

Angular 2

Lesson 2—Introduction to TypeScript



Introduction

In Angular, JavaScript is replaced by TypeScript (TS).

TypeScript 1.0 was released on April 2014, with ES6 alignment.



TypeScript

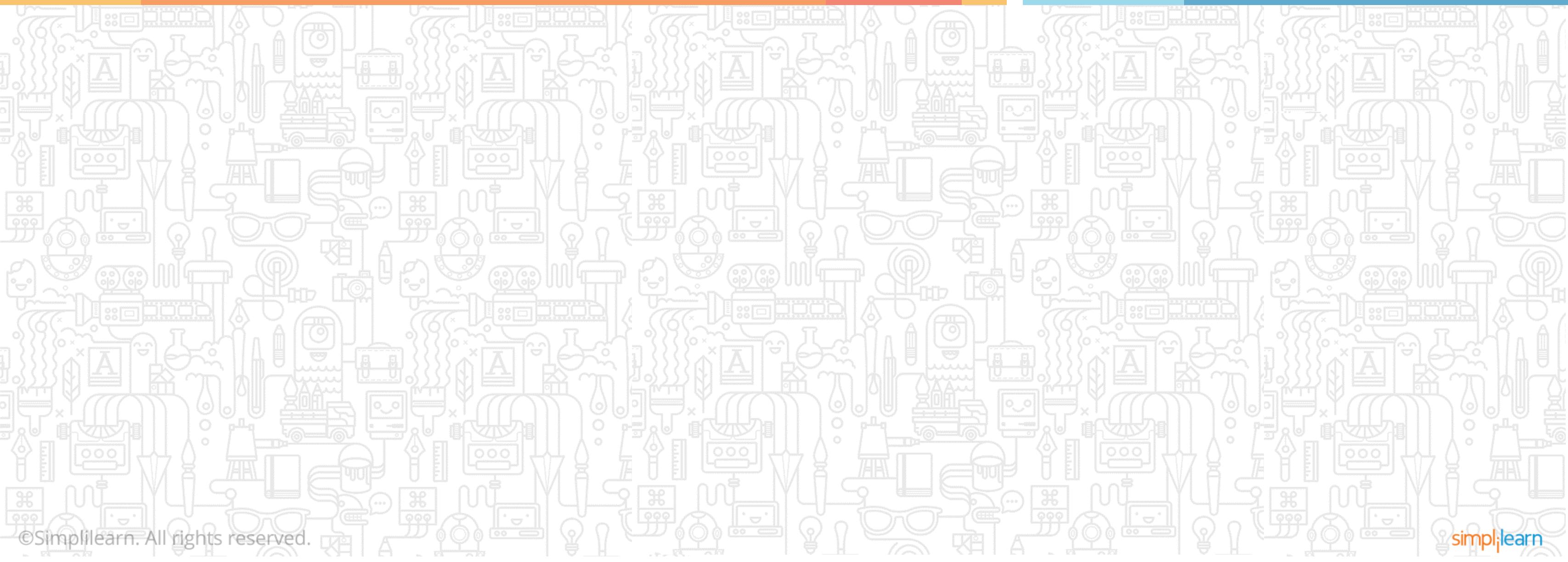
Learning Objectives



- ✓ The need for TypeScript
- ✓ The necessity for Nodemon in TypeScript
- ✓ The usage of String Templates
- ✓ The usage of var, let and const type
- ✓ Knowledge of class and interface
- ✓ Knowledge of class and interface
- ✓ The usage of new collection types, maps, and sets

Introduction to TypeScript

Topic 1—The Need for TypeScript



What's Good With JavaScript?



It's everywhere

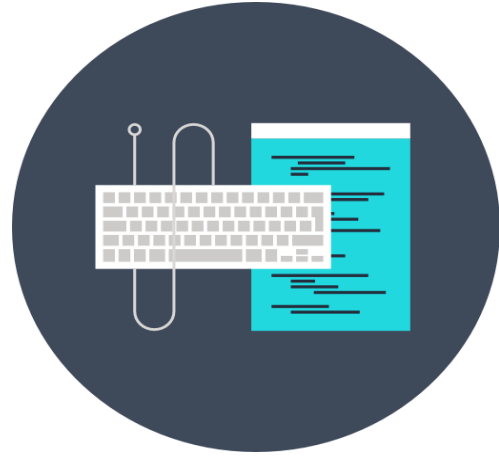


Huge amount of Libraries

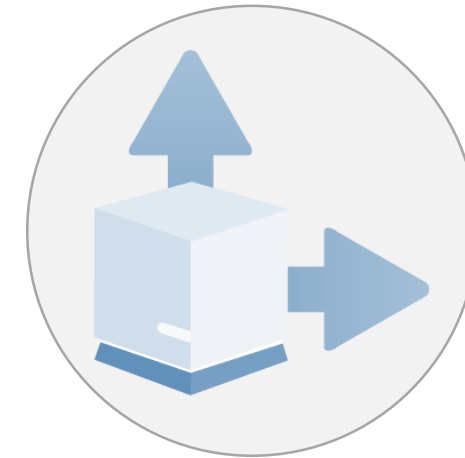


Flexible

What's Wrong With JavaScript?



Dynamic typing



Lack of modularity



Verbose patterns
(Immediately Invoked Function Expression (IIFE))

In short:

JavaScript development scales badly

Advantages of TS Over JS



Scalable HTML5 client-side development



Non-invasive (existing Libs, browser support)



Modular development



Long-term vision

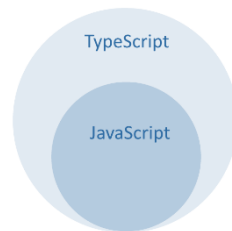


Easy learning for Java Developer



Clean output

TypeScript Features



Superset of JavaScript



Compiles to ES3/ES5



Optionally typed

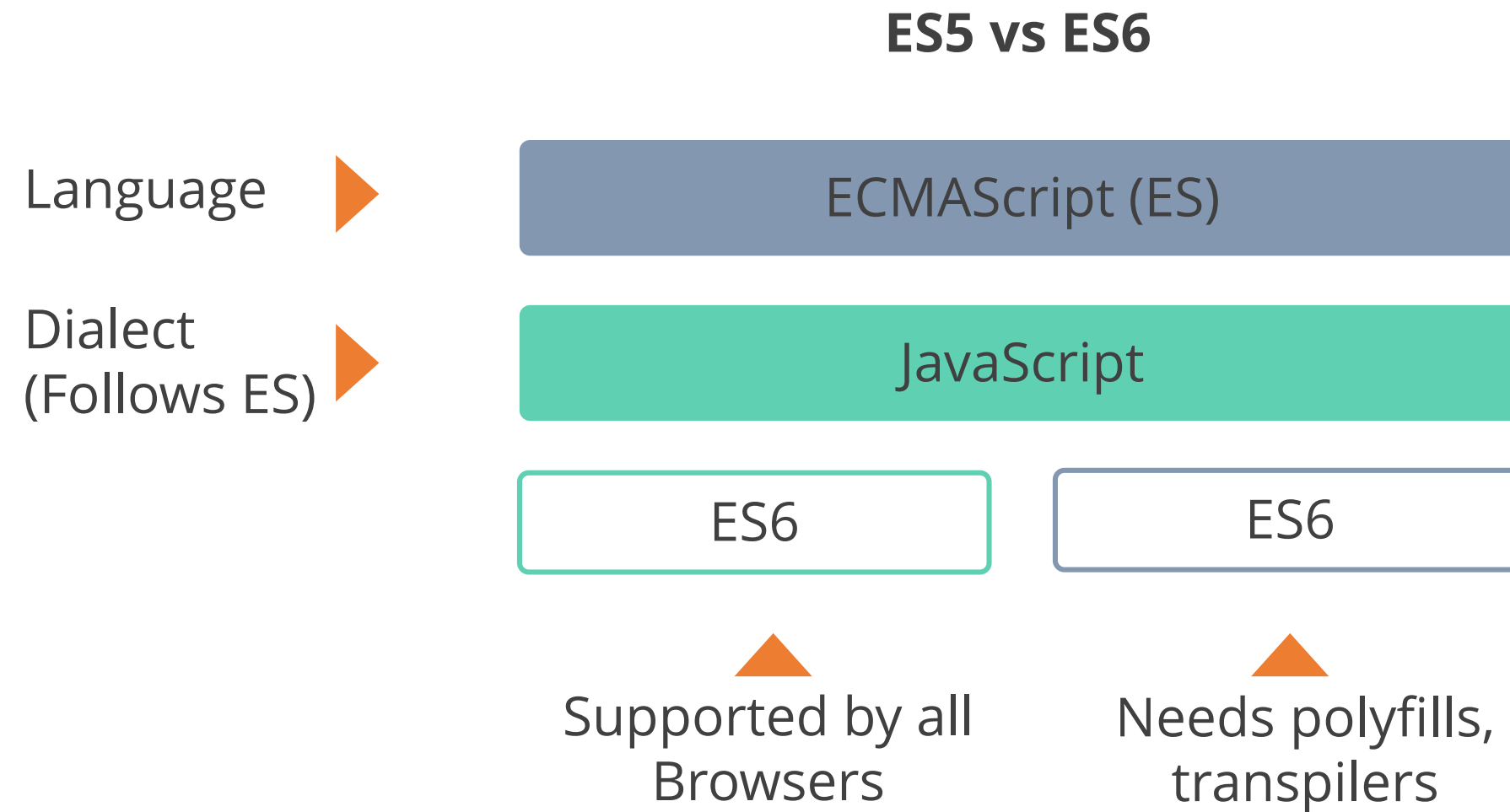


No special runtime

In short:

Lightweight productivity booster

Comparison Between ES5 vs. ES6

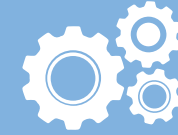


Start And End With JavaScript

Starts and ends with JavaScript

- TypeScript starts from the same syntax and semantics that millions of JavaScript developers know today. Use existing JavaScript code, incorporate popular JavaScript libraries, and call TypeScript code from JavaScript.
- TypeScript compiles to clean, simple JavaScript code which runs on any browser, in Node.js, or in any JavaScript engine that supports ECMAScript (or newer).

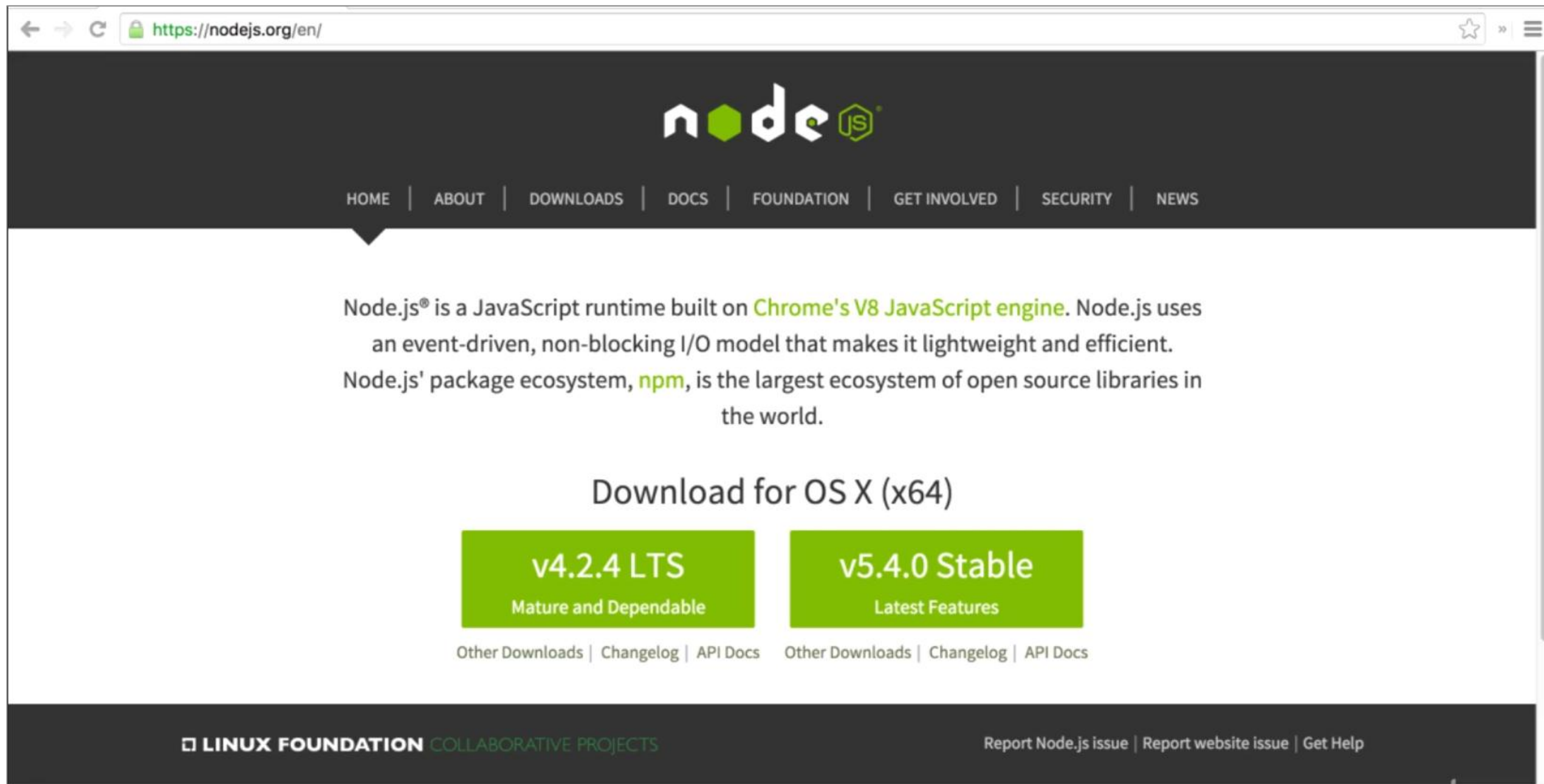
Strong tools for large apps



- Types enable JavaScript developers to use highly-productive development tools and practices like static checking and code refactoring when developing JavaScript applications.
- Types are optional, and type inference allows a few type annotations to make a big difference to the Static Verification of your code. Types let you define interfaces between software components and gain insights into the behavior of existing JavaScript libraries.

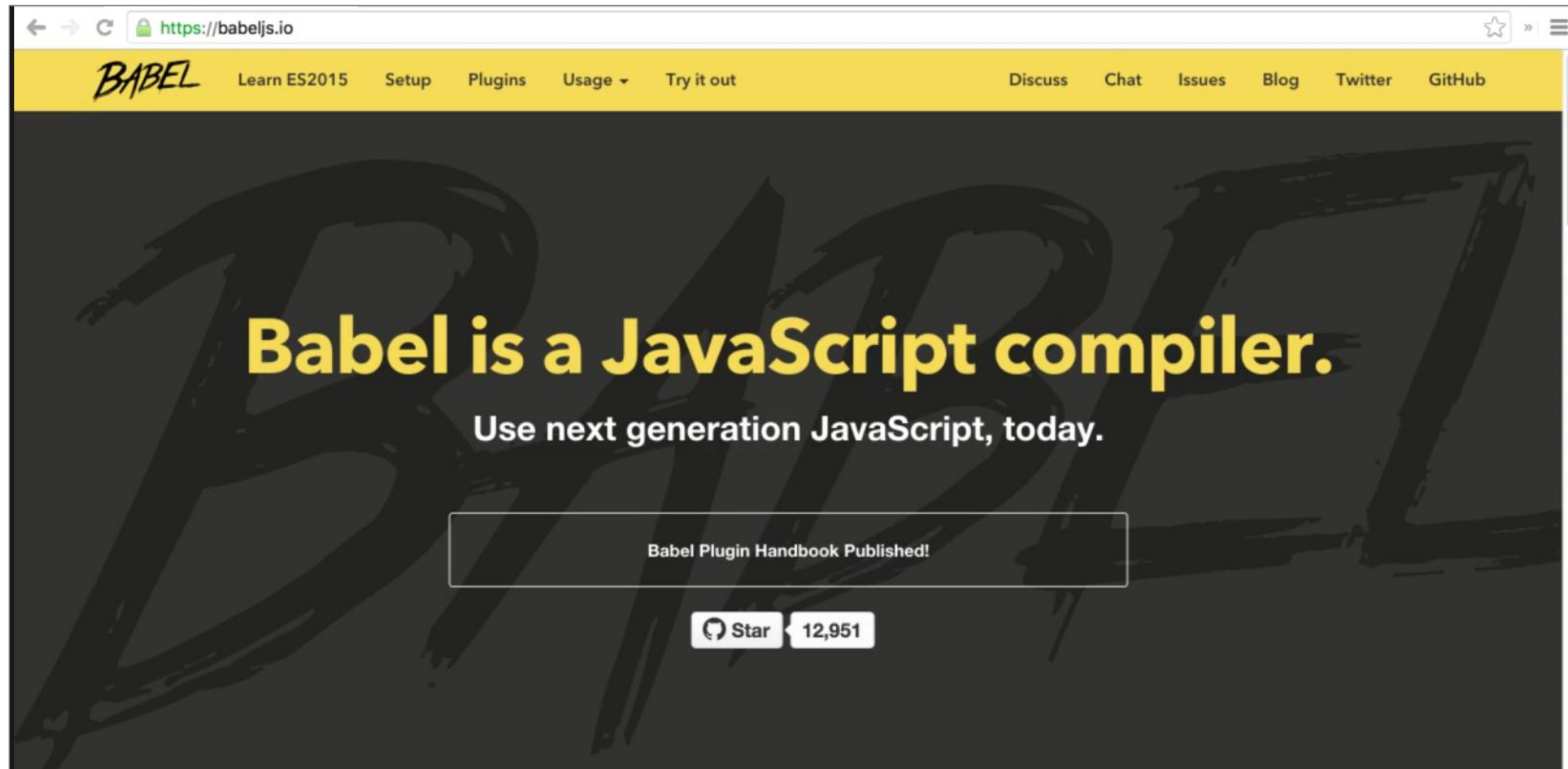
Setting up TypeScript

Download NODEJS



Setting up TypeScript—BABEL

Babel is a translator that is used for converting ES6 specific standard to ES5 standard.



Getting TypeScript

Node.js

The command-line TypeScript compiler can be installed as a Node.js package.

INSTALL

```
npm install -g typescript
```

COMPILE

```
tsc helloworld.ts
```

Visual Studio



Visual Studio 2015



Visual Studio 2013



Visual Studio Code

And More...



Sublime Text



Atom



Eclipse



Emacs



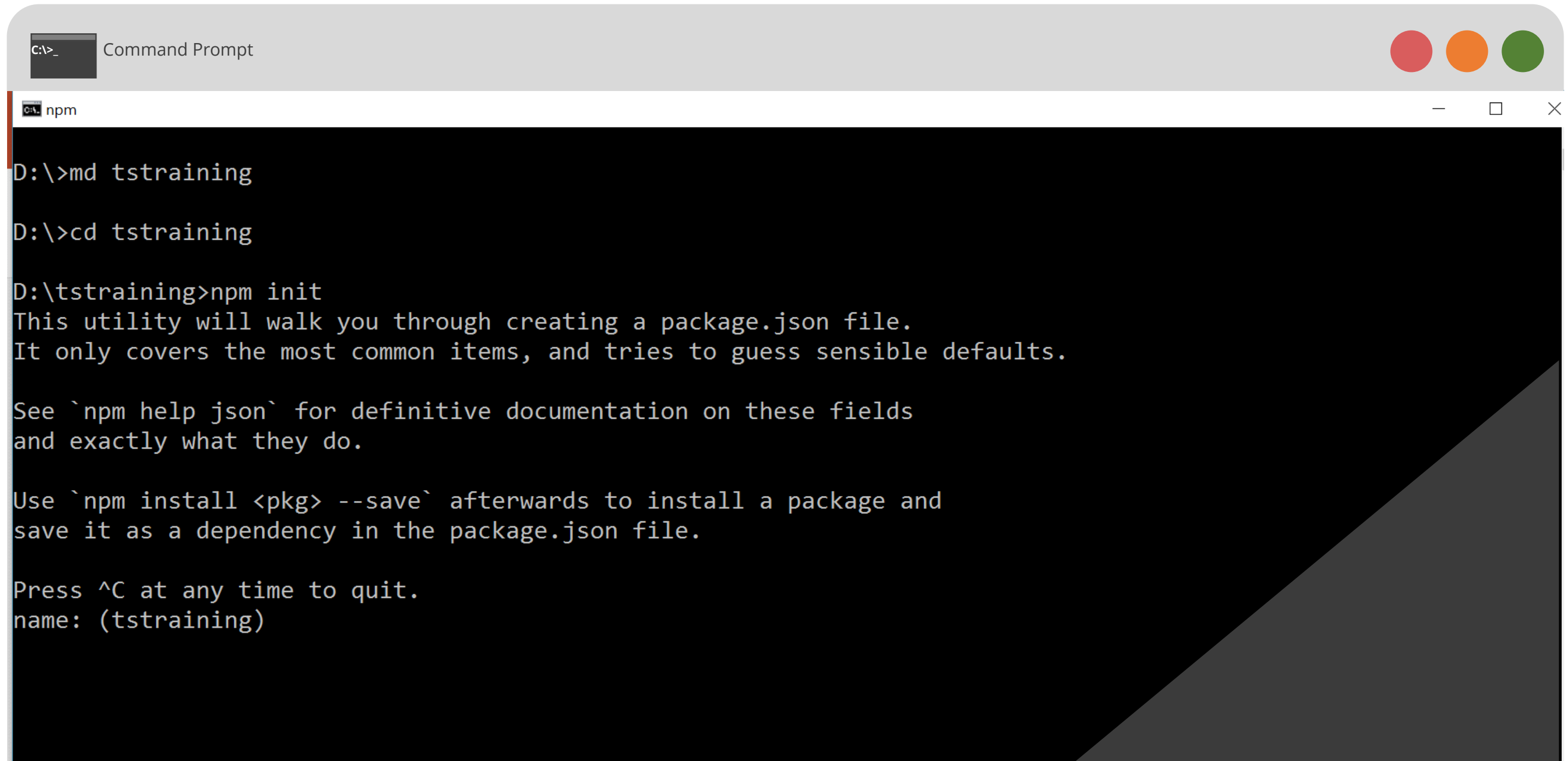
WebStorm



Vim

Setting up TypeScript

CREATE PACKAGE.JSON FILE



```
C:\> Command Prompt

npm

D:\>md tstraining

D:\>cd tstraining

D:\tstraining>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

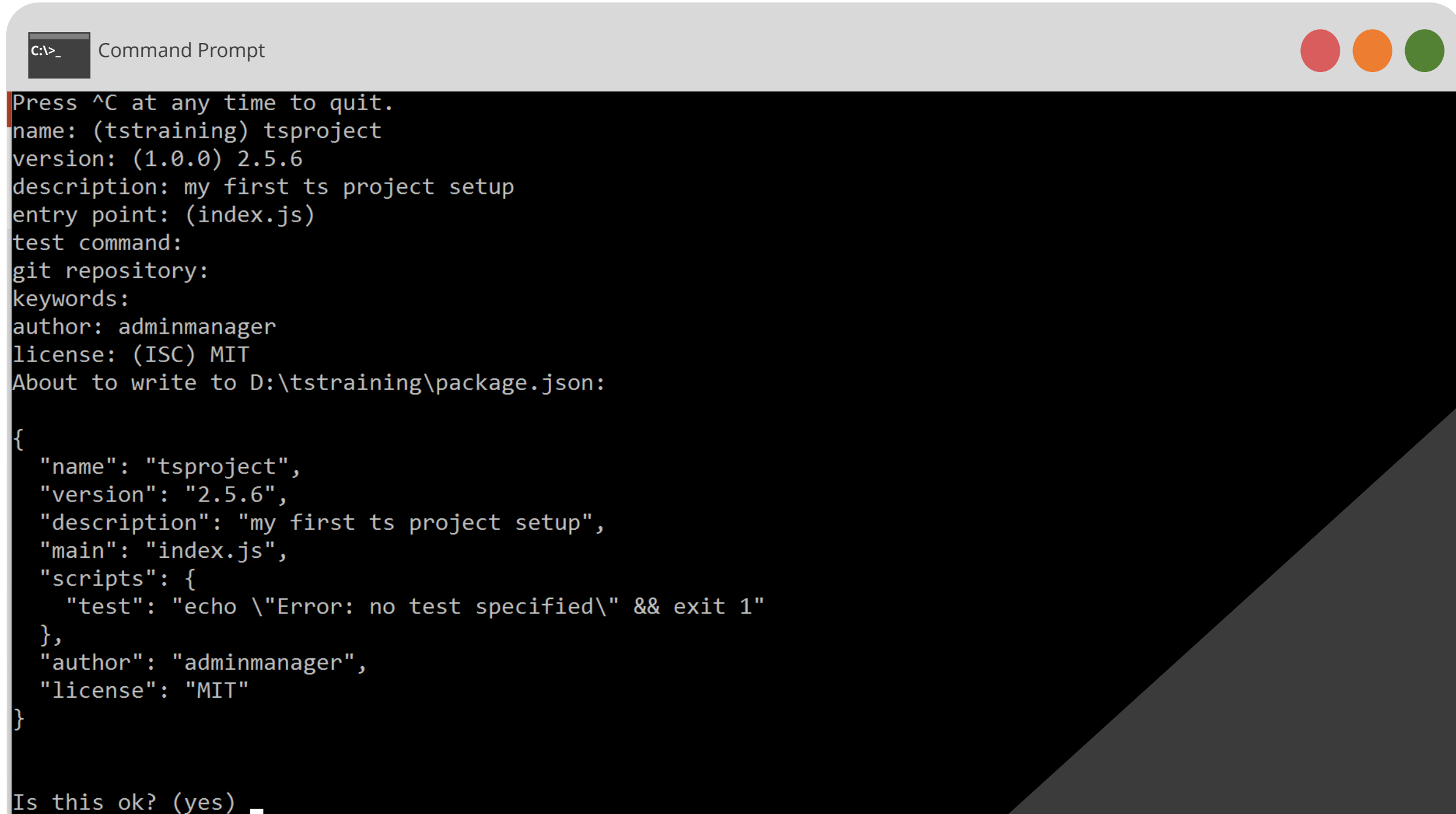
See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (tstraining)
```

Setting up TypeScript (Contd.)

CREATE PACKAGE.JSON FILE



```
C:\> Command Prompt

Press ^C at any time to quit.
name: (tstraining) tsproject
version: (1.0.0) 2.5.6
description: my first ts project setup
entry point: (index.js)
test command:
git repository:
keywords:
author: adminmanager
license: (ISC) MIT
About to write to D:\tstraining\package.json:

{
  "name": "tsproject",
  "version": "2.5.6",
  "description": "my first ts project setup",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "adminmanager",
  "license": "MIT"
}

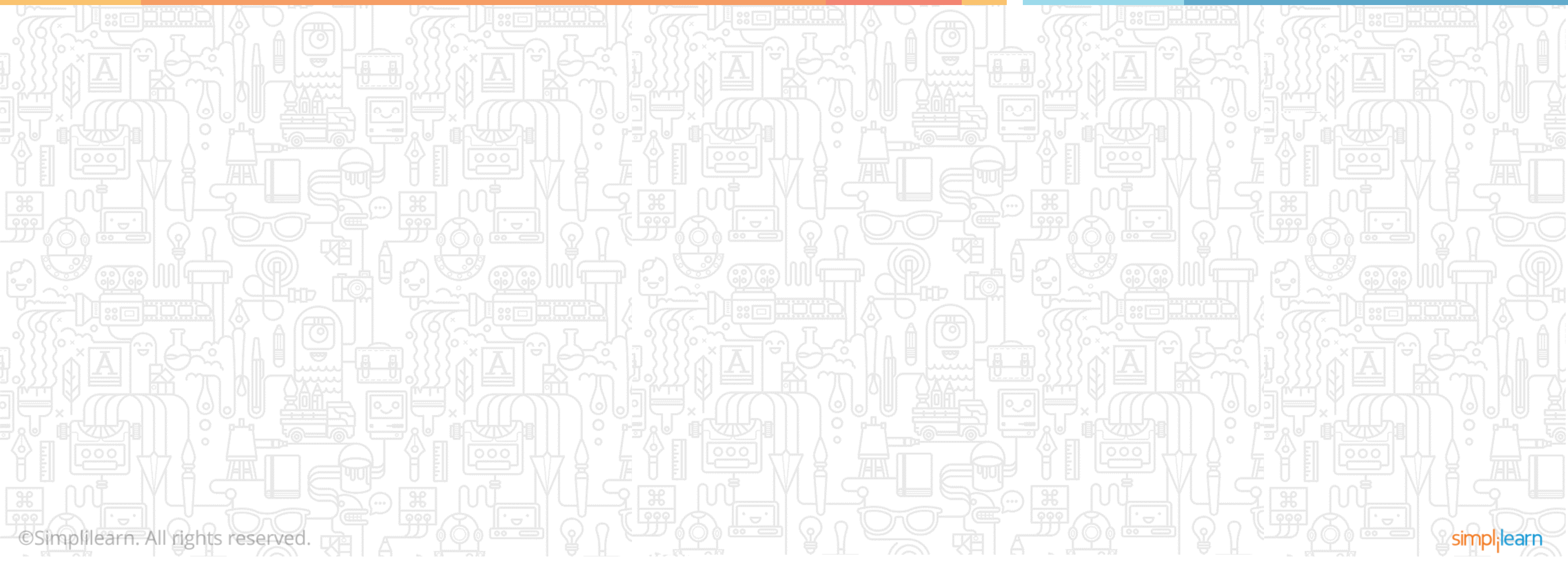
Is this ok? (yes)
```

Lab—Demo



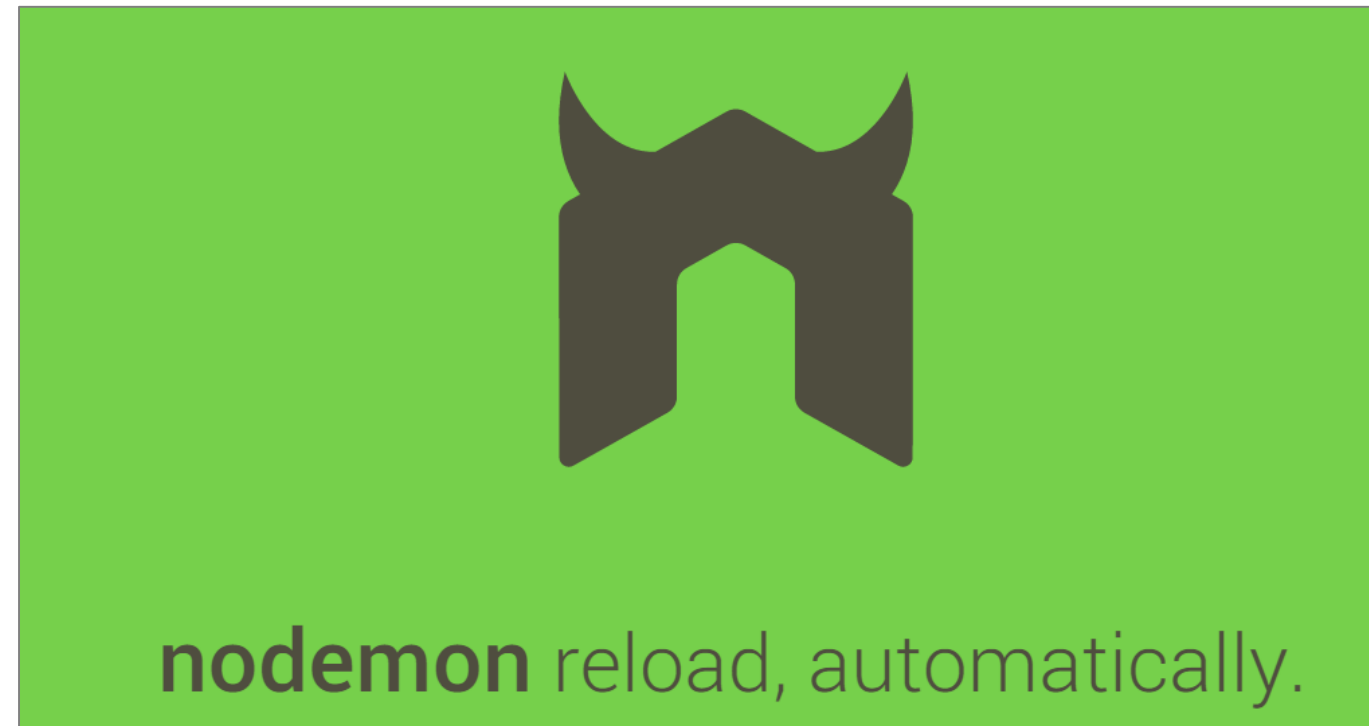
Introduction to TypeScript

Topic 2—Nodemon



Introduction to Nodemon

Nodemon is a utility that will monitor for any changes in your source and automatically restart your server. It is perfect for development. You can install it using [npm](#).



Introduction to Nodemon (Contd.)

- Just use Nodemon instead of node to run your code, and now your process will automatically restart when your code changes.
- To install, get node.js., and then from your terminal run:

```
npm install -g nodemon
```

Features of Nodemon



Automatic restarting of application



Detects default file extension to monitor



Default support for node and CoffeeScript, but easy to run any executable (such as Python, Make, etc.)



Ignores specific files or directories



Watches specific directories



Works with server applications or one time run utilities and Read-Eval-Print-Loop (REPLs)



Required in node apps



Open source and available on GitHub

Lab—Demo



Topic 3—String Template

Topic 3—String Template

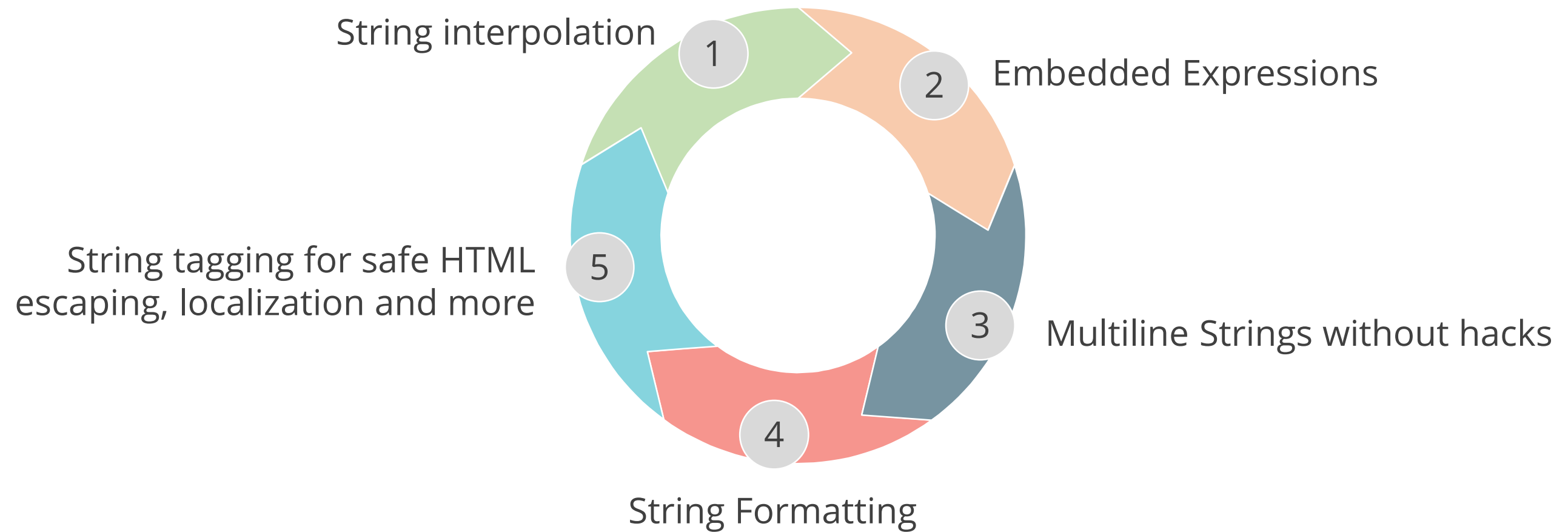
TypeScript Strings Template

- **TypeScript** Template Strings bring a solution to the limitations of JavaScript strings.
- Strings in JavaScript have been historically limited, lacking the capabilities one might expect coming from languages like Python or Ruby.



TypeScript Strings Template

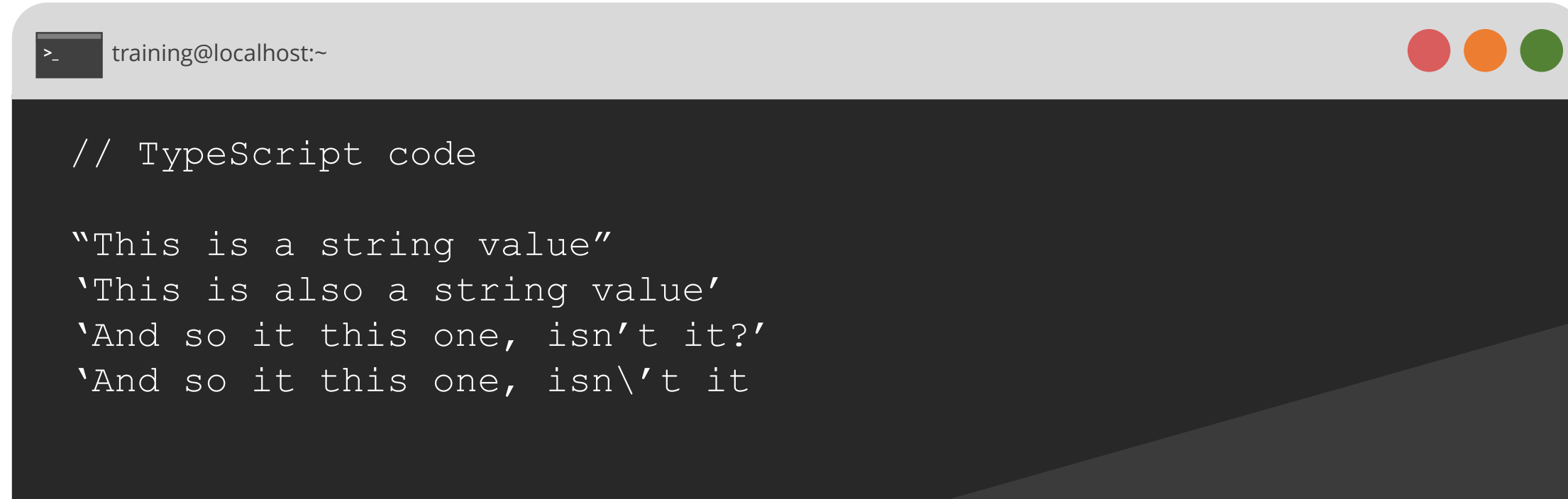
Template Strings introduce a way to define strings with domain-specific languages (DSLs), bringing better:



Rather than stuffing yet another feature into Strings, as we know them today, Template Strings introduce a completely different way of solving these problems.

TypeScript Strings Template—Example

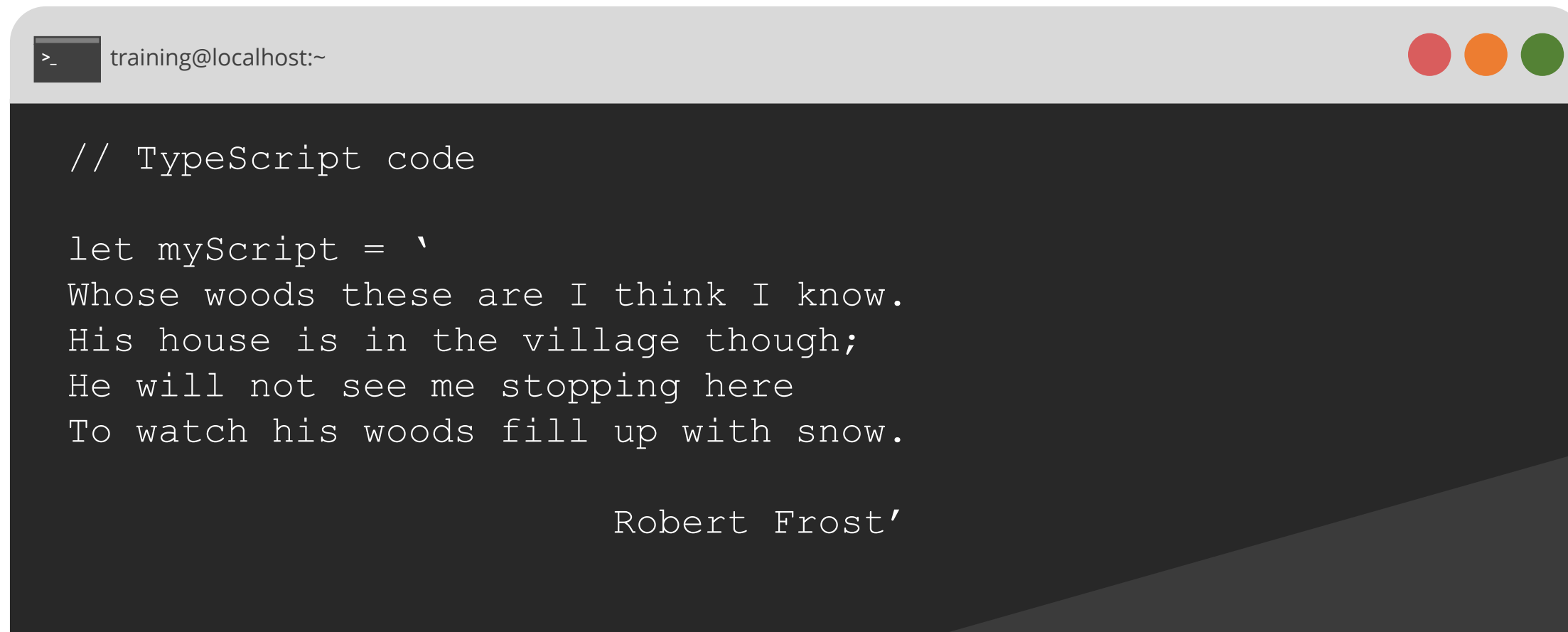
In TypeScript, as in JavaScript, string values are normally wrapped with double or single quotes.

A terminal window with a title bar showing 'training@localhost:~' and three window control buttons (red, orange, green). The terminal content shows TypeScript code for string literals.

```
>_ training@localhost:~  
  
// TypeScript code  
  
"This is a string value"  
'This is also a string value'  
'And so it this one, isn't it?'  
'And so it this one, isn\'t it'
```

TypeScript Strings Template

We can also use backticks to create **string templates** which will be rendered as they are.

A terminal window with a light gray title bar. The title bar contains a prompt icon, the text 'training@localhost:~', and three window control buttons (red, orange, green). The terminal area has a dark background and displays the following TypeScript code in a light gray monospace font:

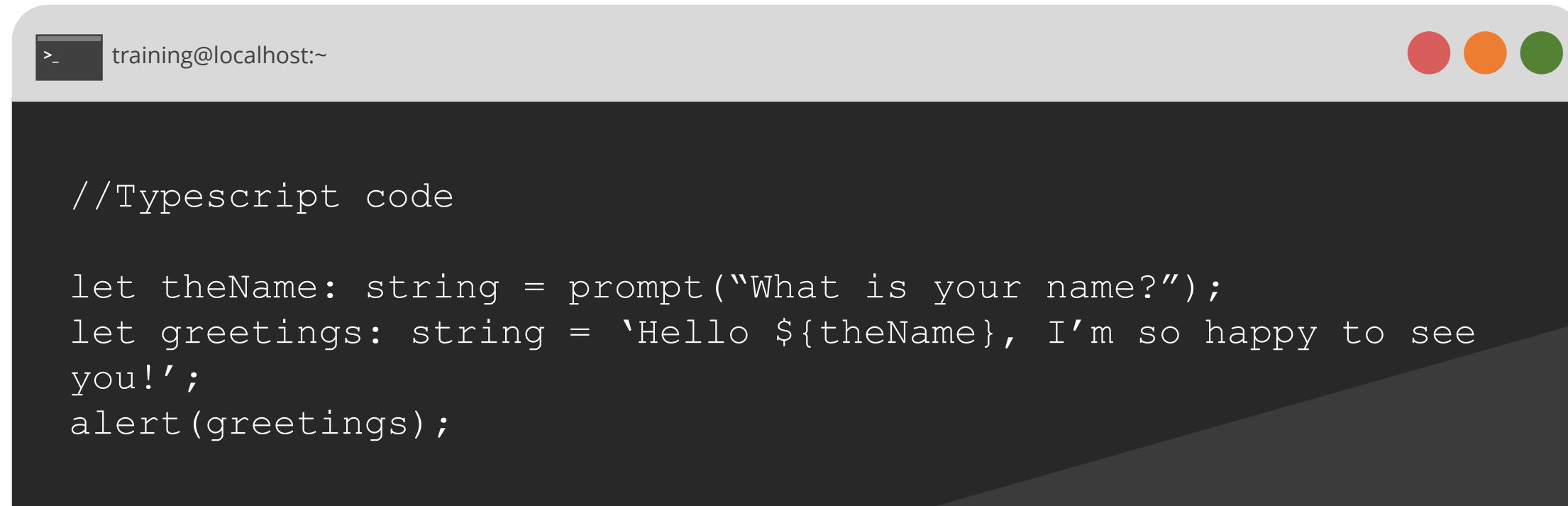
```
// TypeScript code

let myScript = `
Whose woods these are I think I know.
His house is in the village though;
He will not see me stopping here
To watch his woods fill up with snow.

                Robert Frost`
```

TypeScript Strings Template—Example

String templates can contain placeholders for variable data as seen in the example:

A terminal window with a title bar showing 'training@localhost:~' and three window control buttons (red, orange, green). The terminal content shows TypeScript code using string templates.

```
//Typescript code

let theName: string = prompt("What is your name?");
let greetings: string = `Hello ${theName}, I'm so happy to see
you!`;
alert(greetings);
```

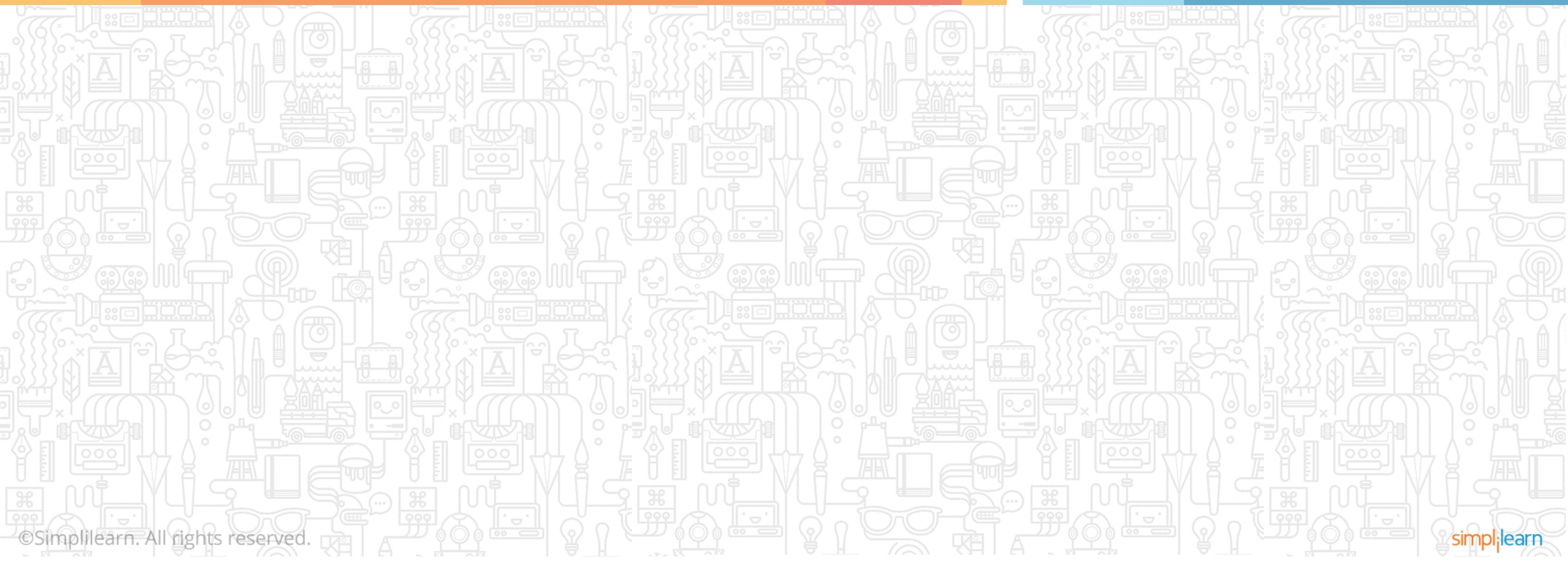
The data collected by the *prompt* is assigned to variable *theName*, which then is embedded in the `${ }` template and displayed along with the data from variable *greetings*.

Lab—Demo



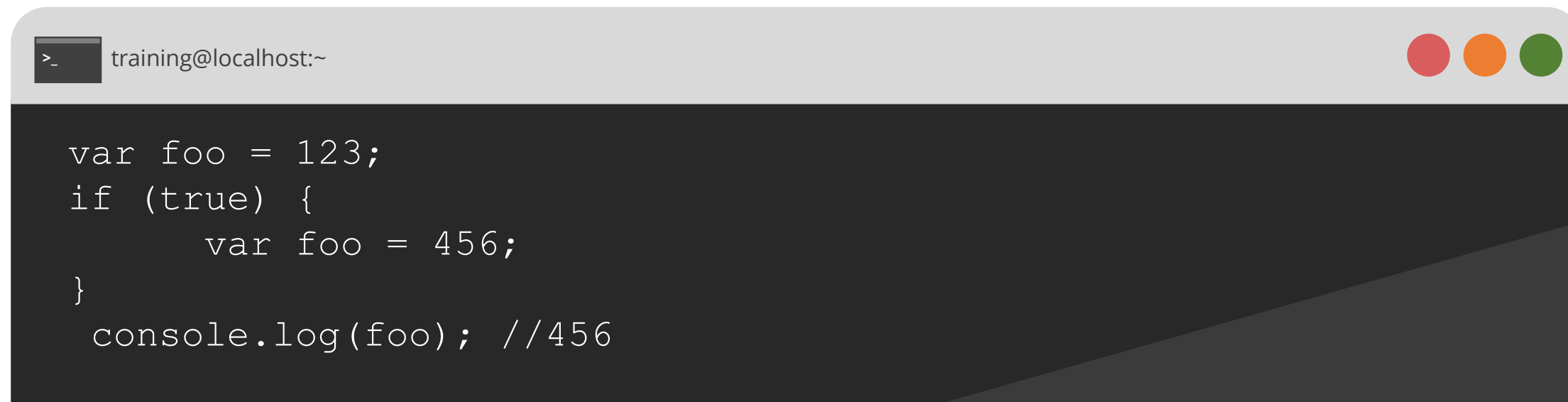
Introduction to TypeScript

Topic 4—var, let, and const Keyword



var

- **var** variables in JavaScript are function scoped.
- This is different from many other languages (C# / Java etc.) where the variables are block scoped.
- If you bring a block scoped mindset to JavaScript, you would expect the following to print 123. Instead, it will print 456.



```
>_ training@localhost:~  
  
var foo = 123;  
if (true) {  
    var foo = 456;  
}  
console.log(foo); //456
```

let Declaration

Developers introduced the *let* statements because there were problems with *var* statements. Apart from the keyword used, *let* statements are written the same way as *var* statements.

```
let hello = "Hello!";
```

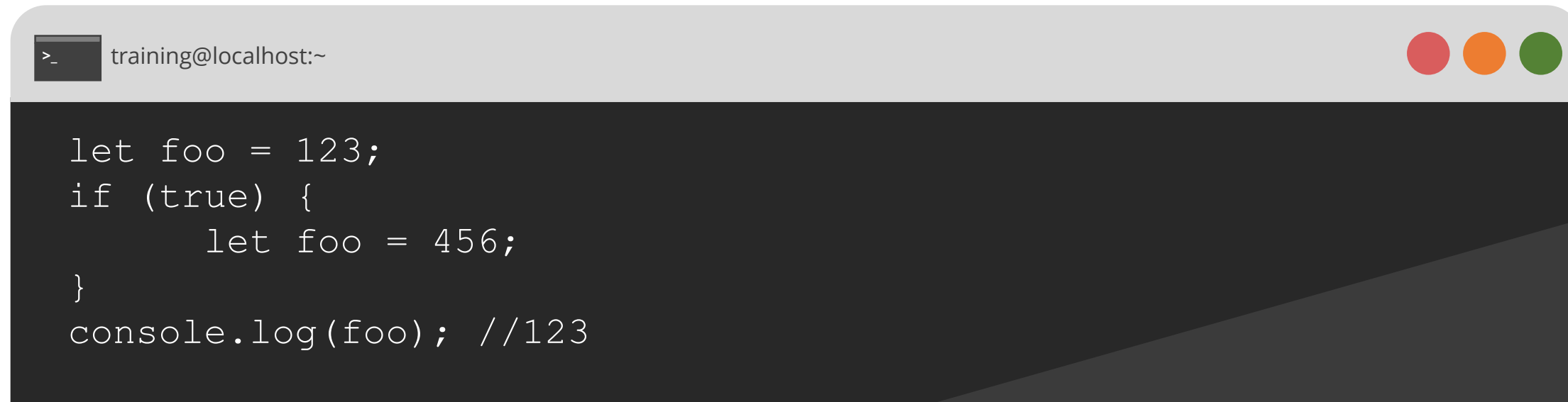
The key difference is not in the syntax, but in the semantics.

Block scoping

- When a variable is declared using `let`, it uses what some call lexical-scoping or block scoping.
- Unlike variables declared with `var`, whose scopes leak out to their containing function, block-scoped variables are not visible outside of their nearest containing block or for-loop.

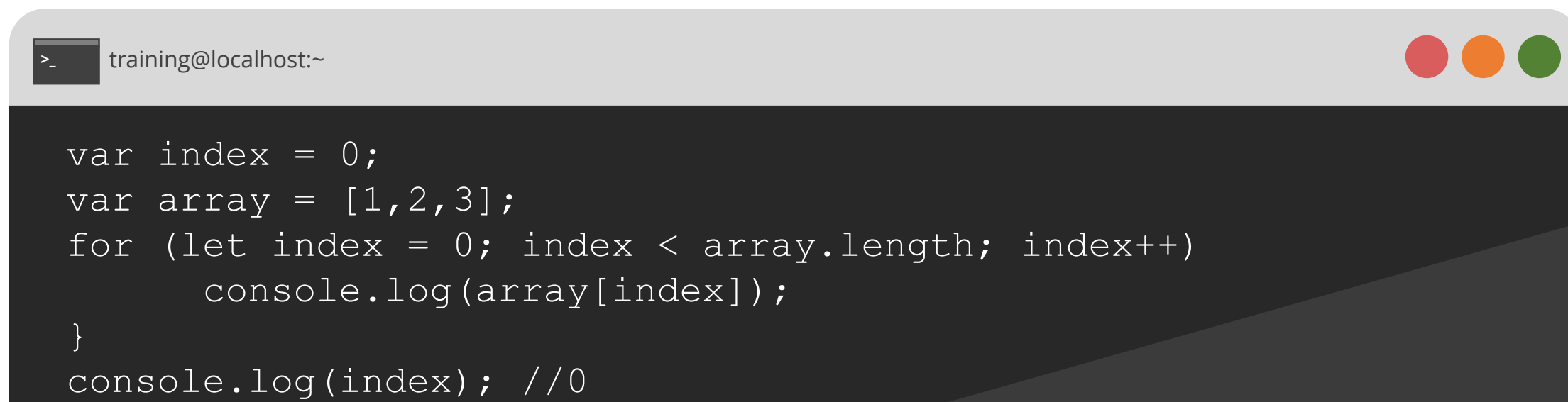
let Declaration—Example

The same example is demonstrated with let:

A terminal window with a title bar showing 'training@localhost:~' and three window control buttons (red, orange, green). The terminal content shows a JavaScript code snippet using the 'let' keyword for variable declaration and assignment within a conditional block.

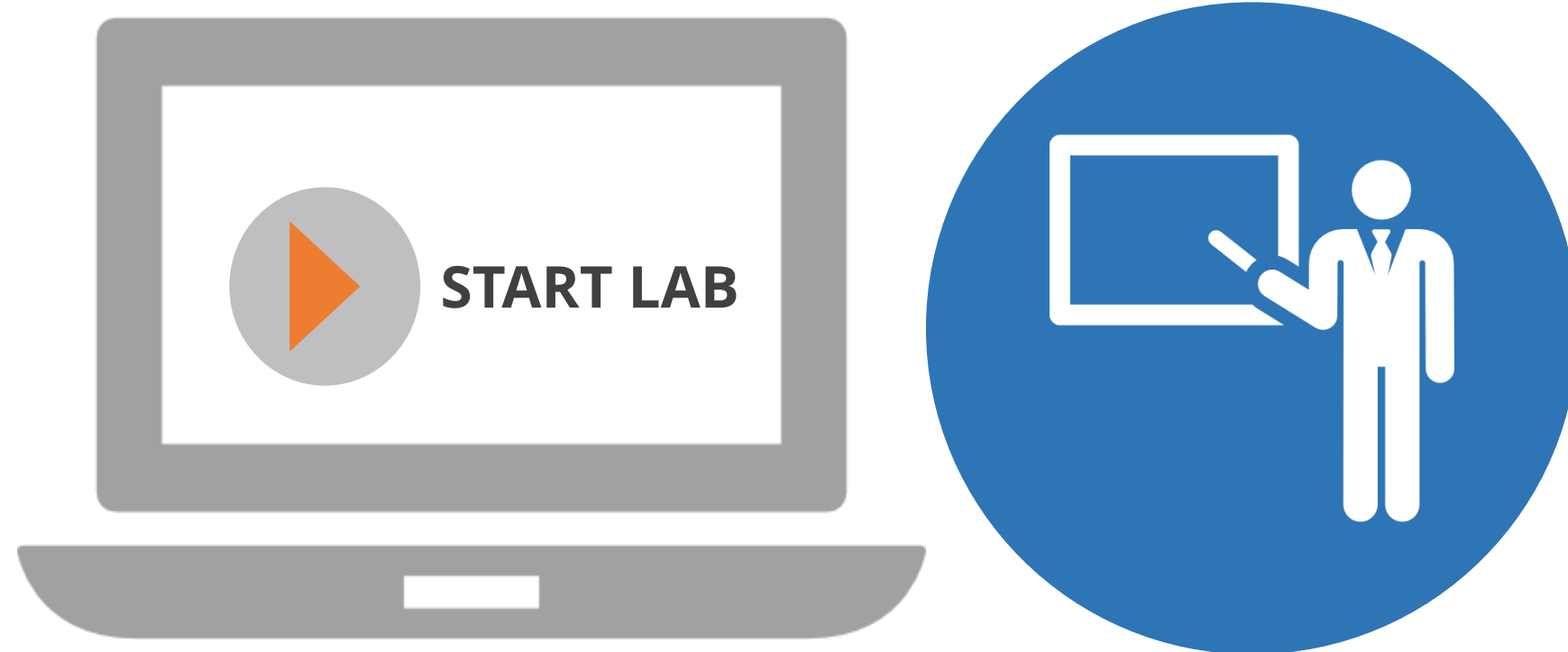
```
>_ training@localhost:~  
  
let foo = 123;  
if (true) {  
    let foo = 456;  
}  
console.log(foo); //123
```

Another place where let would save you from errors is loops:

A terminal window with a title bar showing 'training@localhost:~' and three window control buttons (red, orange, green). The terminal content shows a JavaScript code snippet using 'let' to declare a loop index, demonstrating how it prevents variable leakage from the loop scope.

```
>_ training@localhost:~  
  
var index = 0;  
var array = [1,2,3];  
for (let index = 0; index < array.length; index++)  
    console.log(array[index]);  
}  
console.log(index); //0
```


Lab—Demo



const Declaration

- const is a welcome addition offered by ES6 / TypeScript. It allows you to be immutable with variables.

- This is good from a documentation as well as a runtime perspective.

- To use const, just replace var with const:
const foo = 123;

- Using const is good for both readability and maintainability. It avoids using magic literals.

```
>_ training@localhost:~
```

```
// Low readability  
if (x>10){  
}
```

```
//Better!  
const maxRows = 10;  
if(x>maxRows) {
```

const Declaration (Contd.)

- They are like let declarations, but as the name implies, their value cannot be changed once they are bound.
- In other words, they have the same scoping rules as let, but you can't reassign to them.
- This should not be confused with the idea that the values they refer to are immutable.

A *const* is block scoped like the let declaration:

```
>_ training@localhost:~  
  
const foo = 123;  
if(true) {  
    const foo = 456; //Allowed as itsa new variable limited to  
    this 'if' block  
  
}
```

let vs. const

Applying the principle of least privilege, all declarations other than those you plan to modify should use const.

| const | let |
|---|--|
| Using const also makes code more predictable when reasoning about flow of data. | On the other hand, let is shorter to write than var, and many users will prefer its brevity. |

Lab—Demo



Debugging TypeScript in Firefox or Chrome

Firefox and Chrome have built-in debuggers, so you can easily debug the generated JavaScript. However, if you want to develop in the language you have programmed (be it TypeScript, Dart, or CoffeeScript), you need another option.



Debugging TypeScript in Firefox or Chrome (Contd.)

Source Maps:



All major browsers support source maps now, but may require you to jump through a couple hoops. A source map is basically what it says—a map from one language to another. So the debugger can run the JavaScript code, but shows you the line that actually generated it.

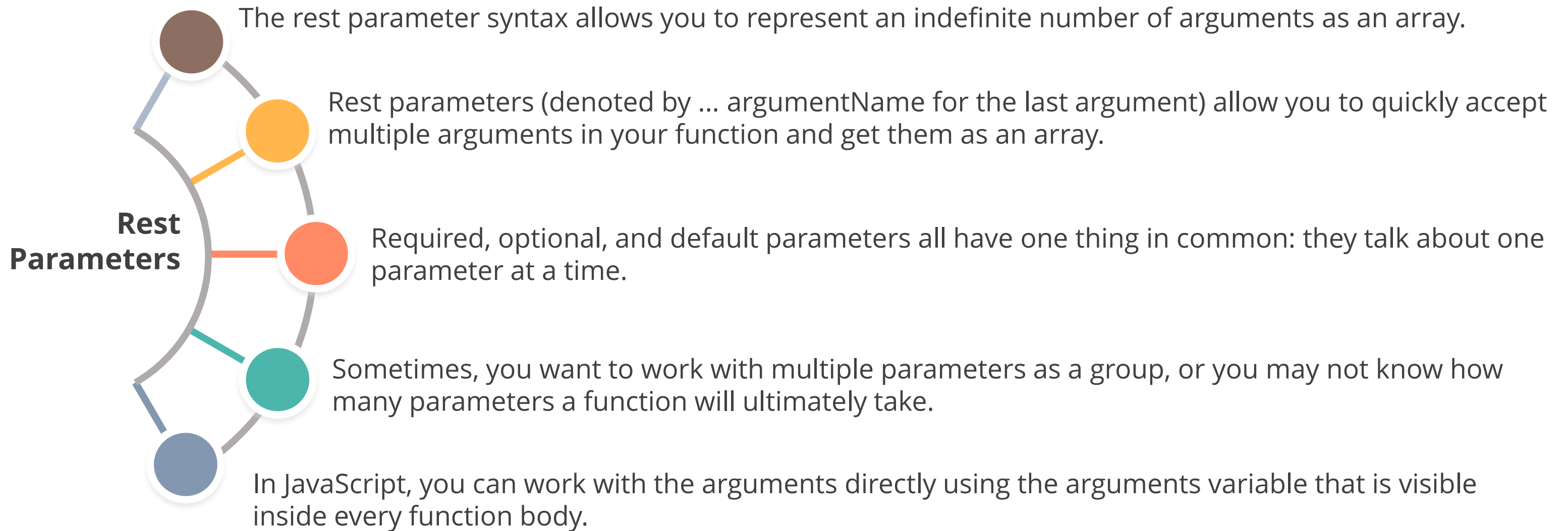


First you need to tell Visual Studio to generate a source map for your TypeScript project. Right click your Project file in Solution Explorer and select Properties.

Lab—Demo



Rest Parameter



Examples for Rest Parameter and Without Rest Parameter

```
>_ training@localhost:~  
  
function foo(x:number, y:number, z:number){  
    console.log(x,y,z);  
}  
var args:number[,] = [0,1,2];  
  
(<any>foo) (...args);
```

This results in the same apply function call, that you'd expect:

```
>_ training@localhost:~  
  
function foo(x, y, z){  
    console.log(x,y,z);  
}  
var args = [0,1,2];  
  
foo.apply(void 0, args);
```

With Rest Parameter:

```
>_ training@localhost:~  
  
function foo(...x: number[]){  
    console.log(JSON.stringify(x));  
}  
var args:number[] = [0,1,2];  
  
foo(...args);
```

Spread Operator

The spread operator is the opposite of destructuring. It allows you to spread an array into another array, or an object into another object. For example:

```
training@localhost:~  
let first = [1,2];  
let second = [3,4]  
let bothPlus = [0, ...first, ...second, 5];
```

This gives bothPlus the value [0, 1, 2, 3, 4, 5]. Spreading creates a shallow copy of first and second. The values are not changed by the spread.

You can also spread objects:

```
training@localhost:~  
let defaults = {food: "spicy", price: "$$", ambiance: "noisy"};  
let search = {...defaults, food: "rich"};
```

Lab—Demo



Arrow Functions

- Described as the fat arrow (because `->` is a thin arrow and `=>` is a fat arrow) and also called a lambda function (because of other languages).

```
() => { }
```

- Another commonly used feature is the fat arrow function `()=>something`.

The advantage of a fat arrow is:

- You don't need to keep typing function
- It lexically captures the meaning of this
- It lexically captures the meaning of arguments
- For a language that claims to be functional, in JavaScript you tend to be typing function quite a lot. The fat arrow makes it simple for you to create a function

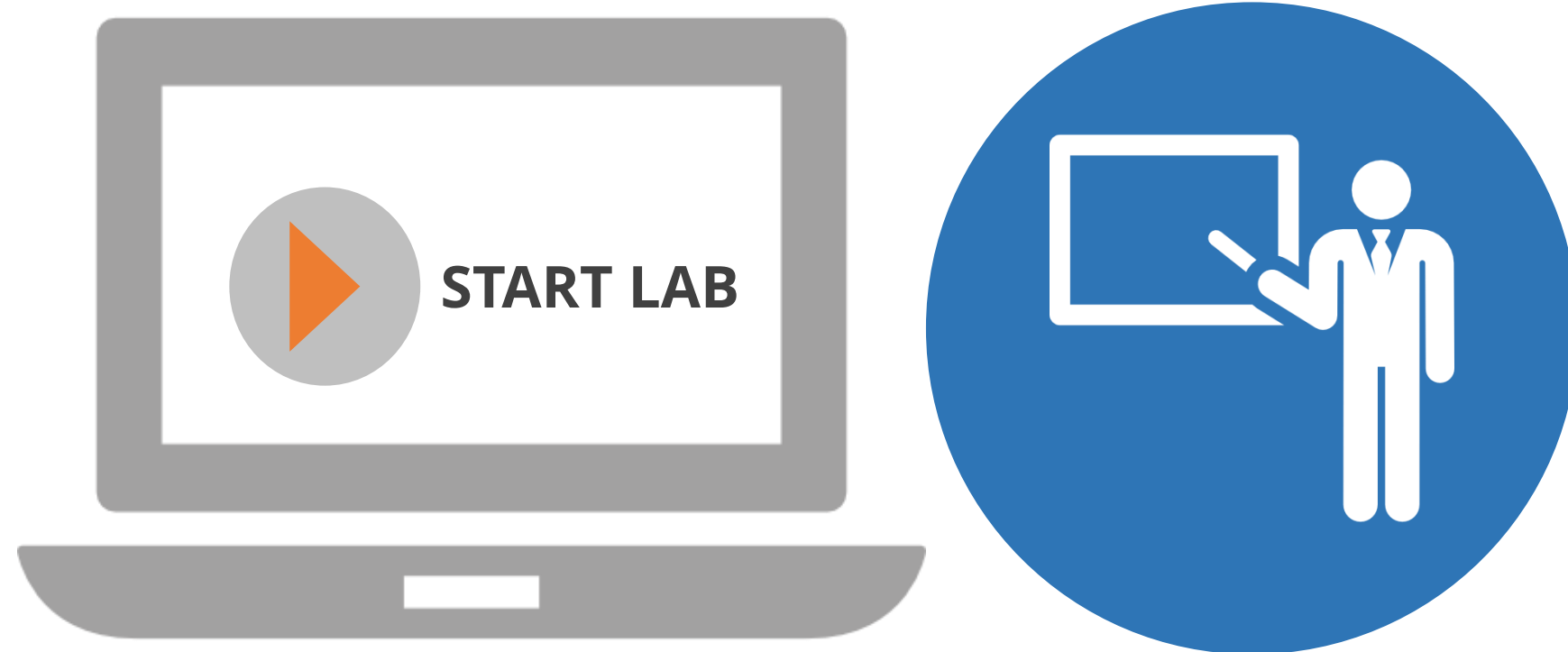
```
var inc = (x)=>x+1;
```

this Loop

“this” has traditionally been a pain point in JavaScript. Fat arrows fix it by capturing the meaning of this loop from the surrounding context.

```
>_ training@localhost:~  
  
function Person(age) {  
  this.age = age;  
  this.growOld = () => {  
    this.age++;  
  }  
}  
var person = new Person(1);  
setTimeout(person.growOld, 1000);  
  
setTimeout(function() { console.log(person.age); }, 2000); //2
```

Lab—Demo



for...of

A common error experienced by JavaScript developers is that *for...in* for an array does not iterate over the array items.

Instead, it iterates over the keys of the object passed in.

This is demonstrated in the example. Here you would expect 9,2,5, but you get the indexes 0,1,2:

```
training@localhost:~  
  
var someArray = [9,2,5];  
for (var item in someArray) {  
    console.log(item); //0,1,2  
}
```

This is one of the reasons why *for...of* exists in TypeScript (and ES6). The following iterates over the array correctly logging out the members as expected:

```
training@localhost:~  
  
var someArray = [9,2,5];  
for (var item of someArray) {  
    console.log(item); //9,2,5  
}
```


Lab—Demo



Topic 5 — Class Interfaces

Topic 5 — Class Interfaces

Introduction to Classes

Traditional JavaScript uses functions and prototype-based inheritance to build up reusable components. But this may feel a bit awkward to programmers who are more comfortable with an object-oriented approach, where classes inherit functionality and objects are built from these classes.

Starting with ECMAScript 2015, also known as ECMAScript 6, JavaScript programmers will be able to build their applications using this object-oriented class-based approach.

In TypeScript, developers 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.

Importance of Classes



The reason why it's important to have classes in JavaScript as a first class item is that:

Classes offer a useful structural abstraction.

Provides a consistent way for developers to use classes instead of every framework (emberjs, reactjs etc.) coming up with their own version.

Object Oriented developers already understand classes.

Finally, JavaScript developers can have class. Here, there is a basic class called Point.

Classes

>_ training@localhost:~

```
class Point {  
    x: number;  
    y: number;  
    constructor(x:number, y:number) {  
        this.x = x;  
        this.y = y;  
    }  
    add(point: Point) {  
        return new Point(this.x + point.x, this.y +  
point.y);  
    }  
}  
  
var p1 = new Point(0,10);  
var p2 = new Point(10,20);  
var p3 = p1.add(p2); // {x:10,y:30}
```

Lab—Demo



Interfaces

One of TypeScript's core principles is that type-checking focuses on the shape that values have.

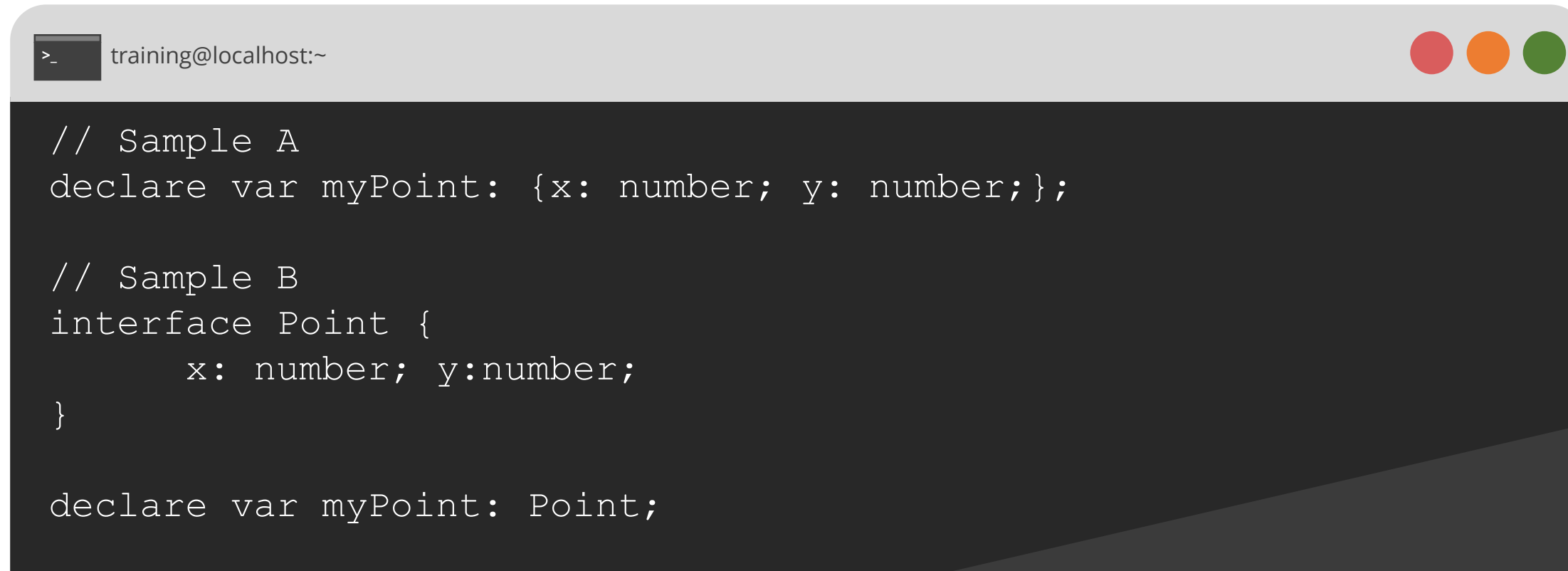
This is sometimes called “duck typing” or “structural subtyping.”

In TypeScript, interfaces fill the role of naming these types and are used as a powerful way of defining contracts within your code as well as contracts with code outside of your project.

Interfaces have zero runtime JS impact. There is a lot of power in TypeScript interfaces to declare the structure of variables.

Interfaces

The two samples are equivalent declarations, the first uses an inline annotation, the second uses an interface.

A terminal window with a title bar showing 'training@localhost:~' and three window control buttons (red, orange, green). The terminal content shows two code samples. Sample A uses an inline object type for myPoint. Sample B uses an interface named Point for the same purpose.

```
>_ training@localhost:~  
  
// Sample A  
declare var myPoint: {x: number; y: number;};  
  
// Sample B  
interface Point {  
    x: number; y: number;  
}  
  
declare var myPoint: Point;
```

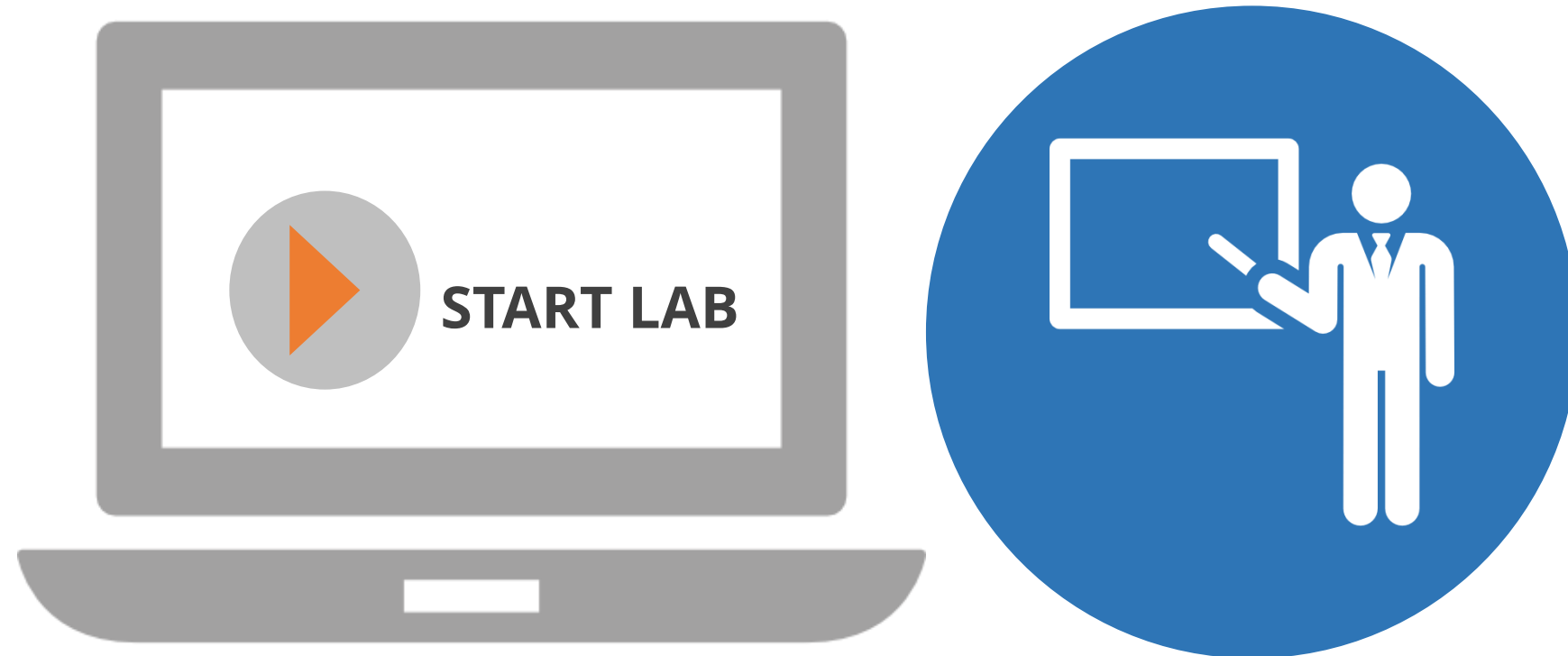
However, the advantage of Sample B is that if someone authors a library that builds on the myPoint library to add new members, they can easily add to the existing declaration of myPoint.

Classes Can Implement Interfaces

If you want to use classes that must follow an object structure that someone declared for you in an interface, you can use the implements keyword to ensure compatibility.

```
>_ training@localhost:~  
  
interface Point{  
    x: number; y: number;  
}  
  
class MyPoint implements Point{  
    x: number; y: number; //Same as Point  
}
```

Lab—Demo



Introduction to Generics

A major part of software engineering is building components that not only have well-defined and consistent APIs but are also reusable.

Components that are capable of working on the data of today as well as the data of tomorrow will give you the most flexible capabilities for building up large software systems.

In languages like C# and Java, one of the main tools in the toolbox for creating reusable components is generics, that is, being able 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.

Introduction to Generics

The key motivation for generics is to provide meaningful type constraints between members.

The members can be:

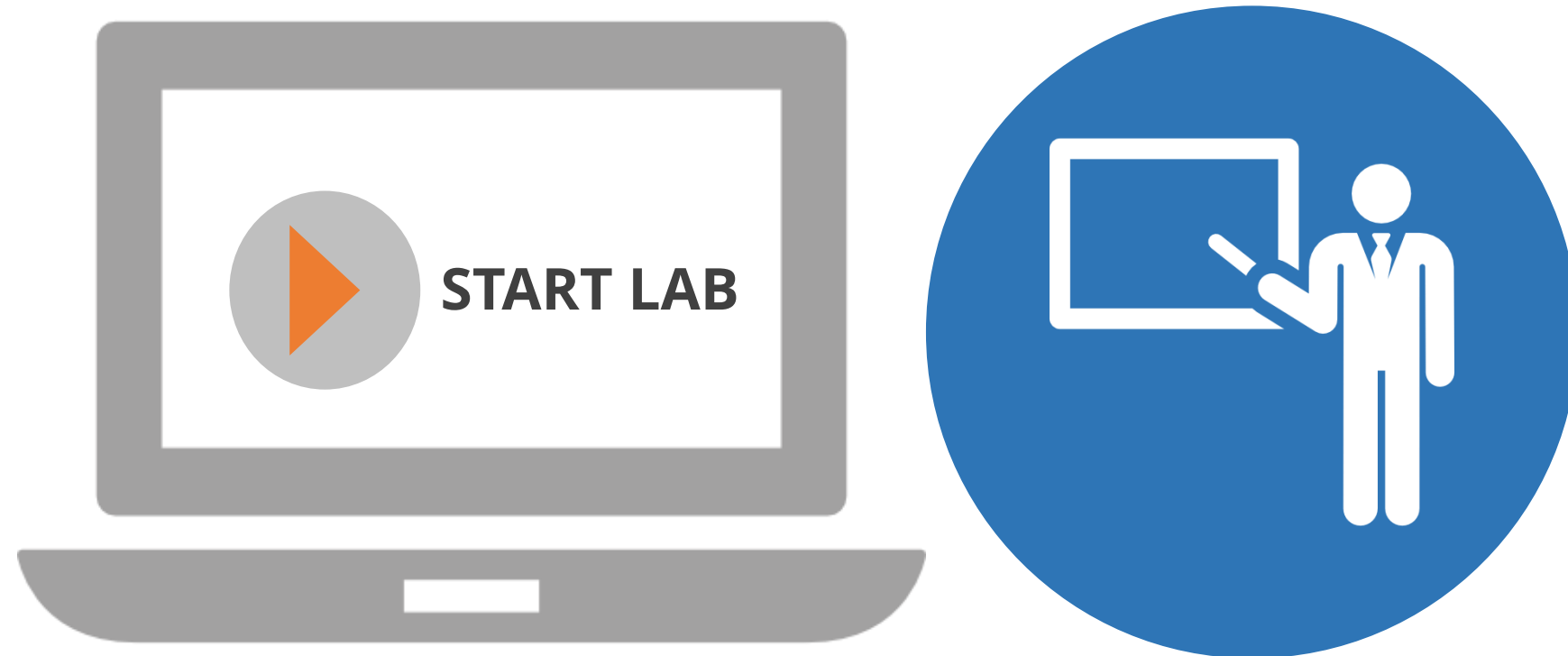
- class instance members
- class methods
- function arguments
- function return value

Generics

>_ training@localhost:~

```
class QueueNumber {  
  
    private data = [];  
    push = (item:number) => this.data.push(item);  
    pop = (): number => this.data.shift();  
}  
  
const queue = new QueueNumber();  
queue.push(0);  
queue.push("1"); //ERROR : cannot push a string. Only numbers  
allowed  
  
// ^ if that is fixed the rest would be fine too
```

Lab—Demo



Topic 6—Object Destructuring

Topic 6—Object Destructuring

Introduction to Object Literal

- ✓ ECMAScript 6 makes declaring object literals even more succinct by providing shorthand syntax for initializing properties from variables and defining function methods.
- ✓ It also enables the ability to have computed property keys in an object literal definition.
- ✓ The object literal may very well be one of the best and the most popular features in JavaScript.
- ✓ It's a superset of JSON, which has now become the de facto standard for data transport on the web, quickly replacing XML.
- ✓ The great thing about object literals is that they make it easy to assemble an arbitrarily nested and dynamic data object with a definition syntax that is human-readable because it's so succinct.

Lab—Demo



Demo

```
>_ training@localhost:~  
  
var age = 25;  
  
function printHello () {  
    console.log("Hello");  
}  
  
var person = {  
    name: 'Admin',  
    age,  
    printHello,  
    ['hello' + (age + 2)]: 'I am here',  
    printAge () {  
        console.log(this.age);  
    }  
};  
  
// console.log(person.hello27);  
person.printAge();
```

Introduction to Destructuring

- TypeScript supports the following forms of Destructuring (literally named after de-structuring, that is, breaking up the structure):
 - Object Destructuring
 - Array Destructuring
- It is easy to think of destructuring as an inverse of structuring. The method of structuring in JavaScript is the object literal.

A terminal window with a title bar showing 'training@localhost:~' and three window control buttons (red, orange, green). The terminal content shows a TypeScript object literal being assigned to a variable 'foo'.

```
>_ training@localhost:~  
  
Var foo = {  
    bar: {  
        bas: 123  
    }  
};
```

Object Destructuring

Destructuring is useful because it allows you to do in a single line, what would otherwise require multiple lines.

Consider the following case:

```
>_ training@localhost:~  
  
var rect = {x:0, y:10, width:15, height:20};  
  
//Destructuring assignment  
  
var {x,y,width,height} = rect;  
console.log(x,y,width,height); //0,10,15,20
```

Array Destructuring

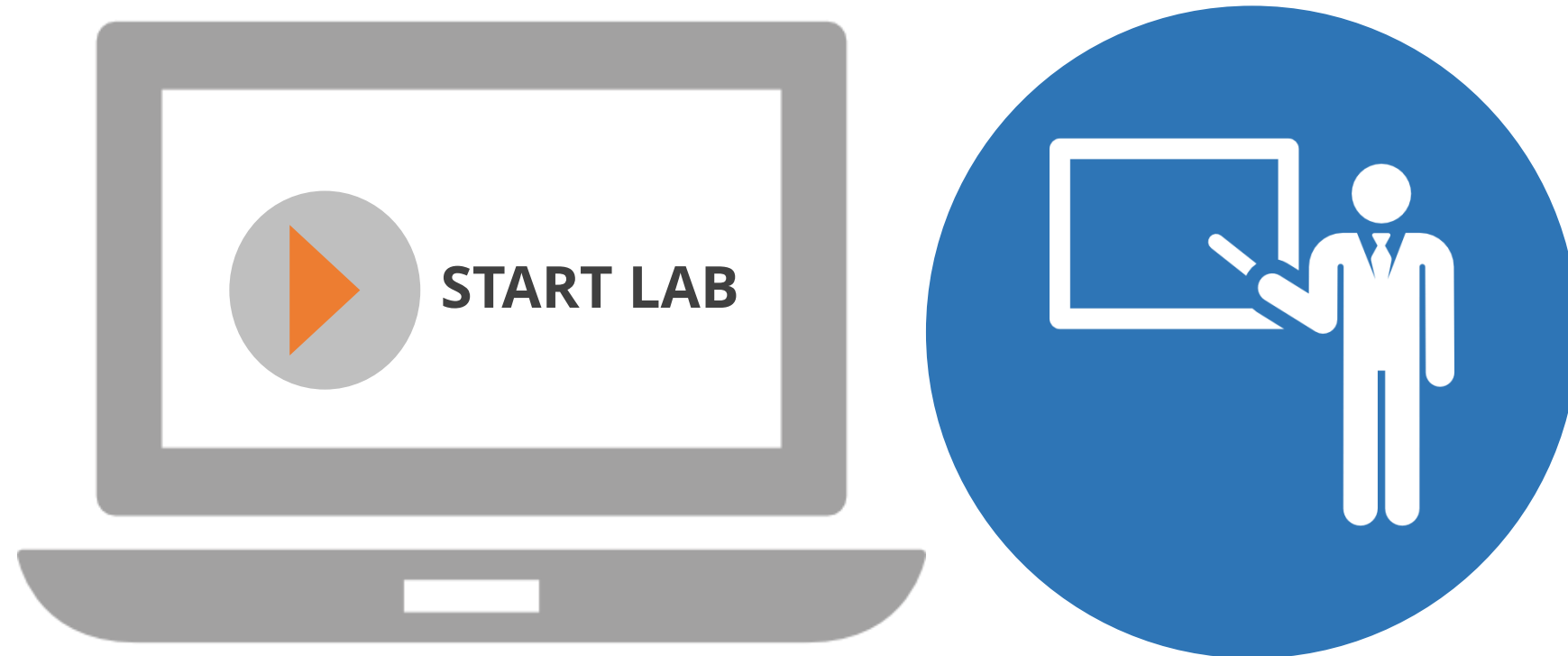
- A common programming question: "How to swap two variables without using a third one?"
- The TypeScript solution:

```
training@localhost:~  
  
var x = 1, y = 2;  
[x, y] = [y, x];  
console.log(x, y); // 2,1
```

- Array Destructuring with rest:

```
training@localhost:~  
  
var [x, y, ...remaining] = [1, 2, 3, 4];  
  
console.log(x, y, remaining); //1, 2, [3,4]
```

Lab—Demo



Introduction to TypeScript Module System

Modules provide the possibility to group related logic, encapsulate it, structure your code, and prevent pollution of the global namespace.

Modules can provide functionality that is only visible inside the module, and they can provide functionality that is visible from the outside using the export keyword.

Modules are declarative; the relationships between modules are specified in terms of imports and exports at the file level.

In TypeScript, just as in ECMAScript 2015, any file containing a top-level import or export is considered a module.

Exporting a Declaration

Any declaration (such as a variable, function, class, type alias, or interface) can be exported by adding the export keyword.

```
training@localhost:~  
  
export interface StringValidator {  
    isAcceptable(s: string) : Boolean;  
}
```

```
training@localhost:~  
  
export const numberRegex = /^ [0-9] +$/;  
  
export class ZipCodeValidator implements StringValidator {  
    isAcceptable(s: string) {  
        return s.length === 5 && numberRegex.test (s);  
    }  
}
```


Lab—Demo



Topic 7—Maps and Sets

Topic 7—Maps and Sets

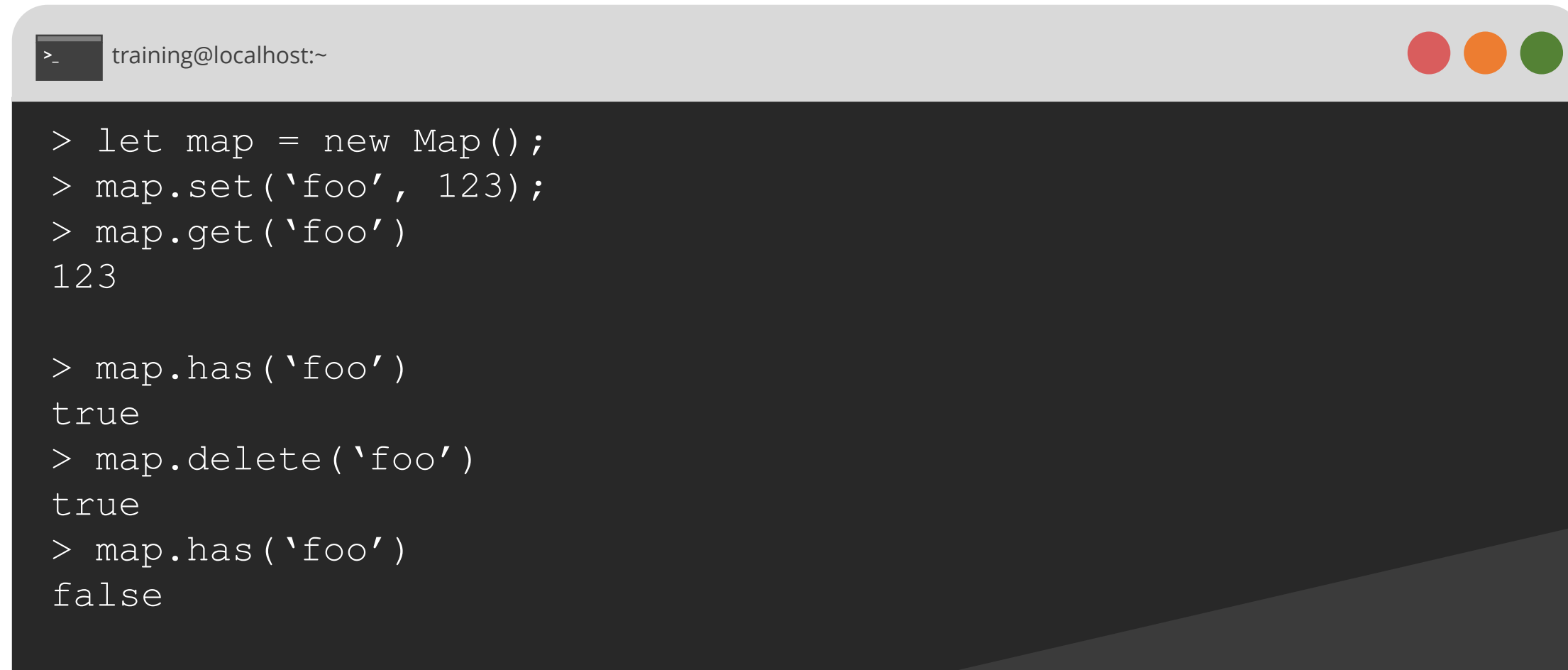
Map



- ✓ JavaScript has always had a very spartan standard library. But, a data structure for mapping values to values is not available.
- ✓ The best you can get in ECMAScript 5 is a map from strings to arbitrary values, by abusing objects. Even then there are several pitfalls that can trip you up.
- ✓ The Map data structure in ECMAScript 6 lets you use arbitrary values as keys.

Basic Operations

Working with single entries:

A terminal window with a light gray title bar. The title bar contains a prompt icon (a small box with '>_') and the text 'training@localhost:~'. On the right side of the title bar are three colored window control buttons: red, orange, and green. The main area of the terminal is dark gray and contains the following text:

```
> let map = new Map();  
> map.set('foo', 123);  
> map.get('foo')  
123  
  
> map.has('foo')  
true  
> map.delete('foo')  
true  
> map.has('foo')  
false
```

Set

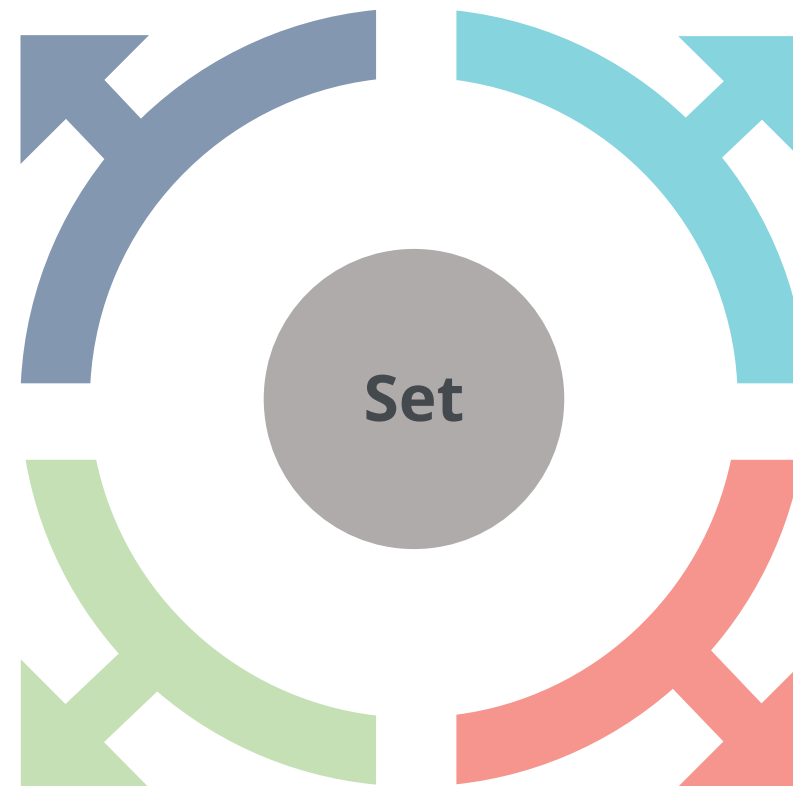
ECMAScript 5 doesn't have a set data structure, either. There are two possible work-arounds:

- Use the keys of an object to store the elements of a set of strings.
- Store (arbitrary) set elements in an array: Check whether it contains an element via `indexOf()`, remove elements via `filter()`, etc.
- This is not a very fast solution, but it's easy to implement. One issue to be aware of is that `indexOf()` can't find the value `NaN`.
- ECMAScript 6 has the data structure `Set`, which works for arbitrary values, is fast and handles `NaN` correctly.

Set (Contd.)

ECMAScript 6 has the data structure Set which works for arbitrary values, is fast, and handles NaN correctly.

This is not a very fast solution, but it's easy to implement. One issue to be aware of is that `indexOf()` can't find the value NaN.



Use the keys of an object to store the elements of a set of strings.

Store (arbitrary) set elements in an array:

Check whether it contains an element via `indexOf()`, remove elements via `filter()`, etc.

Setting up a Set

- You can set up a set via an iterable over the elements that make up the set. For example, via an array:

```
let set = new Set(['red', 'green', 'blue']);
```

- Alternatively, the add method is chainable:

```
let set = new Set().add('red').add('green').add('blue');
```

A terminal window with a title bar showing 'training@localhost:~' and three window control buttons (red, orange, green). The terminal content shows a series of JavaScript commands and their outputs: creating a set, adding 'red', checking if 'red' is in the set (returns true), deleting 'red', and checking again (returns true).

```
>_ training@localhost:~  
  
> let set = new Set();  
> set.add('red')  
  
> set.has('red')  
true  
> set.delete('red')  
true  
> set.has('red')  
true
```

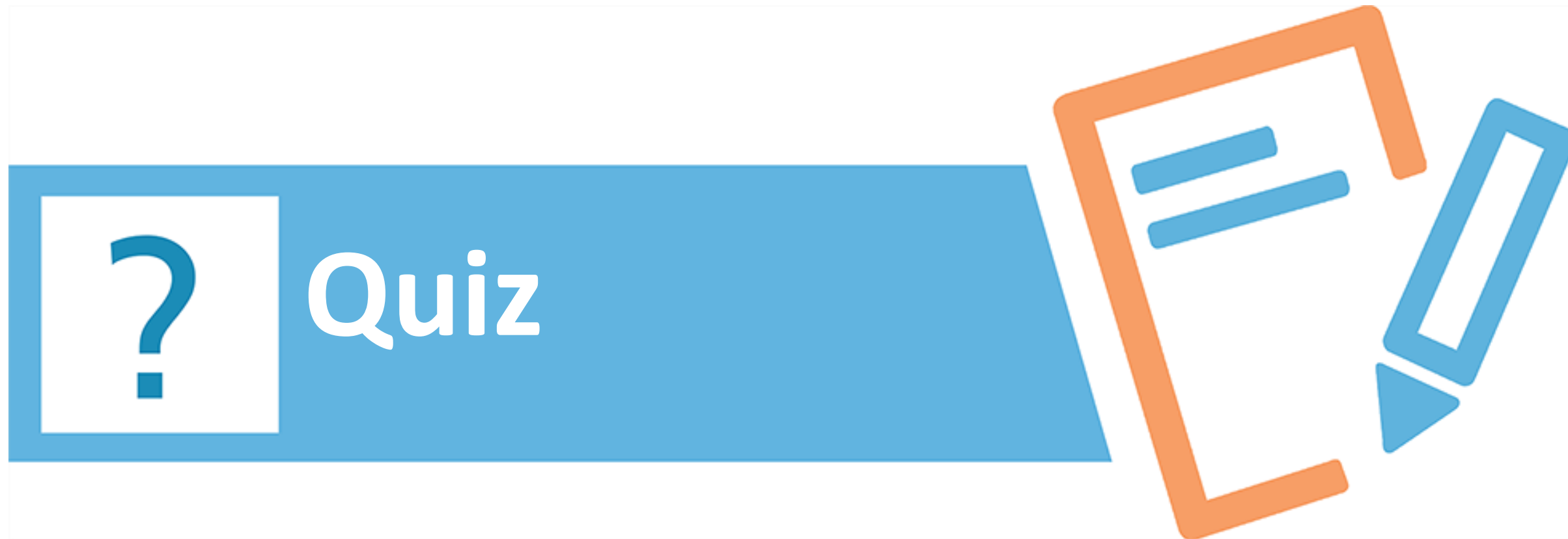
Lab—Demo



Key Takeaways



- ✔ Typescript is ES-6 implementation, optionally typed, and a Superset of JavaScript.
- ✔ Nodemon is a utility that will monitor for any changes in your source and automatically restart your server.
- ✔ Strings Template introduce a completely different way of solving these problems.
- ✔ *let* is block-scoped variables and are not visible outside of their nearest containing block or for-loop. *const* allows you to be immutable with variables.
- ✔ Classes offer a useful structural abstraction. Interfaces fill the role of naming these types, and are used as a powerful way of defining contracts within your code as well as contracts with code outside of your project.
- ✔ Set uses the keys of an object to store the elements of a set of strings.
- ✔ The Map and data structure in ECMAScript 6 lets you use arbitrary values as keys.



**QUIZ
1**

Which is NOT true about TypeScript?

- a. It is interpreted like JavaScript
- b. It is a superset of JavaScript
- c. It does support static data types
- d. TypeScript is case sensitive



**QUIZ
1**

Which is NOT true about TypeScript?

- a. It is interpreted like JavaScript
- b. It is a superset of JavaScript
- c. It does support static data types
- d. TypeScript is case sensitive



The correct answer is **a.**

TS translate into JS using the interpreter by Babel or TSC.

**QUIZ
2**

What will be output of :
`var a:string=47; console.log("Value of a= "+a);`

- a. Value of a=47
- b. Value of a=0
- c. Value of a=
- d. None of the above



**QUIZ
2**

What will be output of :
`var a:string=47; console.log("Value of a= "+a);`

- a. Value of a=47
- b. Value of a=0
- c. Value of a=
- d. None of the above



The correct answer is **d.**

There is an error in line 1 of the code.

QUIZ
3

```
class Bird {  
  Bird () { console.log("I am Bird"); }  
}
```

What will be output of : `var obj=new Bird();`?

- a. Error
- b. I am Bird
- c. Bird am I
- d. None of the above



QUIZ
3

```
class Bird {  
  Bird () { console.log("I am Bird"); }  
}
```

What will be output of : `var obj=new Bird();`?

- a. Error
- b. I am Bird
- c. Bird am I
- d. None of the above



The correct answer is **d.**

Constructor keyword is used to built constructor and not a function with same name as class.

QUIZ
4

Output of: `class MyClass {
var a=12; }
var obj=new MyClass(); console.log("Value of a= "+a);`

- a. Error
- b. Value of a =12
- c. A is private and cannot be accessed
- d. None of the above



QUIZ
4

Output of: class MyClass {
var a=12; }
var obj=new MyClass(); console.log("Value of a= "+a);

- a. Error
- b. Value of a =12
- c. A is private and cannot be accessed
- d. None of the above



The correct answer is **a.**

var cannot be used inside class.

QUIZ
5

Program that converts a code from one high level language to another high level language is called_____.

- a. Compiler
- b. Decompiler
- c. Inter translator
- d. None of the above



**QUIZ
5**

Program that converts a code from one high level language to another high level language is called_____.

- a. Compiler
- b. Decompiler
- c. Inter translator
- d. None of the above



The correct answer is **d.**

Program that converts a code from one high level language to another high level language is called Transpiler.



Thank You