TypeScript

Peter Kassenaar – info@kassenaar.com

Peter Kassenaar

Over Peter Kassenaar:

- Trainer, auteur, developer – sinds 1996
- Specialisme: *“Everything JavaScript”*
- JavaScript, ES6, AngularJS, NodeJS, jQuery, PhoneGap, TypeScript

www.kassenaar.com/blog

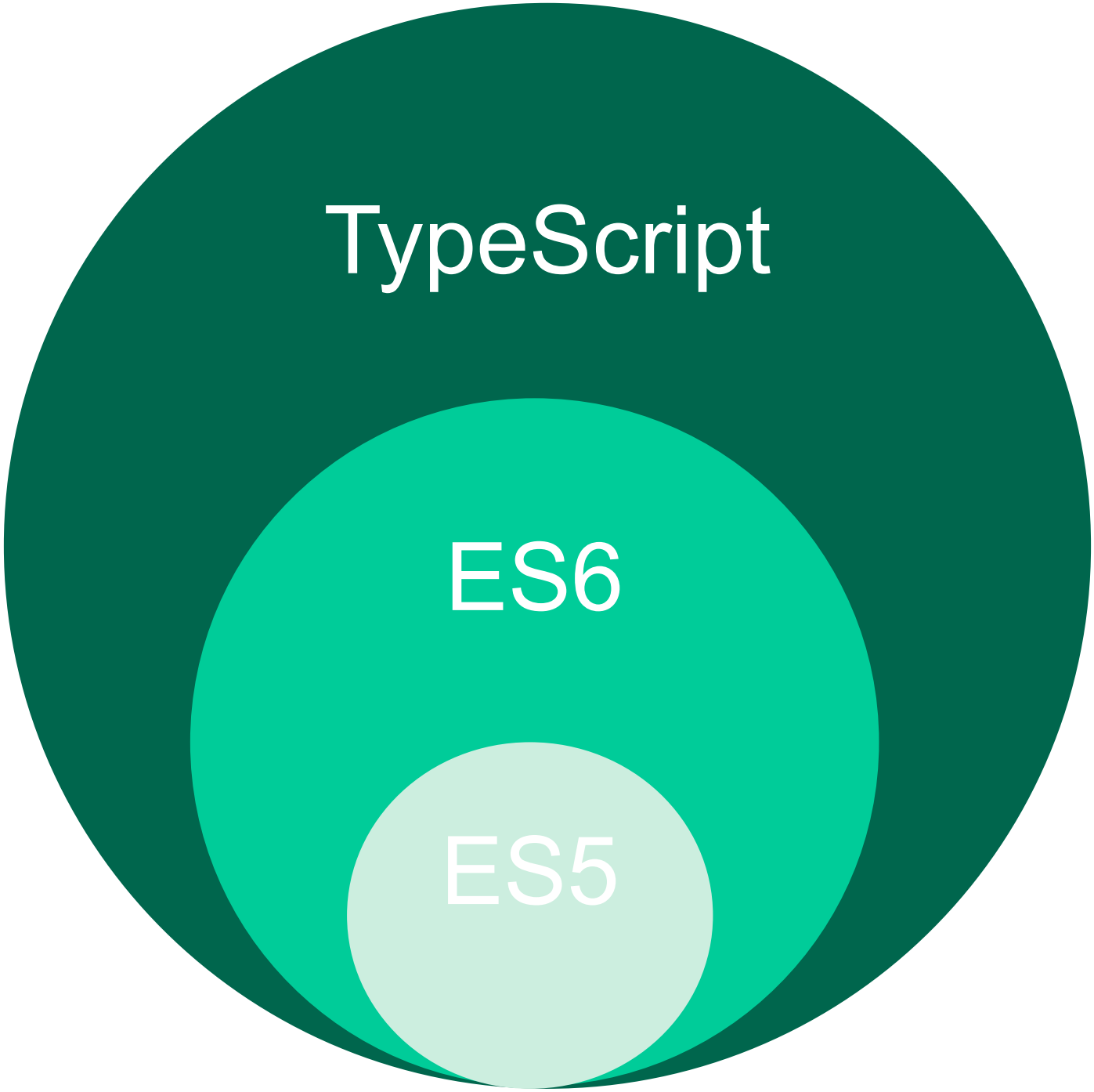
info@kassenaar.com

Twitter: @PeterKassenaar

What wrong with JavaScript?

Maar we willen…

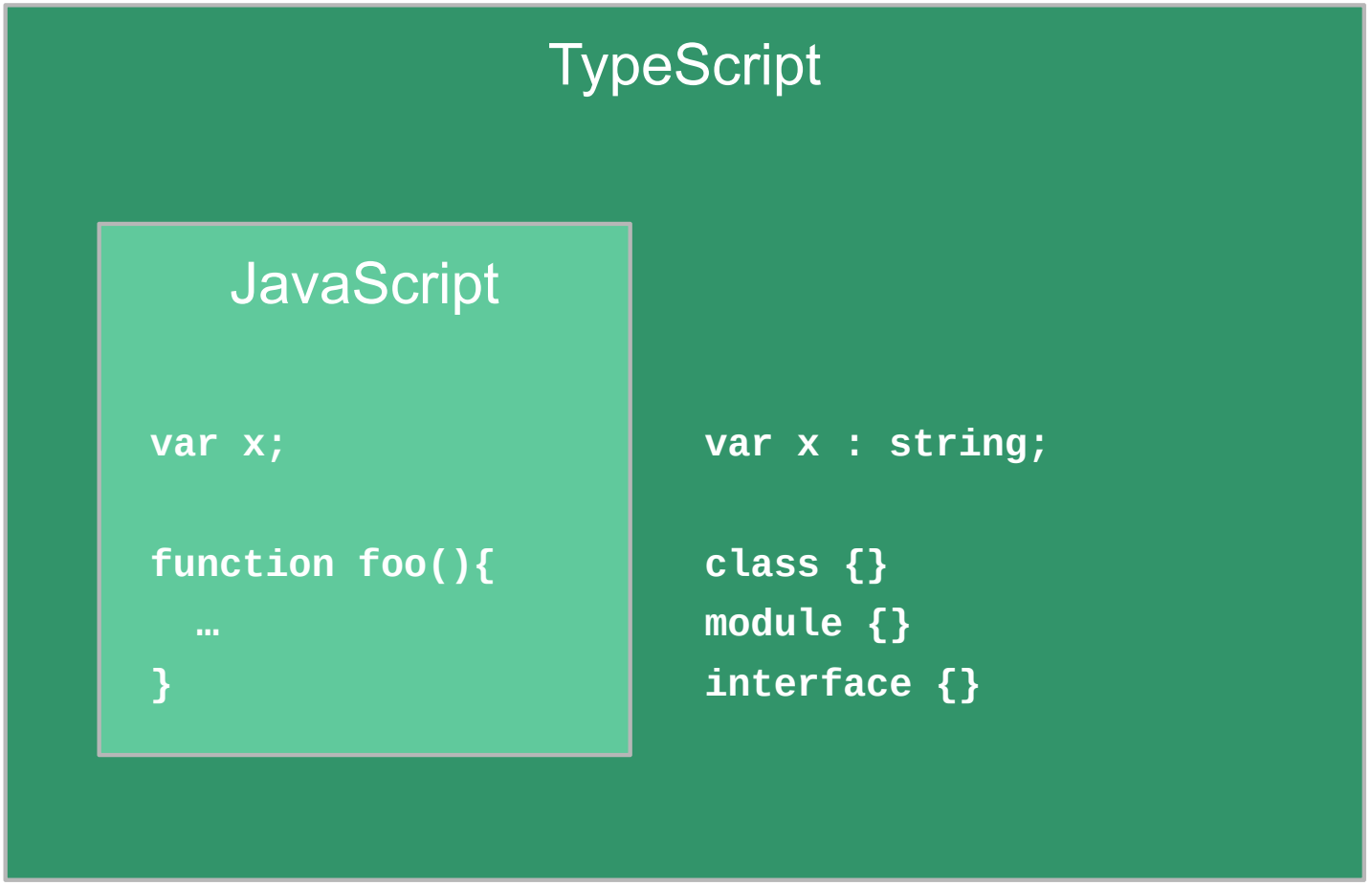
Programmeertalen


TypeScript
ES6
ES5

What is TypeScript?

- **Superset van JavaScript**
- **Compileert naar plain JavaScript**
  - “Transpiling”
- **Strongly typed**
- **Class-based object-orientation**
  - ‘echte’ OO binnen bereik van JavaScript

“Superset van JavaScript”

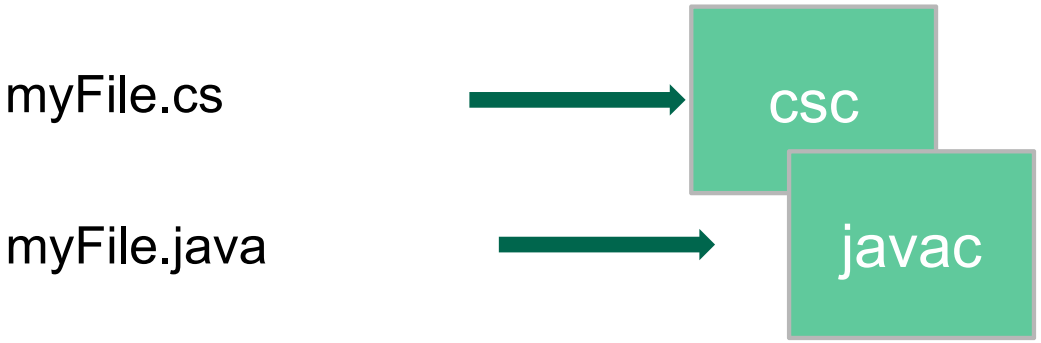


Snel TypeScript maken?

*Hernoem alle *.js-bestanden naar *.ts-bestanden.*

Voila

TypeScript *transpiles* to JavaScript




```
var x : string;

class productController {}
```

```
var x;

function productController() {}
```

Waarom dan TypeScript?

Waarom dan al die moeite doen? Kun je net zo goed direct JavaScript schrijven

Because dev

Voordelen voor developers

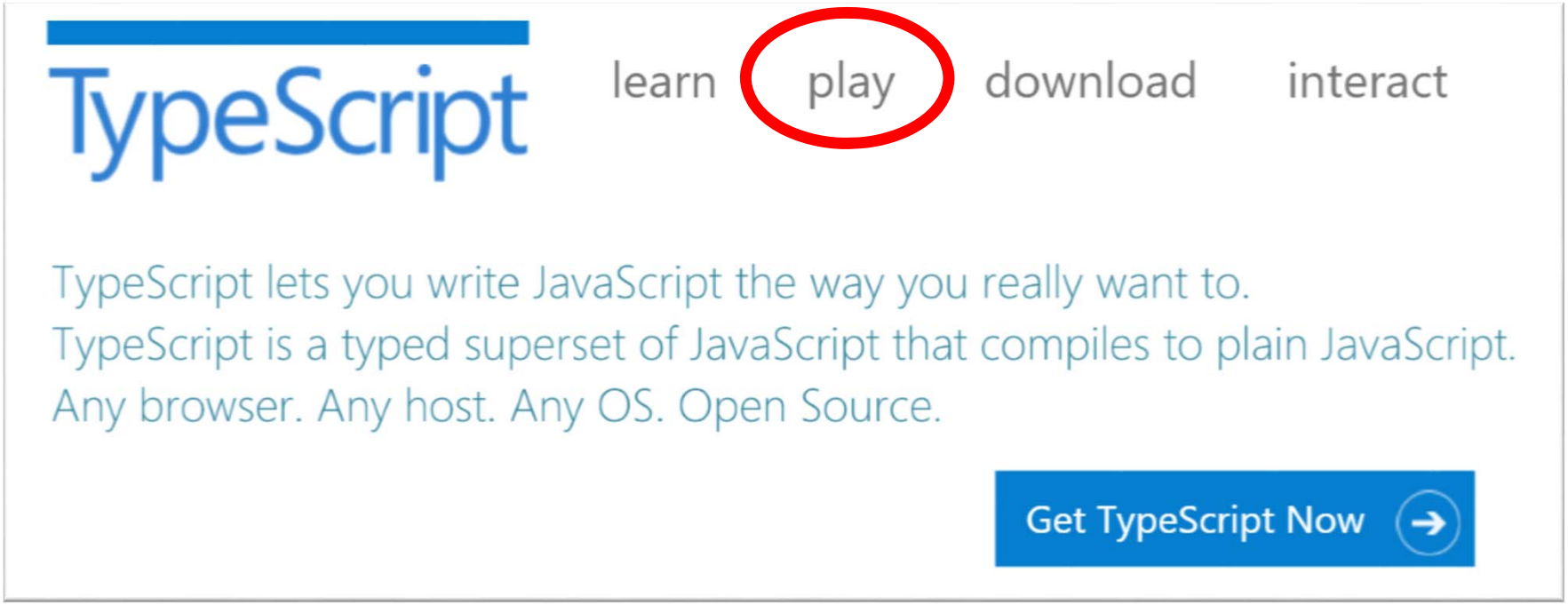
1. Build better code

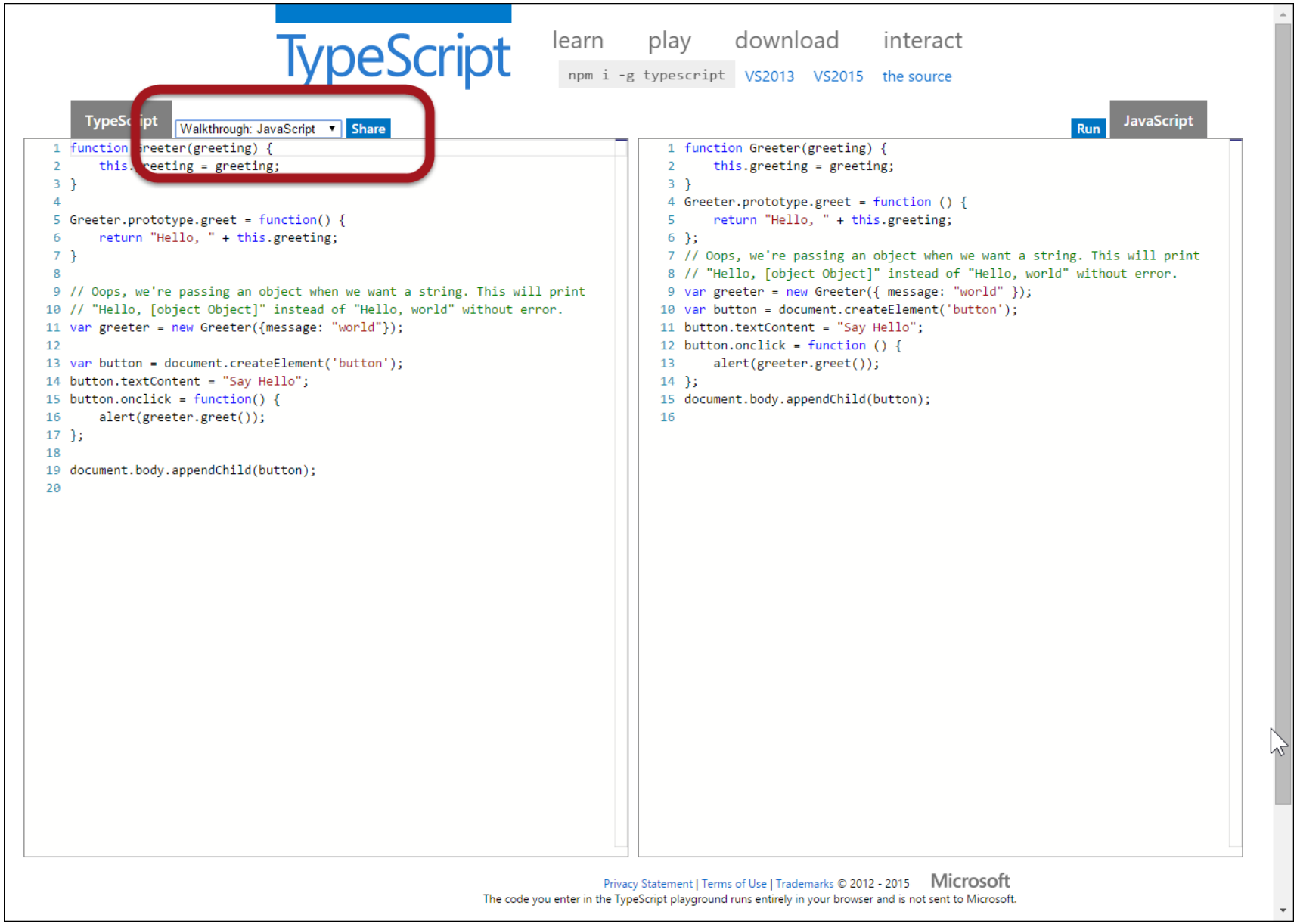
2. Safely refactor code

3. Reason about code

Uiteindelijk: *voordeel voor de gebruiker*

typescriptlang.org




TypeScript
learn play download interact
npm i -g typescript VS2013 VS2015 the source
TypeScript
Walkthrough: JavaScript
Share
Run
JavaScript
function Greeter(greeting) {
this.greeting = greeting;
}
Greeter.prototype.greet = function() {
return "Hello, " + this.greeting;
}
// Oops, we're passing an object when we want a string. This will print
// "Hello, [object Object]" instead of "Hello, world" without error.
var greeter = new Greeter({message: "world"});
var button = document.createElement('button');
button.textContent = "Say Hello";
button.onclick = function() {
alert(greeter.greet());
};
document.body.appendChild(button);
function Greeter(greeting) {
this.greeting = greeting;
}
Greeter.prototype.greet = function () {
return "Hello, " + this.greeting;
};
// Oops, we're passing an object when we want a string. This will print
// "Hello, [object Object]" instead of "Hello, world" without error.
var greeter = new Greeter({ message: "world" });
var button = document.createElement('button');
button.textContent = "Say Hello";
button.onclick = function () {
alert(greeter.greet());
};
document.body.appendChild(button);
Privacy Statement | Terms of Use | Trademarks © 2012 - 2015 Microsoft
The code you enter in the TypeScript playground runs entirely in your browser and is not sent to Microsoft.

Strongly typing

JavaScript:

```
var myVar;
myVar = 42;
myVar = '42';
myVar = {
    firstName: 'Peter',
    lastName : 'Kassenaar'
}
console.log(myVar);
```

TypeScript:

```
var myVar1:number;
var myVar2:string;
var myVar3:Array<string>;
var myVar4:Object;
myVar4 = {
    firstName: 'Peter',
    lastName : 'Kassenaar'
};
console.log(myVar4);
```

Any and Void

any - **variable can literally be of 'any' type**

This is the default value, if not provided

```
// any
var iDontKnow:any;
```

```
// void - no return value
function warnUser(): void {
   console.log ("This is a warning in the console");
}
warnUser();
```

ns that don't

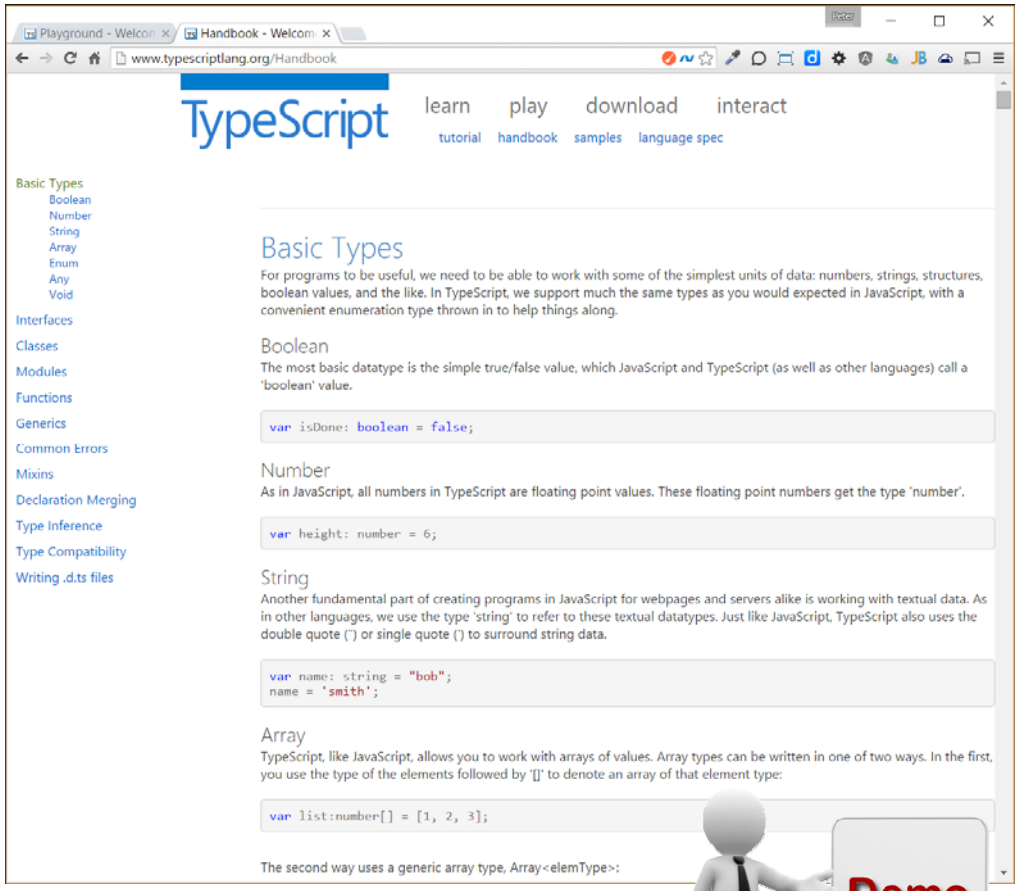
Strongly typing

- Nogmaals: strongly typing helpt de developer.
- In de JavaScript-code blijft er niets van over.
- Doel : in de broncode aangeven wat je bedoeling is.
  - Compile time checking
  - Mede-developers helpen
  - Tooling support

TypeScript basic types

- Primitive types – 6 hoofdtypen
  - boolean
  - number
  - string
  - array
  - object
  - enum

    --
  - any
  - void



http://www.typescriptlang.org/Handbook

TypeScript Playground

Class-based OOP

JavaScript

```
function Customer(customerId){
   this.customerId = customerId;
}
```

TypeScript

```
function Customer(customerId){
   this.customerId = customerId;
}
```

of:

```
class Customer {
   customerId:number;

   constructor(customerId:number) {
      this.customerId = customerId;
   }
}
```

Checkpoint

TypeScript is a Superset of JavaScript

Strong Typing

ES6 Functionality

Simplify Application Maintenance

Catch Issues Early

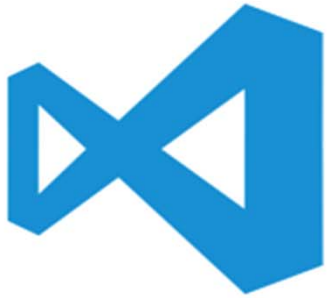
TypeScript – tooling support

Types, Autocompletion.

Compile-time checking in editors.

Alles is optioneel. Je kunt altijd nog gewoon JavaScript gebruiken.

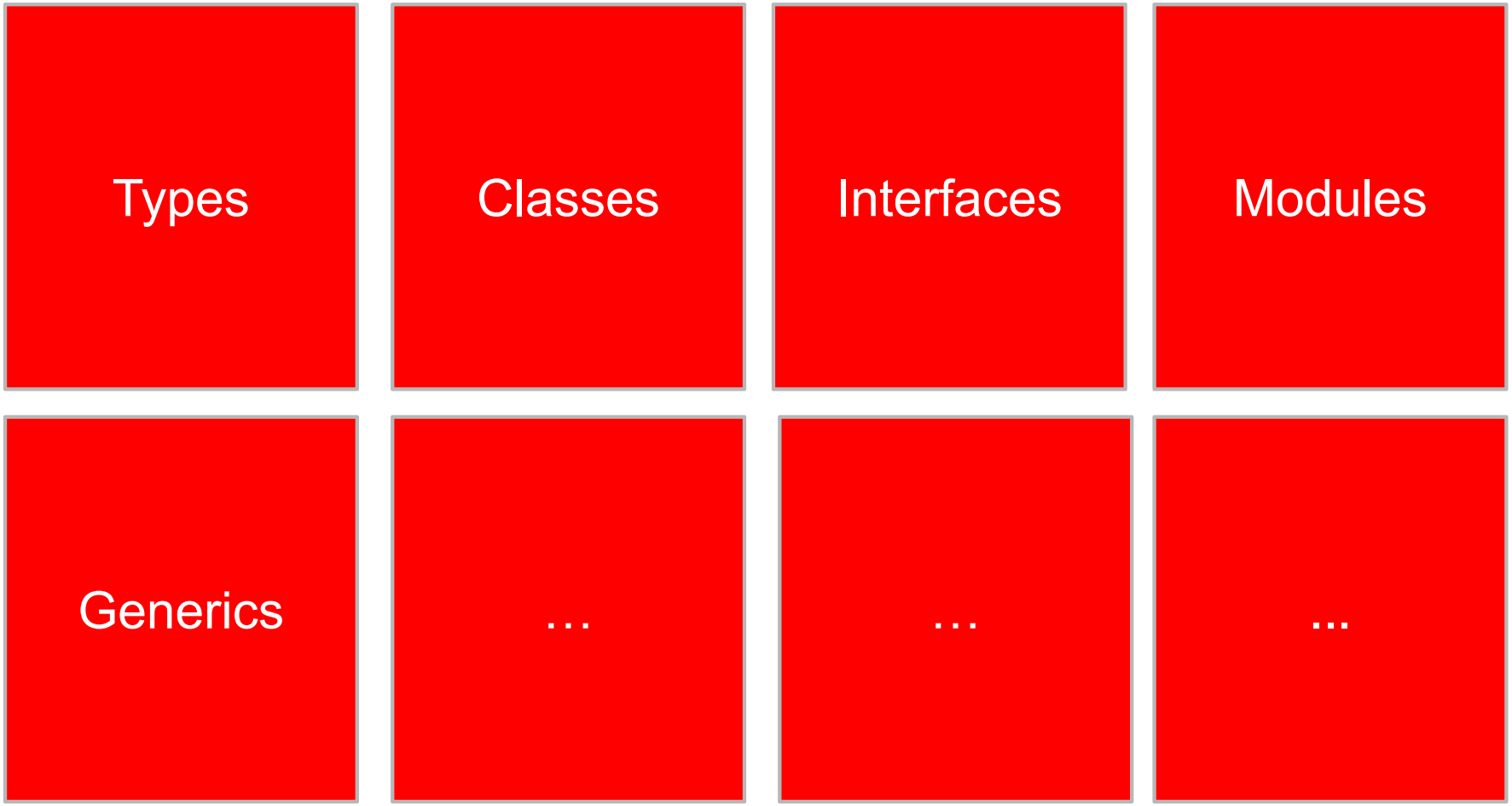
Editors – native TypeScript support



Visual Studio®

WebStorm

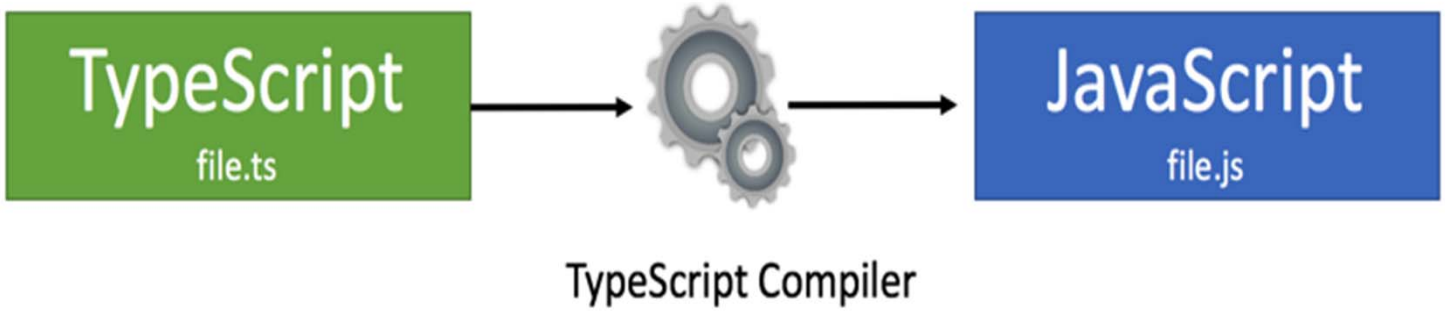
Belangrijke TypeScript Features



TypeScript Compilation

Zowel TypeScript als ES 6: compilatie nodig (*transpiling*).

Huidige generatie browsers begrijpt ES6 en TypeScript niet.



TypeScript Compiler

Installeren via Node.js

- globaal of lokaal

Genereert ook .map-files voor makkelijk debuggen.

Installing TypeScript

```
npm install typescript --save-dev
```

Compiling

via editor (WebStorm, Visual Studio [Code])

Via package.json `scripts { ... }` section

```
"scripts"         : {
   "tsc"  : "tsc --project src --watch --sourceMap --outDir src/js",
 },
```

Hello World Demo

```
class HelloWorld {
   msg:string;

   constructor(msg:string) {
      this.msg = msg;
   }

   sayHello() {
      return this.msg;
   }
}

var hello = new HelloWorld('Hello World');
console.log(hello.sayHello());
```

Defining Typed Variables

- **Type inference**
  - Bepaal type aan de hand van declaratie : implicit en explicit

```
// Type inference in Action
// TypeScript infers the variable num and set its type to number.
var num1 = 2 ;
console.log('num is: ', num1);

// explicitly set the type of variable num 2
var num2 : number = 2 ;

// …
```

Possible Typed Variables

```
var variableName: typeScriptType = value;
```

```
var age: number = 5;
var name: string = 'Anders';
var isLoaded: boolean = false;
var pets: string[] = ['Fido', 'Lassie','Rover'];
```

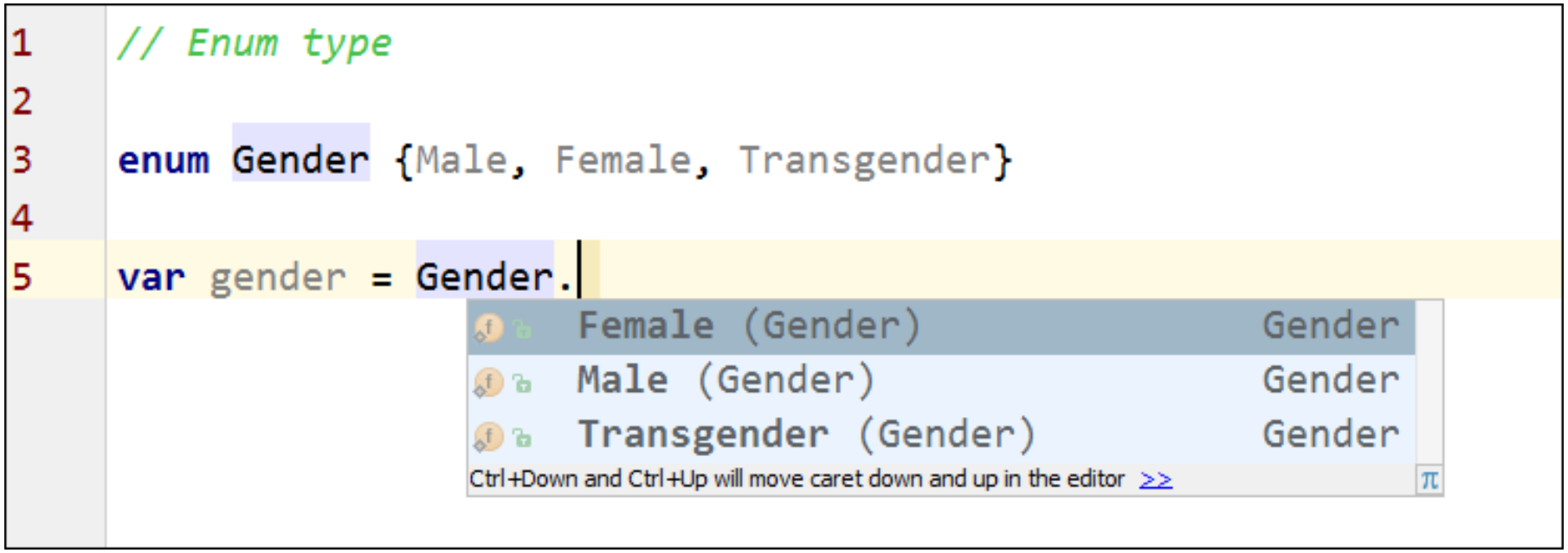
Array of strings

Typed Parameters

```
//Assigning a type to function parameters
function add(msg: string, x: number, y: number) {
    console.log(msg + (x + y));
}

add('Total = ', 3, 2);
```

Enum type

“Enum is a way to give more friendly names to sets of numeric values”



http://www.typescriptlang.org/Handbook#basic-types-enum

Const enum

```
const enum Gender { Male, Female };


var gender = Gender.Female;
```

Compiles hardcoded to:
var gender = 1;

Checkpoint

TypeScript supports strongly-typed variables

Parameter types can be assigned a type

Union types can minimize the number of function overloads

Const Enums reduce the amount of generated code

TypeScript functions

Verschillende soorten functies. Hoe ver wil je gaan met declareren...

TypeScript functions

- Functions can be defined in several ways:
  - Named functions
  - Anonymous functions
  - Lambda functions
  - Class functions

Named functions and Anonymous functions

```
// 1. Named functions in plain ES5
function add5(x, y){
    return x + y;
}

// 2. Anonymous function in plain ES5
var add6 = function(x, y){
    return x + y;
};

// no problem
console.log(add5(10, 20));
console.log(add6(30, 40));
```

Adding type annotation to function definition

```
// 3. Adding type to function definition
function add7(x:number, y:number):number {
    return x + y;
}
console.log(add7(50, 60));
```

This makes sense, IMO

Adding more type annotation to function definition…

```
var add8:(x:number, y:number) => number =
      function (x:number, y:number):number {
         return x + y;
      };
console.log(add8(70, 80));
```


```
var add8:(x:number, y:number) => number =
      function (x:number, y:number):number {
         return x + y;
      };
console.log(add8(70, 80));
```

Hmmm…. It works, but it's kinda awkward to read…

Using lambdas =>

```
// 5. Lambda/arrow functions: write functions simpler
var addLambdaSimple = function (x:number, y:number):number {
   return x + y;
};
addLambdaSimple(10, 20);

// 6. Even simpler, by using the => syntax and no more 'function'
var addLambdaSimpler = (x:number, y:number) => x + y;

addLambdaSimpler(30, 20);
```

No more `function` keyword. Takes some time to getting used to, but is much shorter and (eventually) easy to read

Optional parameters

- Default: function parameters obligated:

```
function buildAddress(address1: string, address2: string, city:
string) {
    //all parameters must be passed
}
```

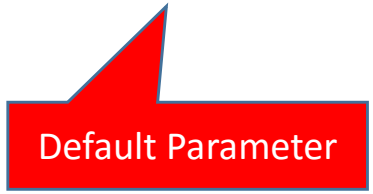
- Optional parameters: use question mark

```
function buildAddress(address1: string, city: string, address2?: string)
{
    //address2 parameter is optional
}

buildAddress('1234 Central', 'Seattle'); //address2 not passed
```

Default Parameters

Optional but provide a "default" value if the parameter isn't passed:

```
function buildAddressDefault(address1: string, city: string, address2 = 'N/A')
{
  //address2 parameter will default to N/A if not passed
}

buildAddress('1234 Central', 'Seattle'); //address2 not passed
```



Must be placed after all required parameters

Rest Parameters

Allows the "rest of the parameters" to be passed as an array using … syntax:


```
function buildAddress(city: string, ...restOfAddress: string[]){
    //city + an array of string parameters can be passed
}

//address & address2 are "rest" parameters
buildAddress(city, address, address2);
```

Must be placed after all required parameters

Lambdas and Using "this"

JavaScript's "`this`" keyword can be tricky to use

Changes context depending on the caller

"this" is start (a button)

```
this.start.addEventListener('click', this.updateTimer);
```

TypeScript lambdas capture "`this`"

"this" is captured

```
this.start.addEventListener('click',() =>
this.updateTimer());
```

`this` always refers to the "*current execution context*" (mostly a containing class)

Checkpoint

Functions can be defined multiple ways

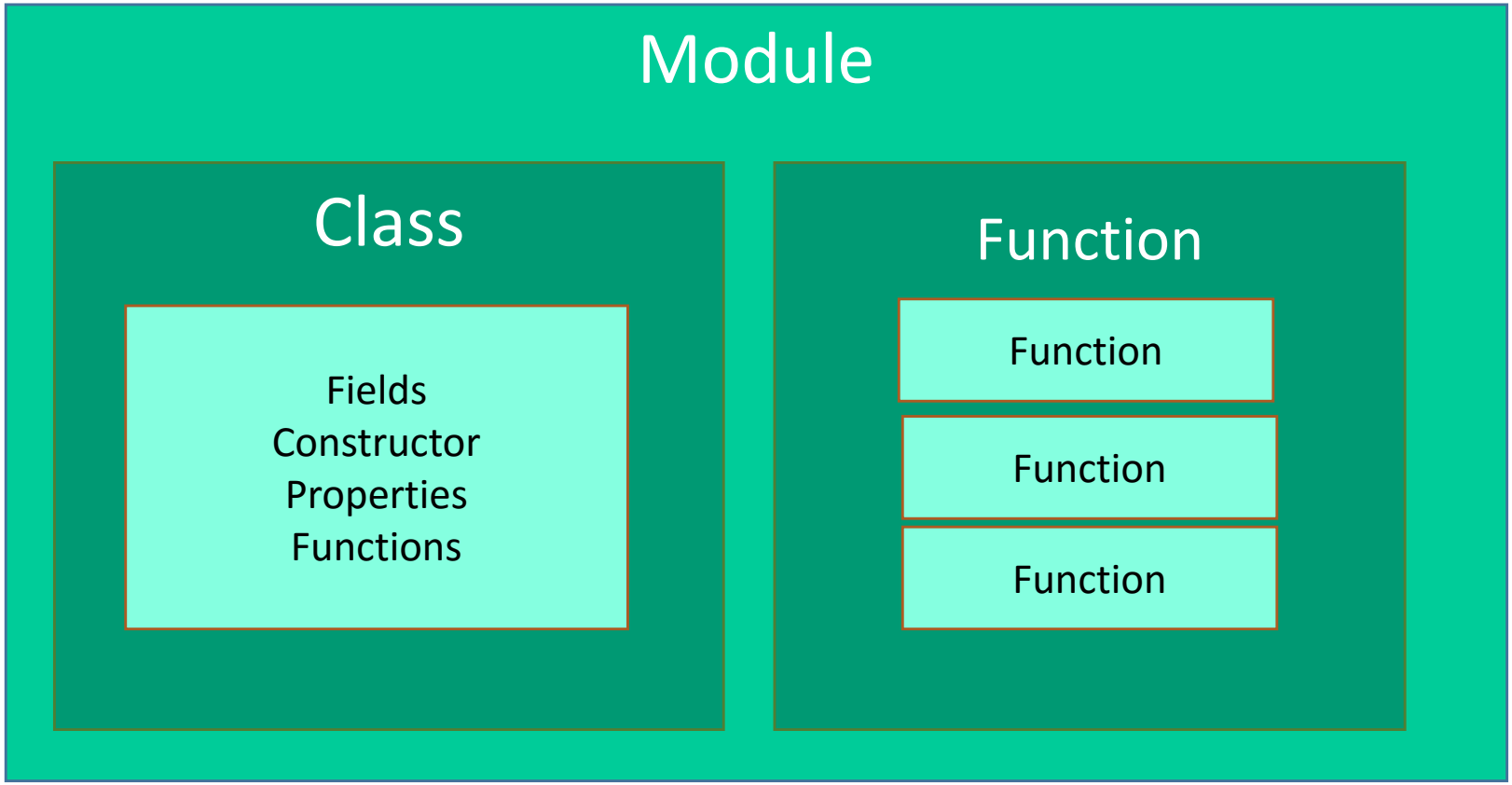
Parameters can be optional, default or rest

Lambdas provide short-cut functionality

Working with "this" can be simplified by using lambdas

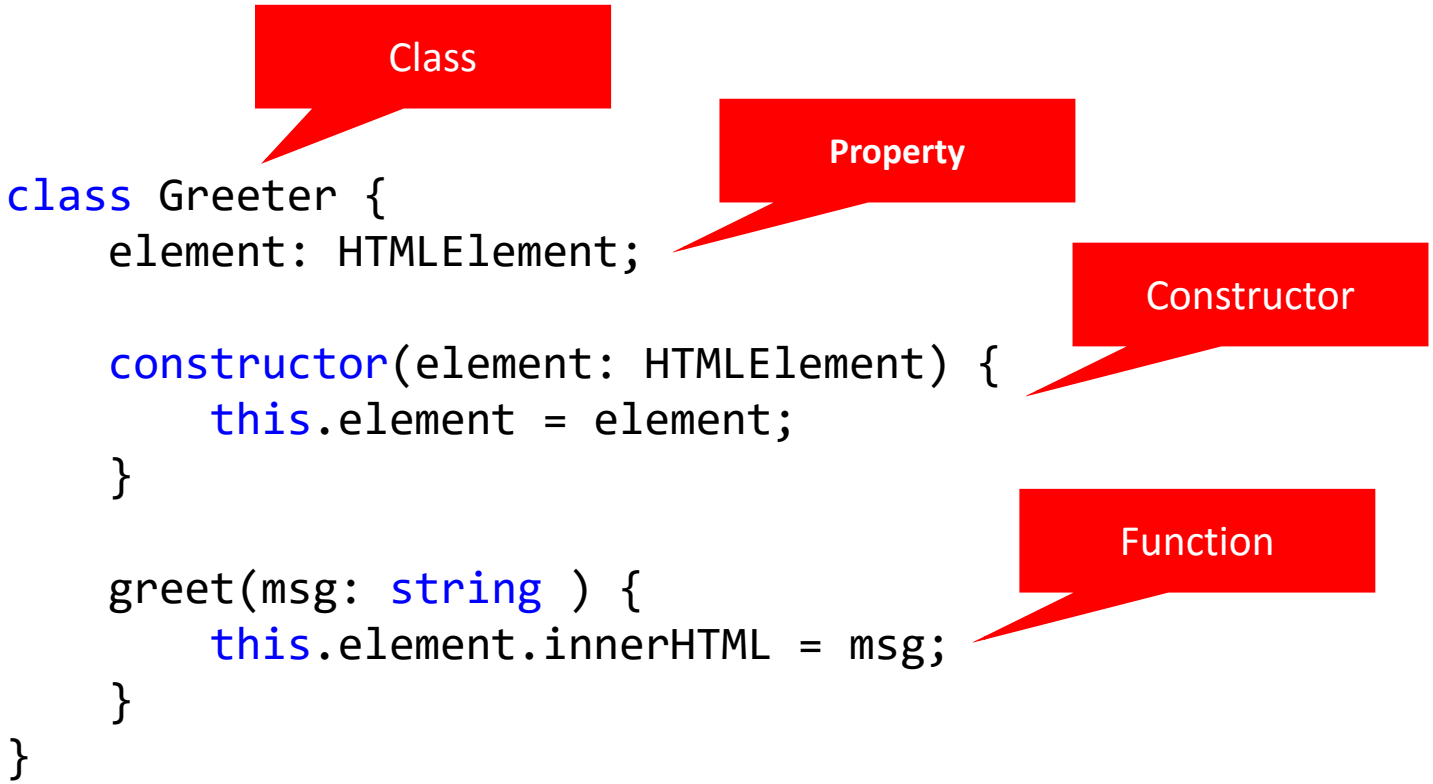
Classes

TypeScript Code Organization



- Classes *can be* organized in a `module` or `namespace` (optional)
- Keyword `function` not used inside classes. Perfectly valid outside classes

Class Example



Converting Classes to ES5 Compliant Code

TypeScript

```
class Greeter {
    element: HTMLElement;

    constructor(element: HTMLElement) {
        this.element = element;
    }

    greet(msg: string ) {
        this.element.innerHTML = msg;
    }
}
```

JavaScript

```
var Greeter = (function () {
    function Greeter(element) {
        this.element = element;
    }
    Greeter.prototype.greet = function (msg) {
        this.element.innerHTML = msg;
    };
    return Greeter;
})();
```

See http://typescriptlang.org for examples

The Constructor and Properties

```
class Greeter {
    element: HTMLElement;

    constructor(element: HTMLElement) {
        this.element = element;
    }

    greet(msg: string ) {
        this.element.innerHTML = msg;
    }
}

var greeter = new Greeter(el);
```

Constructor called when class is initialized

Stores parameter value in a **property**

Invoke Constructor

Auto-Generating Properties

This is usually faster to code, but slightly less easier to read

Because "private" is used the **property** will be auto-generated.

```
class Greeter {

    constructor(private element: HTMLElement) { }

    greet(msg: string ) {
        this.element.innerHTML = msg;
    }
}
```

Defining Properties

Defined using **`get`** and **`set`** keywords:

```
class Account {

    _balance: number = 0;

    get balance() {
        return this._balance;
    }

    set balance(val: number) {
        this._balance = val;
    }

}
```

Working with underscores: personal preference

Public and Private Modifiers

Class members are **`public`** by default:

```
class Account {
    _balance: number = 0;
}
```

Public by default

Members can be marked as **`private`**:

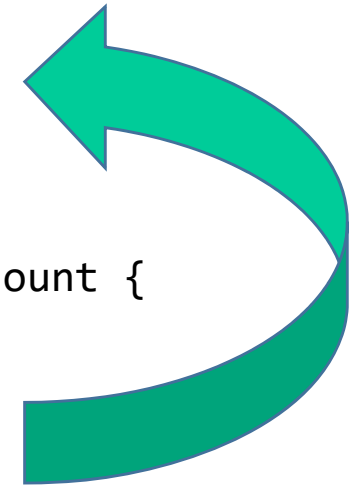
```
class Account {
    private _balance: number = 0;
}
```

Only accessible from within the class

Class Inheritance

Much easier than classic ES5 Prototypal inheritance:

```
class Account {
    private _title: string;
    constructor(title: string) {
        this._title = title;
    }
}

class CheckingAccount extends Account {
    constructor(title: string) {
        super(title);
    }
}
```



Remember though; you *have* to call `super` on the extended/child class

Checkpoint

Classes encapsulate members

Members include fields, constructors, properties, functions

TypeScript supports class extension

The super keyword can be used to call into a base class

Interfaces

An interface is a "code contract"

Drive Consistency across classes

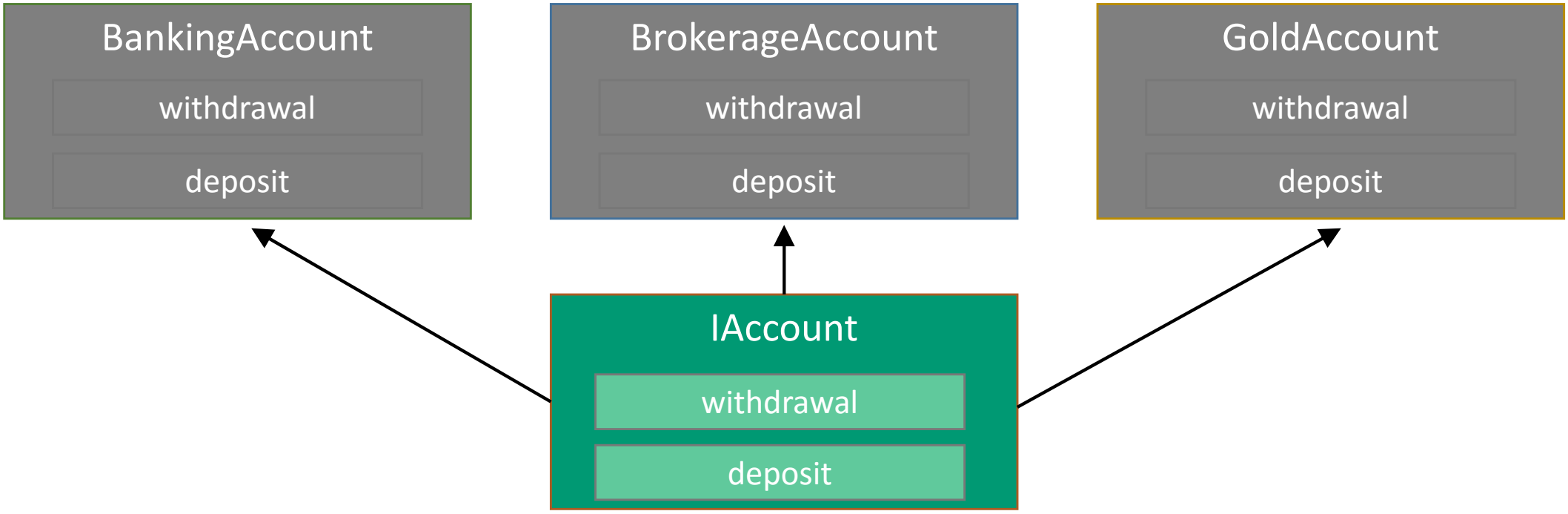
Clarify function parameter and return types

Create custom function and array types

Define type definition files for libraries and frameworks

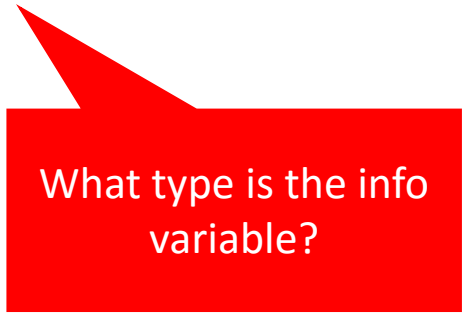
The Need for Interfaces: Scenario 1

Classes can all implement same interface



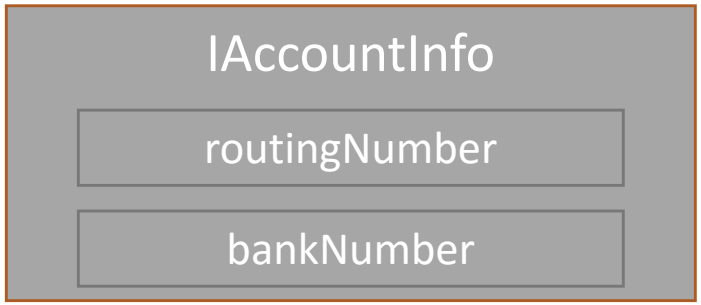
The Need for Interfaces: Scenario 2

```
class BankingAccount {

    get accountInfo() {
        return {
            routingNumber: Constants.ROUTING_NUMBER,
            bankNumber: Constants.BANK_NUMBER
        }
    }
}
var acct = new BankingAccount();
var info = acct.accountInfo();
```



Using an Interface as a Type

```
class BankingAccount {

    get accountInfo() : IAccountInfo {
        return {
            routingNumber: Constants.ROUTING_NUMBER,
            bankNumber: Constants.BANK_NUMBER
        }
    }
}

var acct = new BankingAccount();
var info: IAccountInfo = acct.accountInfo();
```



The type of **info** is clear now

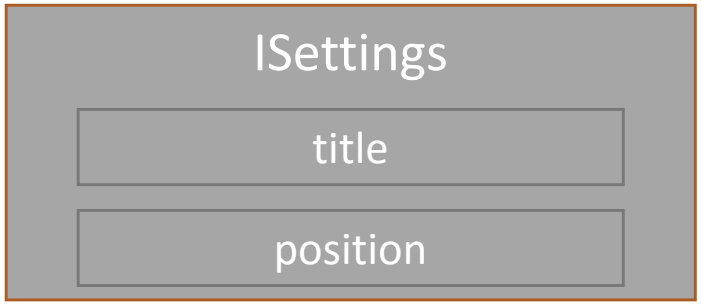
The Need for Interfaces: Scenario 3

```
class MyObject {
    _settings;

    constructor(settings) {
        this._settings = settings;
    }
}
```

What properties does **settings** have?

Using an Interface as a Parameter Type

```
class MyObject {
    _settings: ISettings;

    constructor(settings: ISettings) {
        this._settings = settings;
    }
}
```



Defining an Interface

```
interface IPerson {
    firstName: string;
    lastName: string;
    email: string;
    age?: number
}
```

Implementing an Interface

```
class InterfacePerson implements IPerson {
   constructor(public firstName:string,
            public lastName:string,
            public email:string) {
   }

   greet() {
      return `Hello ${ this.firstName} ${this.lastName},
         your email address is ${this.email}.`
   }
}
```

Creating Custom Array and Function Types on interfaces

Interface as a Function Type


```
interface SearchFunc {
    (source: string, subString: string): boolean;
}

var mySearch: SearchFunc = function (source: string, subString: string) {
    var result = source.search(subString);
    return (result !== -1);
}
```

Checkpoint

Interfaces are Code Contracts

Interfaces can extend other interfaces

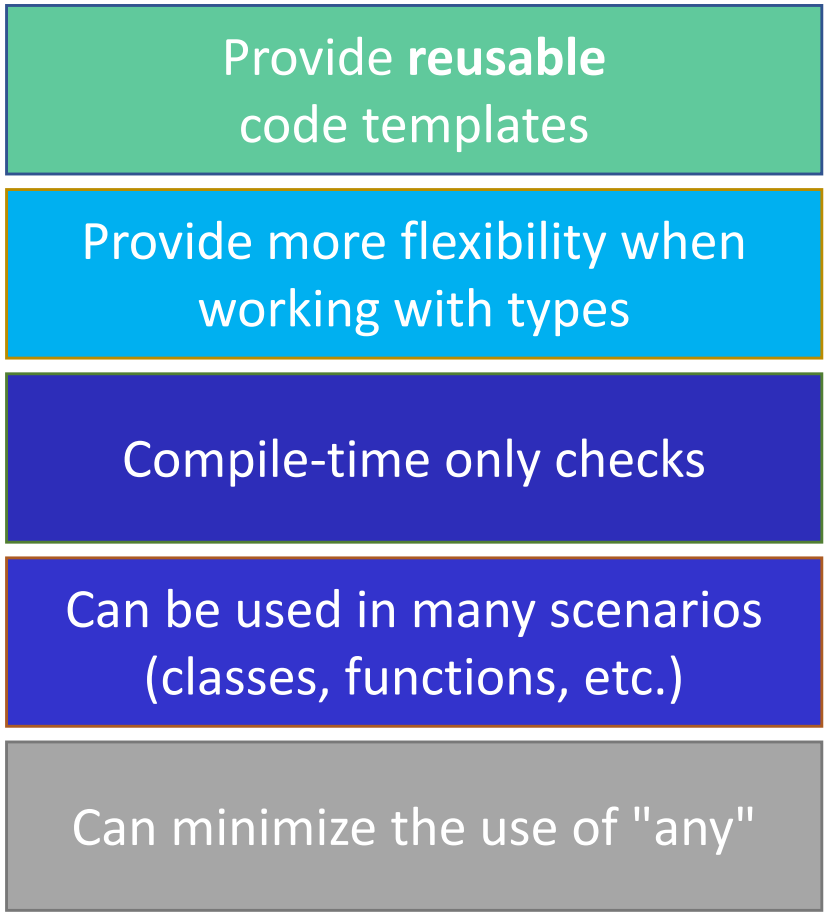
Classes can implement one or more interfaces

Generics

A generic is a "code template" that relies on type variables:

<T>

Generics Features



The Need for Generics

```
class ListOfNumbers {
    _items: number[] = [];

    add(item: number) {
        this._items.push(item);
    }

    getItems(): number[] {
        return this._items;
    }
}
```

```
class ListOfString {
    _items: string[] = [];

    add(item: string) {
        this._items.push(item);
    }

    getItems(): string[] {
        return this._items;
    }
}
```

You find yourself writing duplicate code…

The Answer is Generics

```
class List<T> {
    _items: T[] = [];

    add(item: T) {
        this._items.push(item);
    }

    getItems(): T[] {
        return this._items;
    }
}

var nameList = new List<string>();
```

```
class List {
    _items: string[] = [];

    add(item: string) {
        this._items.push(item);
    }

    getItems(): string[] {
        return this._items;
    }
}
```

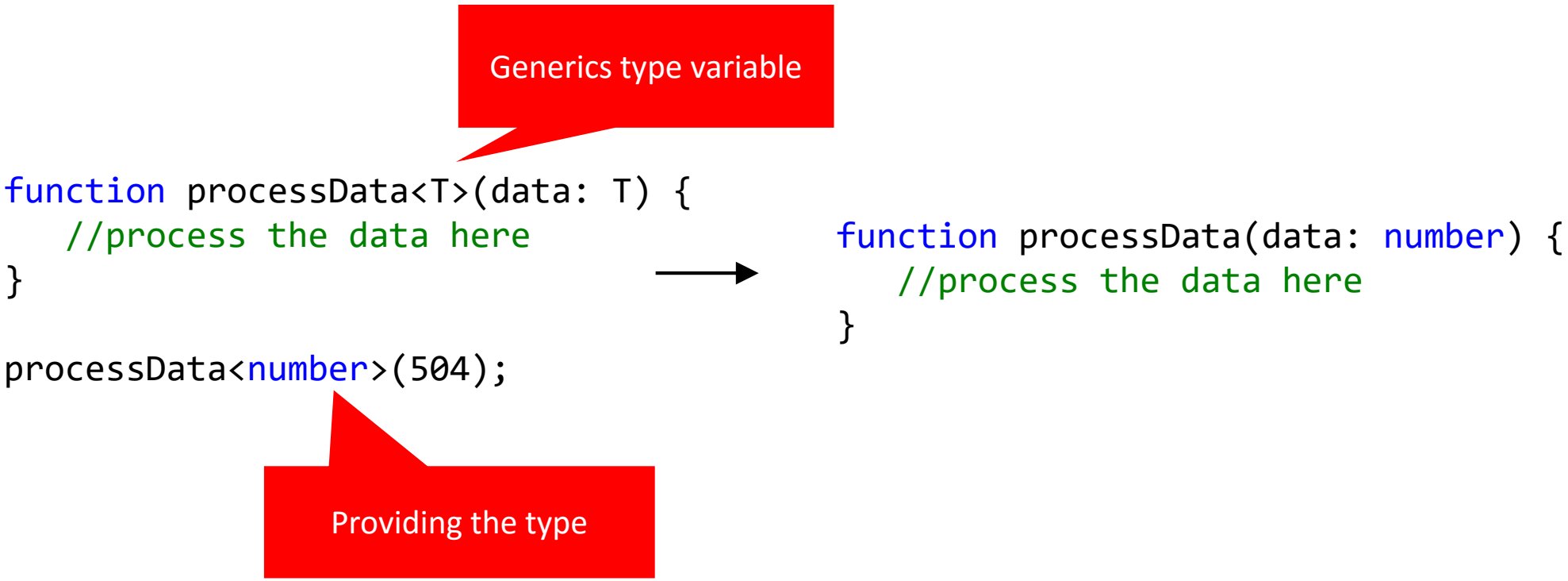
No Generics in transpiled Code

- Again: Generics benefits [only] the developer!
- No type information is found in transpiled code

```
// generics-list.js
var List = (function () {
    function List() {
        this._items = [];
    }
    List.prototype.add = function (item) {
        this._items.push(item);
    };
    List.prototype.getItems = function () {
        return this._items;
    };
    return List;
})();
var nameList = new List();
nameList.add('Peter');
nameList.add('Sandra');
…
```

Advanced Generics

Creating a Generic Function



Using Generics with an Interface

```
interface IAccountInfo<TRouteNumber, TBankNumber> {
    routingNumber: TRouteNumber;
    bankNumber: TBankNumber;
}

class BankingAccount implements IAccount{

    get accountInfo() : IAccountInfo<string, number> {
        return {
            routingNumber: Constants.ROUTING_NUMBER,
            bankNumber: Constants.BANK_NUMBER
        }
    }

}
```

Generic Constraints

T is constrained

```
class List<T extends IAccount> {
    _items: T[] = [];

    add(item: T) {
        this._items.push(item);
    }

    getItems(): T[] {
        return this._items;
    }
}
```

```
interface IAccount extends IDepositWithdrawal {
    accountInfo: IAccountInfo;
    balance : number;
    title: string;
    internalId?: number;
}
```

Checkpoint

Generics are "code templates"

Generic templates rely on type variables: <T>

Generics templates are reusable

Generics provide more flexibility with types