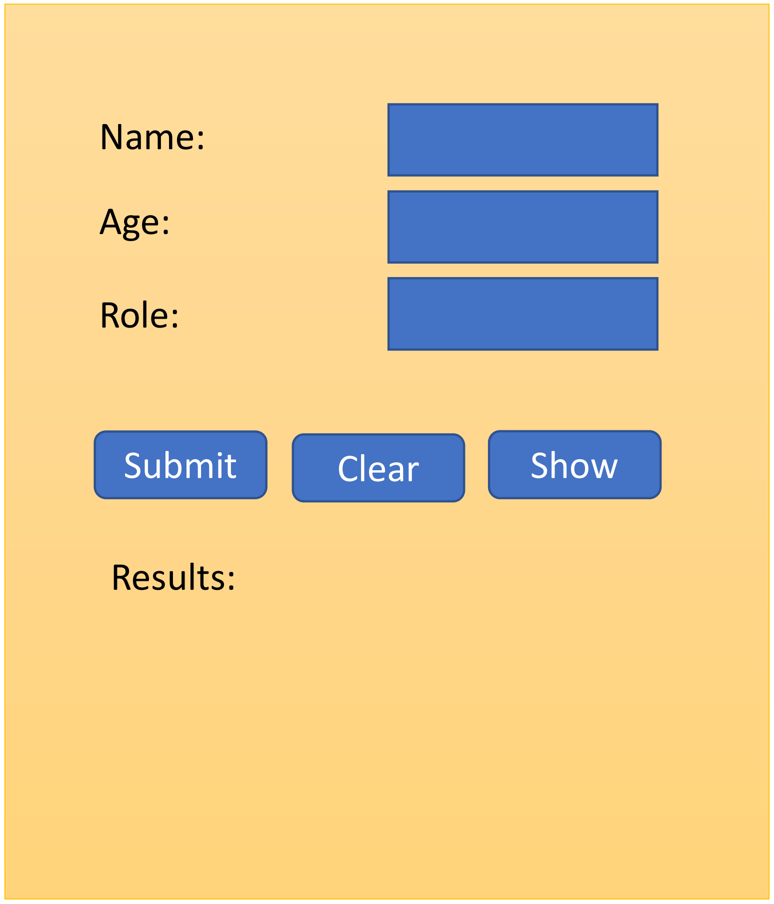
```
let favActivities: string[];
favActivities = ['Sports', 'Cooking'];
```

```
let role: [number, string];
role = [1, 'Author'];
```

```
enum Role {ADMIN, AUTHOR, READONLY};
role = Role.ADMIN;
if(role == Role.ADMIN){}
```

Creative Lab #2

- Create a HTML file as shown
- When Submit is pressed, record the data
- When Clear is pressed remove the data
- When Show is pressed, show the data recorded



Object Types

object

{age:30}

Any JS object, more specific are possibile

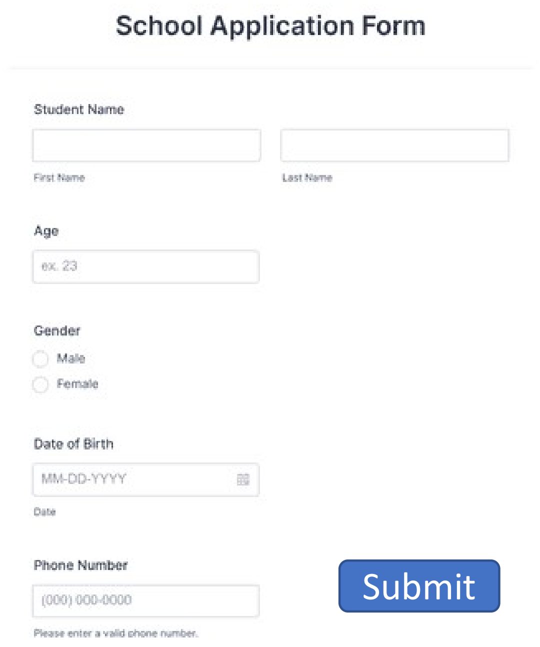
```
const person: {
   name: string;
   age: number;
 } = {
  name: 'Purushotham',
  age: 35
};

console.log(person.name);
```

Nested Objects

```
const product = {
  id: 'abc1',
  price: 12.99,
  tags: ['great-offer', 'hot-and-new'],
  details: {
    title: 'Red Carpet',
    description: 'A great carpet - almost brand-new!'
  }
}
```

```
{
  id: string;
  price: number;
  tags: string[];
  details: {
    title: string;
    description: string;
  }
}
```

Creative Lab #1.3



School Application Form

Student Name

First Name

Last Name

Age

ex. 23

Gender

Male

Female

Date of Birth

MM-DD-YYYY

Date

Phone Number

(000) 000-0000

Please enter a valid phone number.

Any type

any	*	Any kind of value, no specific type assignment

```
let favActivities: any; // any[]
favActivities = 'anything'
```

Union Type

Fine with either number or string

```
function combine(input1: number | string, input2: number | string) {
  let result;
  if (typeof input1 === 'number' && typeof input2 === 'number') {
    result = input1 + input2;
  } else {
    result = input1.toString() + input2.toString();
  }
  return result;
}

const combinedAges = combine(30, 26);
console.log(combinedAges);

const combinedNames = combine('Max', 'Anna');
console.log(combinedNames);
```

Run time type check can be necessary

Literal Types

```
function combine(
  input1: number | string,
  input2: number | string,
  resultConversion: 'as-number' | 'as-text’
) { … }

const combinedStringAges = combine('30', '26', 'as-number');
```

Only these specific values are allowed

Type Aliases/Custom Types

- Type aliases can be used to "create" your own types.

Reusable custom types

```
type Combinable = number | string;
type ConversionDescriptor = 'as-number' | 'as-text';

function combine(
  input1: Combinable,
  input2: Combinable,
  resultConversion: ConversionDescriptor
)
```

Type Aliases and Object Types

An object as a custom type

```typescript
type User = { name: string; age: number };

function greet(user: User) {
  console.log('Hi, I am ' + user.name);
}

function isOlder(user: User, checkAge: number)
{
  return checkAge > user.age;
}
```

Function Return Type and "void"

Return type is number

```
function add(n1: number, n2: number): number {
  return n1 + n2;
}

function printResult(num: number): void {
  console.log('Result: ' + num);
}

function printResult(num: number): undefined {
  console.log('Result: ' + num);
  return;
}
```

Difference b/n void and undefined: use undefined with you use return statement with nothing

Functions as Types

combineValues can take any function that can take two numbers and returns a number type

```
let combineValues: (a: number, b: number) => number;
combineValues = add;
// combineValues = printResult; // XXX
// combineValues = 5; //XXX

console.log(combineValues(8, 8));
```

Call Backs

```
function addAndHandle(n1: number, n2: number, callBack: (num: number) => void) {
  const result = n1 + n2;
  callBack(result);
}

addAndHandle(10, 20, (result) => {
  console.log(result);
});
```

Callback function is specified using a arrow function

The “unknown” Type

- The data type is not guaranteed in **unknown**, it is different from **any**
- When the user is unsure of the data type, but what will done on it eventually
- Extra type check is required

```
let userInput: unknown;
let userName: string;

userInput = 5;
userInput = 'Max';
if (typeof userInput === 'string') {
  userName = userInput;
}
```

The “never” Type

- When your intention is to never return a value

```
function generateError(message: string, code: number): never {
  throw { message: message, errorCode: code };
  // while (true) {}
}

generateError('An error occurred!', 500);
```

Type Casting

- JavaScript doesn’t have a concept of type casting because variables have dynamic types. However, every variable in TypeScript has a type.
- Type castings allow you to convert a variable from one type to another.

```
let input = document.querySelector('input["type="text"]');
console.log(input.value); // Causes Error
```

- To fix the code:

```
let input = document.querySelector('input[type="text"]') as HTMLInputElement;
let input = <HTMLInputElement>document.querySelector('input[type="text"]');
```

- Use the as keyword or <> operator for type castings.

Conditionals: if

- The **if..else if.. else** construct is useful when multiple conditions are available and needs to be tested

```
var num: number = 2
if(num > 0) {
   console.log(num+" is positive")
} else if(num < 0) {
   console.log(num+" is negative")
} else {
   console.log(num+" is neither positive nor negative")
}
```

Creative Lab #1.4

- Write an application to input name, age and then suggest movies based on the age of the person
- Improvise with presenting the trailer of the movies
- Keep a movie:link object for different age groups

Conditionals: switch

- The **switch** statement evaluates an expression, matches the expression's value to a case clause, and executes statements associated with that case.

```
var grade:string = "A";
switch(grade) {
   case "A": {
      console.log("Excellent");
      break;
   }
   case "B": {
      console.log("Good");
      break;
   }
   case "C": {
      console.log("Fair");
      break;
   }
   case "D": {
      console.log("Poor");
      break;
   }
   default: {
      console.log("Invalid choice");
      break;
   }
}
```

Loops: for

- The **for** loop executes the code block for a specified number of times. It can be used to iterate over a fixed set of values, such as an array.

```
var num:number = 5;
var i:number;
var factorial = 1;

for(i = num;i>=1;i--) {
   factorial *= i;
}
console.log(factorial)
```

Loops: while

- The **while** loop executes the instructions each time the condition specified evaluates to true. In other words, the loop evaluates the condition before the block of code is executed.

```
var num:number = 5;
var factorial:number = 1;

while(num >=1) {
   factorial = factorial * num;
   num--;
}
console.log("The factorial  is "+factorial);
```

Loops: do..while

- The **do..while** loop executes the instructions each time the condition specified evaluates to true.

```
var n:number = 10;
do {
   console.log(n);
   n--;
} while(n>=0);
```

Creative Lab #2.1

- Write an application to determine the factors of an input number

Functions: Declaring and Invoking

- In JavaScript, functions are first-class objects. That means you can use them exactly like you would any other object:
  - Assign them to variables, pass them to other functions, return them from functions, assign them to objects and prototypes, write properties to them, read those properties back, and so on.
  - TypeScript models all of those things with its rich type system.
- This is how a function declaration looks like in TS:

```
function add(a: number, b: number): number {
    return a + b
}
```

Function Declaration

```
// Named function
function greet(name: string) {
   return 'hello ' + name
}
// Function expression
let greet2 = function(name: string) {
   return 'hello ' + name
}
// Arrow function expression
let greet3 = (name: string) => {
   return 'hello ' + name
}
// Shorthand arrow function expression
let greet4 = (name: string) =>'hello ' + name
// Function constructor
let greet5 = new Function('name', 'return "hello " + name')
```

Default Parameters

```
function log(message: string, userId = 'Not signed in') {
    let time = new Date().toISOString()
    console.log(time, message, userId)
}

log('User clicked on a button', 'da763be')
log('User signed out')
```

Optional Parameters

- When declaring your function's parameters, required parameters have to come first, followed by optional parameters:

```
function log(message: string, userId?: string) {
    let time = new Date().toLocaleTimeString()
    console.log(time, message, userId || 'Not signed in')
}

log('Page loaded') // Logs "12:38:31 PM Page loaded Not signed in"
log('User signed in', 'da763be') // Logs "12:38:31 PM User signed in da763be"
```

Creative Lab #2.2

- Write a function to calculate the word histogram of a given text
- Take the text from the textarea element (use a submit button as well)
- Example:
  - Input: Mary had a little little lamb
  - Output: {“Mary”:1, “had”:1, “a”:1, “little”:2, “lamb”:1}
  - Function call: getHist(text, ‘as-array’ | ‘as-object’)

Variable Number of Arguments

- **arguments** is an Array-like object accessible inside functions that contains the values of the arguments passed to that function.

```
function func1(a, b, c) {
  console.log(arguments[0]);
  // expected output: 1

  console.log(arguments[1]);
  // expected output: 2

  console.log(arguments[2]);
  // expected output: 3
}

func1(1, 2, 3);
```

```
function sumVariadic(): number {
    return Array
    .from(arguments)
    .reduce((total, n) => total + n, 0)
}
sumVariadic(1, 2, 3) // evaluates to 6 in JS
```

unsafe

Rest Parameters

- A function can have at most one rest parameter, and that parameter has to be the last one in the function's parameter list.

```
function sum( ...numbers: number[] ): number {
    return numbers.reduce((total, n) => total + n, 0)
}

sum(1, 2, 3) // evaluates to 6
```

The call, apply, bind Functions

- In addition to invoking a function with parentheses (), JavaScript supports few other ways to call a function

```
function add(a: number, b: number): number {
    return a + b
}

add(10, 20) // evaluates to 30
add.apply(null, [10, 20]) // evaluates to 30
add.call(null, 10, 20)    // evaluates to 30
add.bind(null, 10, 20)()  // returns a function object, should
                          // be called using ()evaluates to 30
```

this <= null

To safely use .call, .apply, and .bind in your code, be sure to enable the strictBindCallApply option in your *tsconfig.json* (it's automatically enabled if you already enabled strict mode).

Typing this

- The general rule is that this will take the value of the thing to the left of the dot when invoking a method.
- Say you have a utility function for formatting dates that looks like this:

```
 function fancyDate() {
     return `${this.getDate()}/${this.getMonth()}/${this.getFullYear()}`
     }
```

- To use fancyDate, you have to call it with a Date bound to **this**:

```
fancyDate.call(new Date) // evaluates to some date such as"4/4/2020"
```

- If you forget to bind a Date to this, you'll get a runtime exception!

```
fancyDate() // Uncaught TypeError: this.getDate is not a function
            // Run-time error
```

Typing this

- TS has an elegant technique to resolve this: If your function uses this, be sure to declare your expected this type as your function's first parameter
- TypeScript will enforce that **this** really is what you say it is at every call

```
function fancyDate(this: Date) {
    return `${this.getDate()}/${this.getMonth()}/${this.getFullYear()}`
    }

fancyDate.call(new Date) // evaluates to some date such as"4/4/2020“
fancyDate() // Compile time error
```

To enforce that this types are always explicitly annotated in functions, enable the noImplicitThis setting in your tsconfig.json.

Generators

- Generator functions (generators for short) are a convenient way to, well, generate a bunch of values.
- They give the generator's consumer fine control over the pace at which values are produced.
- Because they're lazy—that is, they only compute the next value when a consumer asks for it—they can do things that can be hard to do other-wise, like generate infinite lists.

generator
consumer

Generators

```
function* createFibonacciGenerator() {
    let a = 0
    let b = 1
    while (true) {
    yield a;
    [a, b] = [b, a + b]
    }
}

let fibonacciGenerator = createFibonacciGenerator() // IterableIterator<number>

fibonacciGenerator.next() // evaluates to {value: 0, done: false}
fibonacciGenerator.next() // evaluates to {value: 1, done: false}
fibonacciGenerator.next() // evaluates to {value: 1, done: false}
fibonacciGenerator.next() // evaluates to {value: 2, done: false}
fibonacciGenerator.next() // evaluates to {value: 3, done: false}
fibonacciGenerator.next() // evaluates to {value: 5, done: false}
```

*** makes it a generator**

Generators

- You can also explicitly annotate a generator, wrapping the type it yields in an **IterableIterator**:

```
function* createNumbers(): IterableIterator<number> {
        let n = 0
        while (1) {
        yield n++
        }
}
let numbers = createNumbers()
numbers.next() // evaluates to {value: 0, done: false}
numbers.next() // evaluates to {value: 1, done: false}
numbers.next() // evaluates to {value: 2, done: false}
```

Iterators

- Iterators are the flip side to generators: while generators are a way to produce a stream of values, iterators are a way to consume those values.
- Any object that contains a property called **Symbol.iterator**, whose value is a function that returns an iterator.
- Any object that defines a method called **next**, which returns an object with the properties **value** and **done**.

Iterators

- An iterator to return numbers from 1 to 10:

```
let numbers = {
    *[Symbol.iterator]() {
    for (let n = 1; n <= 10; n++) {
    yield n
    }
  }
}
```

Iterators

- Not only can you define your own iterators, but you can use JavaScript’s built-in iterators for common collection types—Array, Map, Set, String, and so on—to do things like:

```
// Iterate over an iterator with for-of
for (let a of numbers) {
    // 1, 2, 3, etc.
}
// Spread an iterator
let allNumbers = [...numbers] // number[]
// Destructure an iterator
let [one, two, ...rest] = numbers // [number, number, number[]]
```

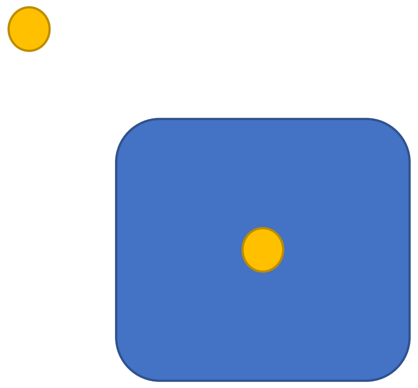
If you’re compiling your TypeScript to a JavaScript version older than ES2015, you can enable custom iterators with the **downlevelIteration** flag in your *tsconfig.json*.

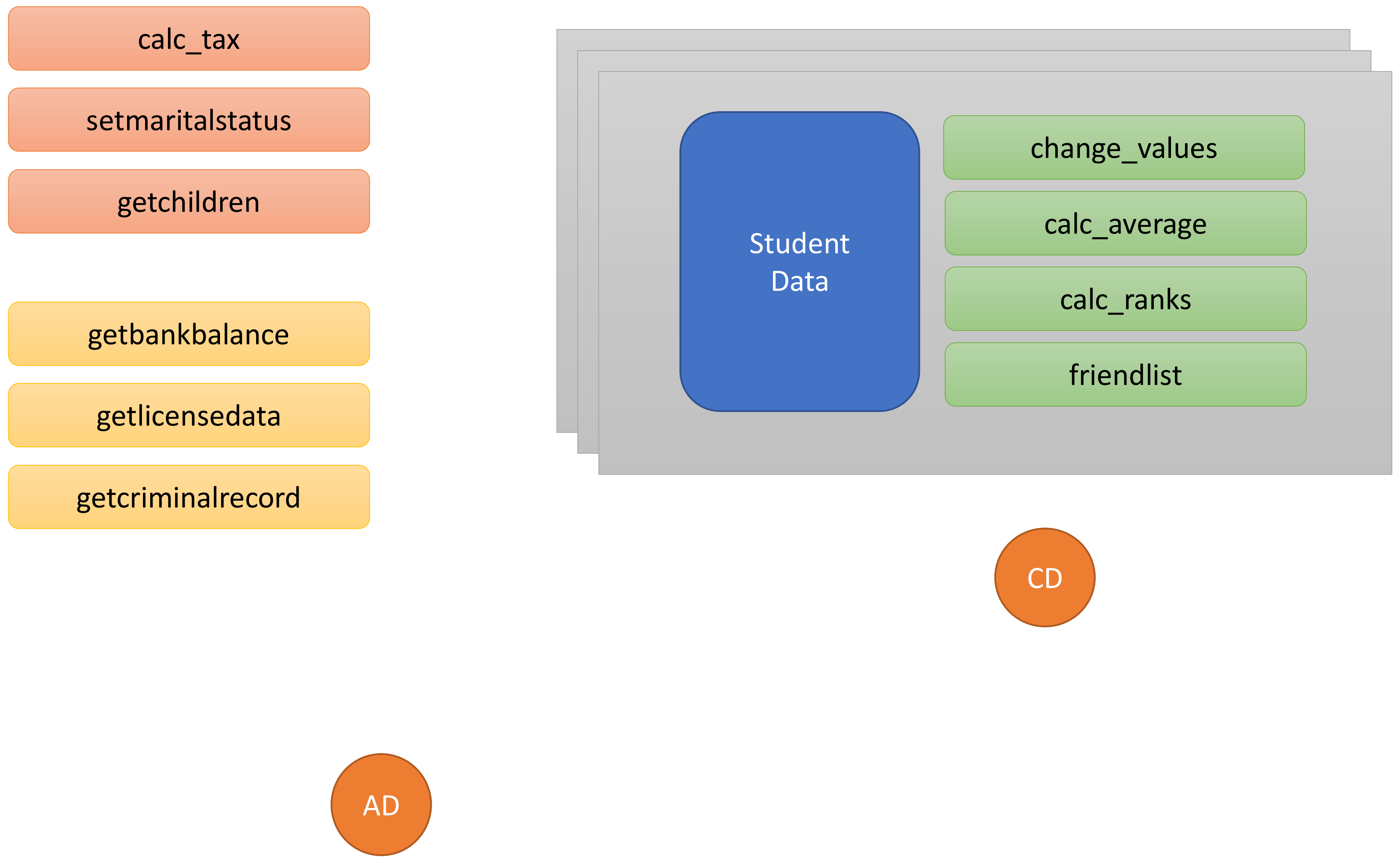
Creative Lab #2.3

- Create a word generator
- Create a word iterator

OOP

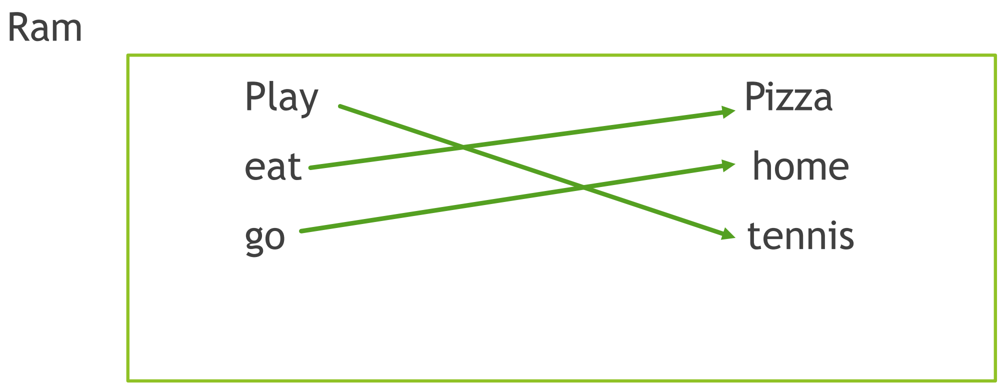
- Why Object Oriented Programming?
- Key issues:
  - Reusability
  - Enhancements and upgradability, scalability
  - Modularity
  - Testability
  - Usability
  - Redundancy
  - Protection of data from accidental changes
  - Maintainability

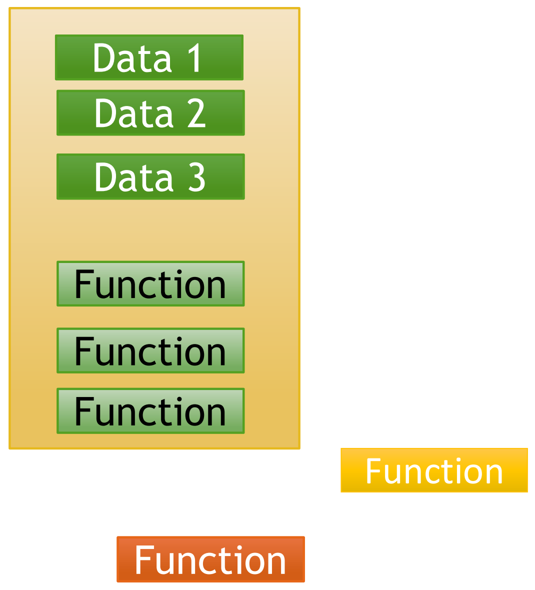



calc_tax
setmaritalstatus
getchildren
getbankbalance
getlicensedata
getcriminalrecord
Student Data
change_values
calc_average
calc_ranks
friendlist
CD
AD

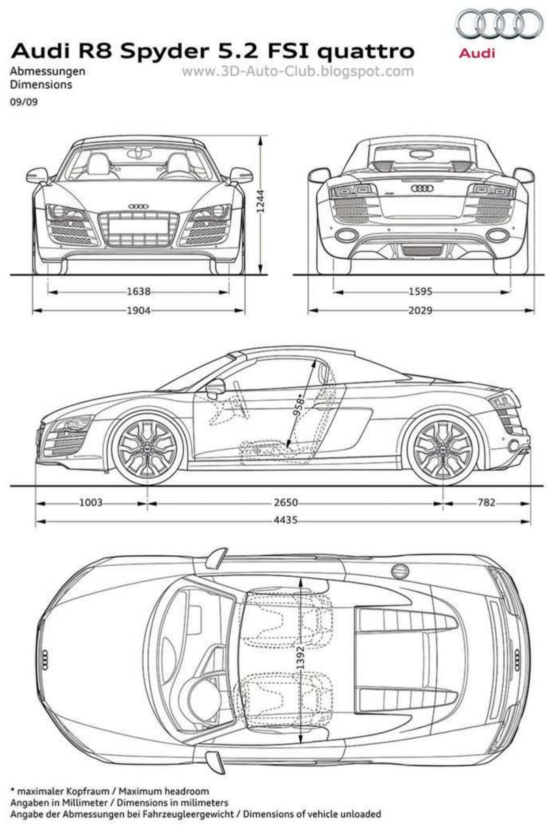
But why "object-oriented" ness in programming

- Combine/associate the data and relevant functions in a single entity



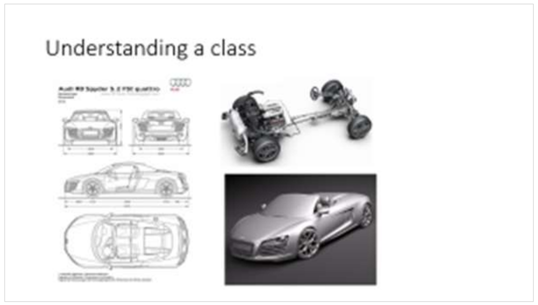


Understanding a class


Audi R8 Spyder 5.2 FSI quattro
Audi
Abmessungen
Dimensions
09/09
www.3D-Auto-Club.blogspot.com
1244
1638
1904
1595
2029
958*
1003
2650
782
4435
1392
* maximaler Kopfraum / Maximum headroom
Angaben in Millimeter / Dimensions in milimeters
Angabe der Abmessungen bei Fahrzeugleergewicht / Dimensions of vehicle unloaded

Object

- An object is a instance of a class and its an entity which has its own set of attributes and functions that were defined by a class



Understanding a class

Public Interface

- The set of all methods provided by a class, together with the description of their behavior is called the public interface of the class

Encapsulation

- Encapsulation is the act of providing a public interface and hiding the implementation details
- Encapsulation enables changes in the implementation without affecting users of the class

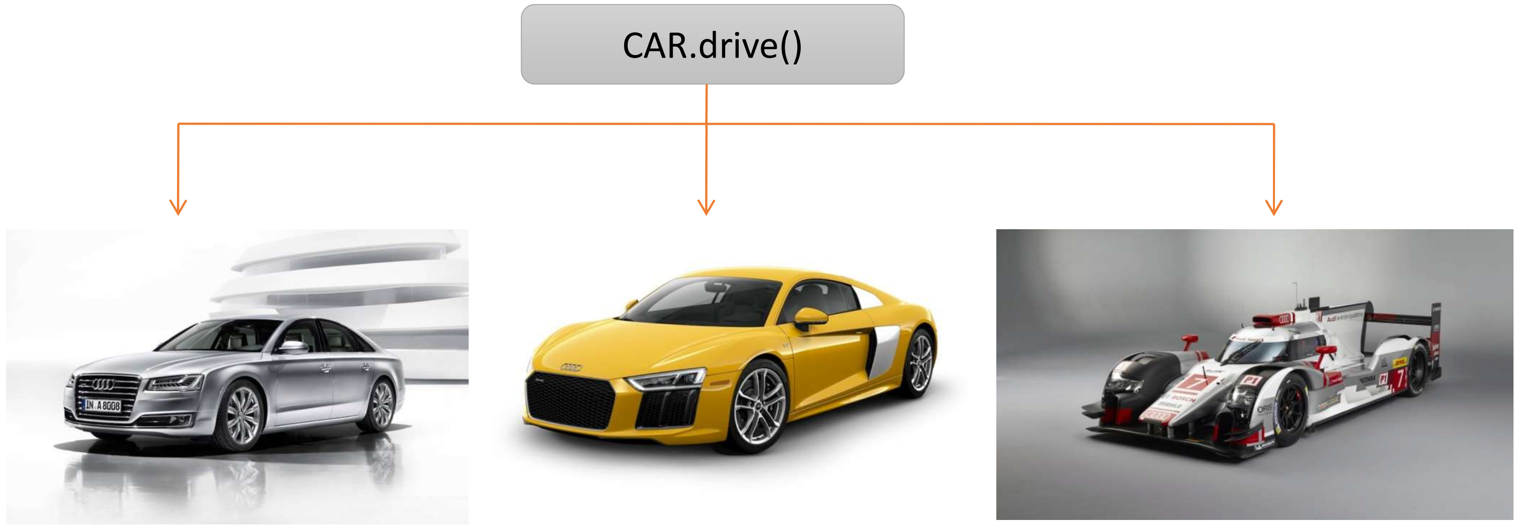
You can drive a car by operating the steering wheel and pedals, without knowing how the engine works. Similarly, you use an object through its methods. The implementation is hidden.

Inheritance

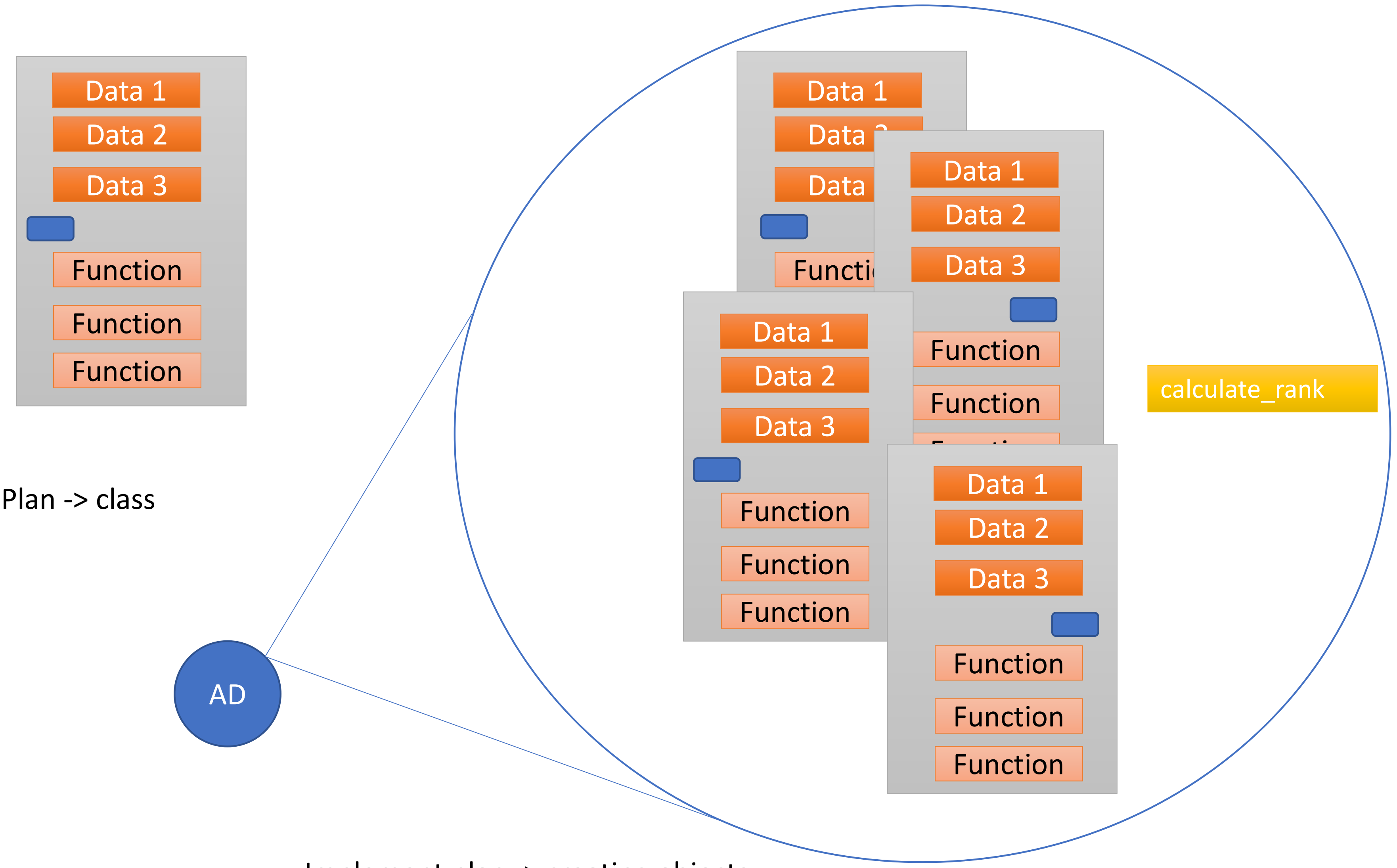
- Inheritance is a way to form new classes and thereafter objects using classes that have already been defined.

Polymorphism

- Polymorphism means that meaning of operation depends on the object being operated on.

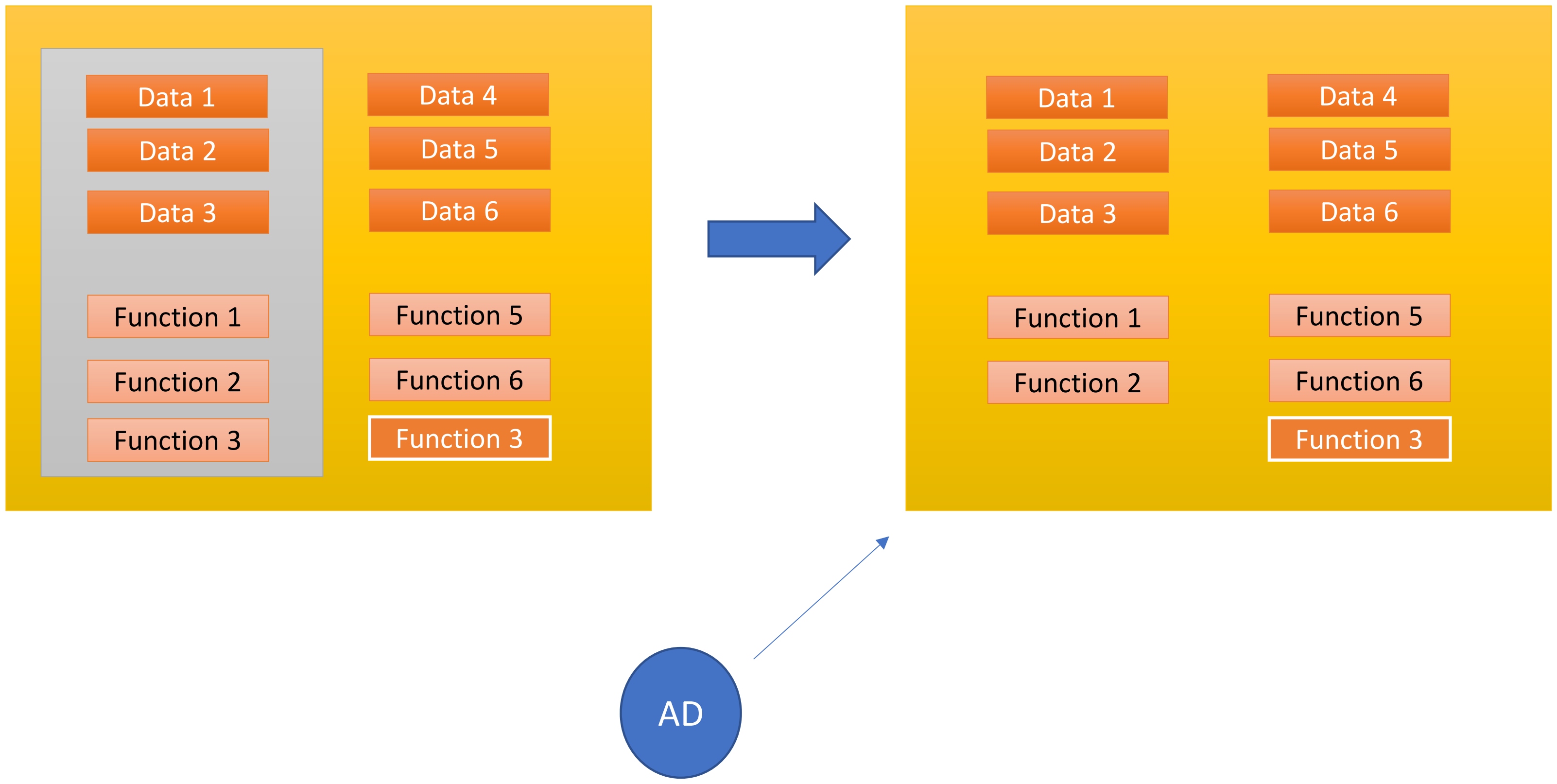


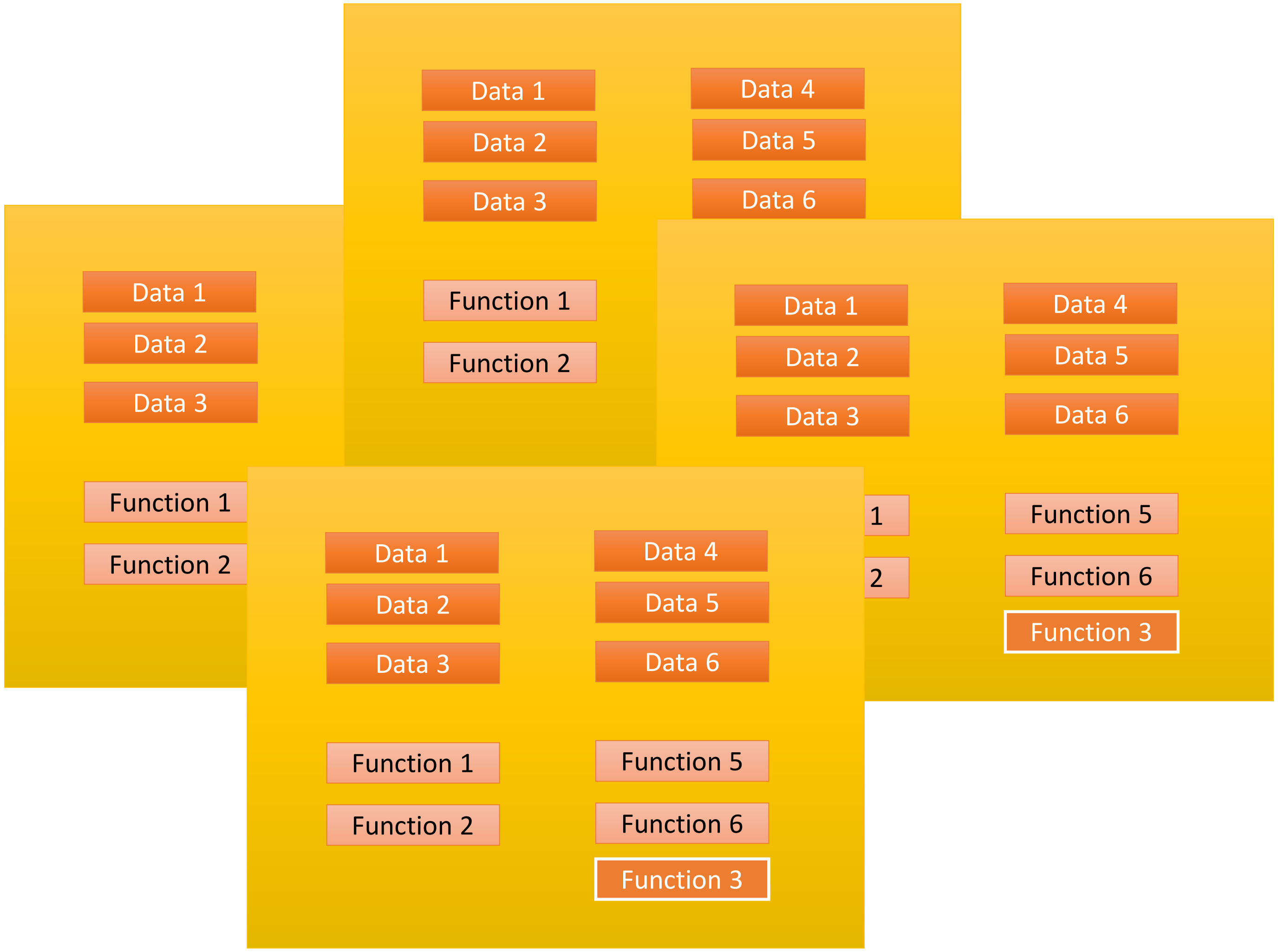
Let’s discuss about driving and maintenance tasks for cars…

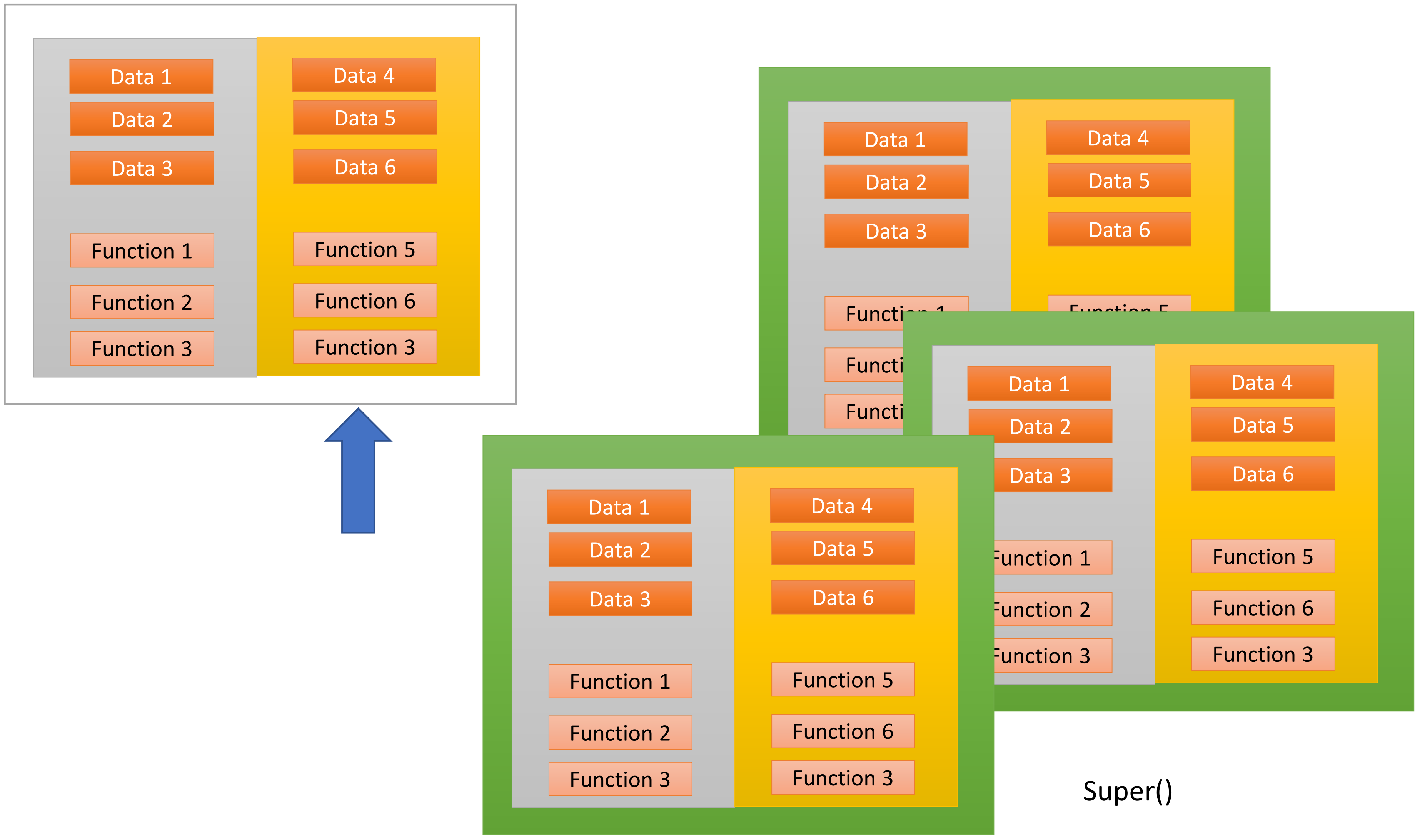


Plan -> class

Implement plan -> creating objects


Data 1
Data 2
Data 3
Function 1
Function 2
Function 3
Data 4
Data 5
Data 6
Function 5
Function 6
Function 3
Data 1
Data 2
Data 3
Function 1
Function 2
Data 4
Data 5
Data 6
Function 5
Function 6
Function 3
AD


Data 1
Data 2
Data 3
Data 4
Data 5
Data 6
Function 1
Function 2
Data 1
Data 2
Data 3
Function 1
Function 2
Data 1
Data 2
Data 3
Data 4
Data 5
Data 6
1
2
Function 5
Function 6
Function 3
Data 1
Data 2
Data 3
Data 4
Data 5
Data 6
Function 1
Function 2
Function 5
Function 6
Function 3


Data 1
Data 2
Data 3
Function 1
Function 2
Function 3
Data 4
Data 5
Data 6
Function 5
Function 6
Function 3

Super()

Classes

- Starting with ECMAScript 2015, also known as ECMAScript 6, JavaScript programmers can build their applications using this object-oriented class-based approach.
- In TypeScript, developers can use these techniques now, and compile them down to JavaScript that works across all major browsers and platforms, without having to wait for the next version of JavaScript.
- Classes wrap properties and functions together such that the there is a strong relationship between the properties and functions.
- Classes are templates from which objects are derived, objects are supposed to exhibit a specific type of behavior according to a given context

Creating the First Class

```
class Greeter {
    greeting: string;

    constructor(message: string) {
      this.greeting = message;
    }

    greet() {
      return "Hello, " + this.greeting;
    }
  }

let greeter = new Greeter("world");
```

attribute

constructor

member function

Creative Lab #3.1

- create a class called student
- properties: name, age, class, phy, chem, math, bio, avg, rank;
- funcitons: showReport, calcAverage
- Create a HTML form to receive the information for a given student
- Accept the data in to a student object and add it into a classroom Array
- (classRoom array is the array that contains student objects)
- Add button to calculate average of all subjects
- Add button to showReport

‘public’ Access Modifiers

- In TypeScript, each member is public by default. You may still mark a member public explicitly.

```
class Animal {
    public name: string;

    public constructor(theName: string) {
      this.name = theName;
    }

    public move(distanceInMeters: number) {
      console.log(`${this.name} moved ${distanceInMeters}m.`);
    }
  }
```

‘private’ Access Modifier

- TypeScript also has its own way to declare a member as being marked private, it cannot be accessed from outside of its containing class.

```
class Animal {
    private name: string;

    constructor(theName: string) {
      this.name = theName;
    }
  }

// Doesn’t work
new Animal("Cat").name;
```

```
// TS 3.8
class Animal {
    #name: string;
    constructor(theName: string) {
      this.#name = theName;
    }
  }

  new Animal("Cat").#name;
```

‘protected’ Access Modifier

- The protected modifier acts much like the private modifier with the exception that members declared protected can also be accessed within deriving classes.

```
class Person {
    protected name: string;
    constructor(name: string) {
      this.name = name;
    }
  }
```

```
class Employee extends Person {
    private department: string;

    constructor(name: string, department: string) {
      super(name);
      this.department = department;
    }

    public getElevatorPitch() {
      return `Hello, my name is ${this.name}
              and I work in ${this.department}.`;
    }
  }
```

```
let howard = new Employee("Howard", "Sales");
console.log(howard.getElevatorPitch());
console.log(howard.name);
```

Doesn’t work

‘readonly’ Properties

- You can make properties readonly by using the readonly keyword. Readonly properties **must be initialized at their declaration** or in the **constructor**.

```
class Octopus {
    readonly name: string;
    readonly numberOfLegs: number = 8;

    constructor(theName: string) {
      this.name = theName;
    }
  }

  let dad = new Octopus("Man with the 8 strong legs");
  dad.name = "Man with the 3-piece suit";
```

Doesn’t work

Accessors

- TypeScript supports getters/setters as a way of intercepting accesses to a member of an object. This gives you a way of having finer-grained control over how a member is accessed on each object.

```
const fullNameMaxLength = 10;

class Employee {
  private _fullName: string = "";

  get fullName(): string {
    return this._fullName;
  }

  set fullName(newName: string) {
    if (newName && newName.length > fullNameMaxLength) {
      throw new Error("fullName has a max length of " + fullNameMaxLength);
    }
    this._fullName = newName;
  }
}
```

```
let employee = new Employee();
employee.fullName = "Bob Smith";

if (employee.fullName) {
  console.log(employee.fullName);
}
```

Static Properties

- We can also create *static* members of a class, those that are visible on the class itself rather than on the instances.

```
class Grid {
    static origin = { x: 0, y: 0 };

    calculateDistanceFromOrigin(point: { x: number; y: number }) {
      let xDist = point.x - Grid.origin.x;
      let yDist = point.y - Grid.origin.y;
      return Math.sqrt(xDist * xDist + yDist * yDist) / this.scale;
    }

    constructor(public scale: number) {}
  }
```

```
let grid1 = new Grid(1.0); // 1x scale
let grid2 = new Grid(5.0); // 5x scale

console.log(grid1.calculateDistanceFromOrigin({ x: 10, y: 10 }));
console.log(grid2.calculateDistanceFromOrigin({ x: 10, y: 10 }));
```

Inheritance: Extending a Class

- In TypeScript, we can use common object-oriented patterns.
- One of the most fundamental patterns in class-based programming is being able to extend existing classes to create new ones using inheritance.

```
// Base Class
class Animal {
    name: string;
    constructor(theName: string) {
      this.name = theName;
    }
    move(distanceInMeters: number = 0) {
      console.log(`${this.name} moved ${distanceInMeters}m.`);
    }
  }
```

Inheritance: Extending a Class

```
class Snake extends Animal {
    constructor(name: string) {
      super(name);
    }
    move(distanceInMeters = 5) {
      console.log("Slithering...");
      super.move(distanceInMeters);
    }
  }

  class Horse extends Animal {
    constructor(name: string) {
      super(name);
    }
    move(distanceInMeters = 45) {
      console.log("Galloping...");
      super.move(distanceInMeters);
    }
  }
```

```
let sam = new Snake("Sammy,Python");
let tom: Animal = new Horse("Tommy,Palomino");

sam.move();
tom.move(34);
```

Abstract Classes

- Abstract classes are base classes from which other classes may be derived. They may not be instantiated directly.

```
abstract class Department {
    constructor(public name: string) {}

    printName(): void {
      console.log("Department name: " + this.name);
    }

    abstract printMeeting(): void; // must be implemented in derived classes
  }
```

```
class AccountingDepartment extends Department {
    constructor() {
      super("Accounting and Auditing");
      // constructors in derived classes must call super()
    }

    printMeeting(): void {
      console.log("The Accounting Department meets each Monday at 10am.");
    }

    generateReports(): void {
      console.log("Generating accounting reports...");
    }
  }
```

```
let department: Department; // ok to create a reference to an abstract type
department = new Department(); // error: cannot create an instance of an abstract class

department = new AccountingDepartment(); // ok to create and assign a non-abstract subclass
department.printName();
department.printMeeting();
department.generateReports();
// error: department is not of type AccountingDepartment, cannot access generateReports
```

Creative Lab #3.2

- Extend the student class
- Add properties: DoB, hobbies:string[]
- Override function printReport()
- setHobby(), delHobby()
- Write some simple code to test it
- Test backward compatibility

Interface

- One of TypeScript’s core principles is that type checking focuses on the shape that values have. This is sometimes called “duck typing” or “structural subtyping”.
- In TypeScript, interfaces fill the role of naming these types, and are a powerful way of defining contracts within your code as well as contracts with code outside of your project.
- Interfaces are a way to name a type so you don’t have to define it inline.
- Type aliases and interfaces are mostly two syntaxes for the same thing

```
type Sushi = {
    calories: number
    salty: boolean
    tasty: boolean
    }
```

→

```
interface Sushi {
    calories: number
    salty: boolean
    tasty: boolean
 }
```

```
type Food = {
    calories: number
    tasty: boolean
    }
type Sushi = Food & {
    salty: boolean
    }
type Cake = Food & {
    sweet: boolean
}
```

→

```
interface Food {
   calories: number
   tasty: boolean
}
interface Sushi extends Food {
      salty: boolean
}
interface Cake extends Food {
      sweet: boolean
}
```

Using an Interface

```
interface LabeledValue {
    label: string;
}

function printLabel(labeledObj: LabeledValue) {
    console.log(labeledObj.label);
}

let myObj = { size: 10, label: "Size 10 Object" };
printLabel(myObj);
```

If the object we pass to the function meets the requirements listed, then it's allowed. It's worth pointing out that the type checker does not require that these properties come in any sort of order, only that the properties the interface requires are present and have the required type.

Using an Interface

```
interface SquareConfig {
  color?: string;
  width?: number;
}

function createSquare(config: SquareConfig): { color: string; area: number } {
  let newSquare = { color: "white", area: 100 };
  if (config.color) {
    newSquare.color = config.color;
  }
  if (config.width) {
    newSquare.area = config.width * config.width;
  }
  return newSquare;
}

let mySquare = createSquare({ color: "black" });
```

Optional Parameters

Using Interfaces with Classes

- When you declare a class, you can use the implements keyword to say that it satisfies a particular interface.

```
interface Animal {
    eat(food: string): void
    sleep(hours: number): void
    }

class Cat implements Animal {
    eat(food: string) {
        console.info('Ate some', food, '. Mmm!')
    }
    sleep(hours: number) {
        console.info('Slept for', hours, 'hours')
    }
}
```

Readonly Interface Properties

- Some properties should only be modifiable when an object is first created. You can specify this by putting readonly before the name of the property

```
interface Point {
    readonly x: number;
    readonly y: number;
 }

 let p1: Point = { x: 10, y: 20 };
 p1.x = 5; // error!
```

Creative Lab #3.3

- Create a interface in for the student class written in Lab 3.1
- Use the interface in the constructor and make sure its working as before
- Extend the interface with new attributes as specified in Lab 3.2
- Use the interface in the constructor of extended student class and make sure its working as before

Generics

- Generics allow us to create a component that can work over a variety of types rather than a single one. This allows users to consume these components and use their own types.

```
function identity(arg: any): any {
    return arg;
  }
```

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

Type allows us to capture the type the user provides (e.g. number), so that we can use that information later. Here, we use Type again as the return type. On inspection, we can now see the same type is used for the argument and the return type. This allows us to traffic that type information in one side of the function and out the other.

Modules

- Module is a collection of features that can be used with export/import syntax

```
// @filename: maths.ts
export var pi = 3.14;
export let squareTwo = 1.41;
export const phi = 1.61;

export class RandomNumberGenerator {}

export function absolute(num: number) {
  if (num < 0) return num * -1;
  return num;
}
```

```
import { pi, phi, absolute } from "./maths.js";

console.log(pi);
const absPhi = absolute(phi);
```

CommonJS

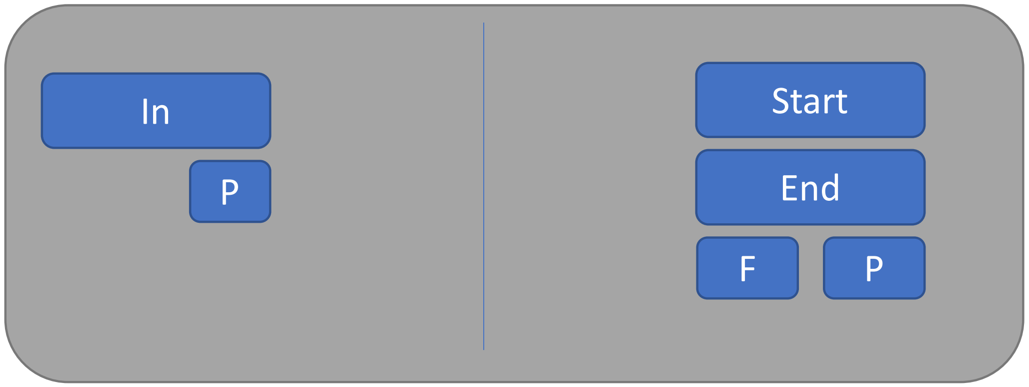
- CommonJS is the format which most modules on **npm** are delivered in. Even if you are writing using the ES Modules syntax above, having a brief understanding of how CommonJS syntax works will help you debug easier.

```
function absolute(num: number) {
    if (num < 0) return num * -1;
    return num;
  }

  module.exports = {
    pi: 3.14,
    squareTwo: 1.41,
    phi: 1.61,
    absolute,
  };
```

```
const maths = require("maths");
maths.pi;

const { squareTwo } = require("maths");
squareTwo;
```

Creative Lab #3.4

- Create a module called series
- primeCheck(), genPrimes(), genFibo() -> export them
- In another file -> main.ts import series
- From HTML



Third Party Libraries

- Method #1: Add it in you HTML file
- Method #2: Install and reference it
  - Just click this link https://github.com/typings/typings , install it via NPM, and find the according definition file.

```
### Install the Typings CLI package first.
npm install typings --global

### Search for the typeing file you need.
typings search jquery

### Install non-global typings.
typings install jquery --save
```

Third Party Libraries

- And in your code, you just need to add a reference at the beginning of the file:

```
/// <reference path="jquery/jquery.d.ts" />

$.ajax("URL")
```

Decorators

- A decorator is a pattern in programming in which you wrap something to change its behavior.
- It can be attached to class declaration, method, accessor, property or parameter.
- Decorators use the for @decorator
- In JavaScript, this feature is currently at stage two. It's not yet available in browsers or Node.js, but you can test it out by using compilers like Babel.
- To activate the feature, you'll need to make some adjustments to your tsconfig.json file

```
{
    "compilerOptions": {
      "target": "ES5",
      "experimentalDecorators": true
    }
  }
```

```typescript
function first() {
  console.log("first(): factory evaluated");
  return function (target: any, propertyKey: string, descriptor: PropertyDescriptor) {
    console.log("first(): called");
  };
}

function second() {
  console.log("second(): factory evaluated");
  return function (target: any, propertyKey: string, descriptor: PropertyDescriptor) {
    console.log("second(): called");
  };
}

class ExampleClass {
  @first()
  @second()
  method() {}
}
```

Compiler Settings

- https://www.typescriptlang.org/tsconfig