

Perhaps it is better to read this document on Google Docs by following this link:
https://docs.google.com/document/d/1QHpngKjSIMkhRXKkRfj4ugPTeBVEGxOh6A8_OEBSWZY/edit?usp=sharing (Microsoft Word ruins parts of the text editing and code example readability)

TypeScript

(Разработка на TypeScript)

*We love TypeScript for many things... With TypeScript, several of our team members have said things like '**I now actually understand most of our own code!**' because they can easily traverse it and understand relationships much better. And we've **found several bugs via TypeScript's checks.**'*

— Brad Green, Engineering Director - AngularJS

Made by KNI-A students

Filip Markoski (161528)
Kristofer Mladenovski (161530)

Abstract

In this project, we will be analyzing client-side programming languages and specifically JavaScript, the most popular one by far, while pointing out its many flaws and reasons as to why you should try out TypeScript, a similar client-side programming language which tries to fix many of JavaScript's faults while keeping its benefits. The basics of TypeScript will be discussed, its advantages over JavaScript, how to make your own TypeScript file and why you would want to use certain elements from TypeScript.

Contents

Introduction

Client-side programming languages

Javascript	5
CoffeeScript (source)	6
TypeScript (source)	6
Elm (source)	7
ClojureJS (source)	8

What is Transpiling (source) 8

ECMAScript 6 (ES6)	9
ECMAScript 2017 (ES8)	10
Babel	10
Compilation context	12

Why use TypeScript? (source) 14

Optional Typing	16
-----------------	----

Why use Types? 16

What does static typing offer?	16
TypeScript Makes Code Easier to Read and Understand	17
A few words and examples regarding the typing	18
When to use TypeScript	19

Installation Process 20

Making your first TypeScript file	20
Additional notes on transpiling, joining files and watcher. (source)	21

Basic Types in TypeScript (source) 22

Boolean	22
Number	22
String	22
Array	23

Tuple	23
Enum	23
Any	24
Void	24
Null and Undefined	24
Never	25
Type Assertions	25
Keywords and Features in TypeScript	26
let	26
Functions create a new scope	27
The Generated JS	27
let in Closures	28
const	29
Block Scoped	30
Deep immutability	30
for...of	31
JS Generation	31
Limitations	32
Multiline Strings	32
Arrow Functions	32
A few helpful Tips	34
Tip: Arrow Function Need	34
Tip: Arrow Function Danger	35
Tip: Arrow functions with libraries that use this	35
Tip: Arrow functions and inheritance	35
Code in JavaScript vs code in TypeScript	35
Pitfalls of Javascript (source)	38
Equality Operator	38
Object Usage and Properties	39
Objects as a Data Type	39
Accessing Properties	39
Deleting Properties	40
Notation of Keys	40
The Prototype	41
Property Lookup	42
Performance	42
Extension of Native Prototypes	42
hasOwnProperty	43
Array type	43
Optional Semicolons	44

Functional Programming (JavaScript)	45
A little bit about Closures (source)	46
Closures have access to the outer function's variable even after the outer function returns:	46
Closures store references to the outer function's variables	47
Closures Gone Awry	47
Reason why Closures are awesome	49
Functions as Units of Behavior	49
Object-oriented Programming in TypeScript (source)	52
Type annotations	55
Classes	56
Running your TypeScript web app	57
Libraries and frameworks that use TypeScript	57
Angular (source)	57
Visual Studio Code (source)	58
Our own implementation of TypeScript	59
Conclusion	64
Literature	65

*"One of Ionic's main goals is to **make app development as quick and easy as possible**, and the **tooling support TypeScript gives us with autocompletion, type checking and source documentation** really aligns with that."*

— Tim Lancina, Tooling Developer - Ionic

Introduction

JavaScript is notorious for being the world's quirkiest programming language. It is often correctly derided as being a toy, but beneath its layer of inconsistencies and disorganization, powerful language features await. JavaScript is now used by an incredible number of high-profile applications, showing that deeper knowledge of this technology is an important skill for any web or mobile developer.

Because of certain pitfalls mentioned in this document things such as Babel, CoffeeScript, TypeScript and many other tools have become relevant because they offer a more straightforward way of creating and developing software which is in big part a JavaScript project. JavaScript relates to a vast field of topics, for example, the language itself, how it can be used, what else can it be used for (client-side and server-side)? How it has evolved over the years and so on.

JavaScript is by far the most popular web programming language so it's only natural that developers have invested time and effort into creating tools which very closely relate to the production process and to the main aim of their certain JavaScript projects. In this document we give an overview of some of those tools which are available for free, we also examine JavaScript and its weaknesses and how certain tools have dealt with these shortcomings. Afterwards we focus on explaining TypeScript, we cover the installation process, the type notation it offers, when and why use TypeScript, we also explain the types it offers the additional keywords it uses, how it related to newer versions of ECMAScript, we mention how popular frameworks and ourselves have implemented TypeScript to serve a certain purpose and we have mentioned a few interesting details along the way, such as, what is a closure in JavaScript, what's the difference between the `==` and the `===` operator and so on.

Additionally, we have offered links and sources for the purpose of additional reading and verification of the information presented in this document.

Client-side programming languages

Client-side programming is the name of the program that runs entirely on the client. Client-side programming mostly deals with the user interface which the user interacts with which the user interacts in the web. The browser on the user's machine runs the code and it is usually done in scripting languages like JavaScript, CoffeeScript, TypeScript, Elm and ClojureJS.

JavaScript

JavaScript is a high-level, dynamic, untyped, interpreted run-time language. Alongside HTML and CSS, JavaScript is one of the three core technologies of World Wide Web content production; most websites employ it, and all modern Web browsers support it without the need for plug-ins. JavaScript is prototype-based with first-class functions, making it a multi-paradigm

language, supporting object-based, imperative, and functional programming styles. Here is an example of JavaScript code:

CoffeeScript ([source](#))

CoffeeScript is a programming language that transcompiles to JavaScript. It adds syntax similar to Python and Ruby to make JavaScript more readable and succinct. Specific additional features include list comprehension and pattern matching. Almost everything is an expression in CoffeeScript, for example `if`, `switch` and `for` expressions (which have no return value in JavaScript) return a value. The control statements in CoffeeScript also have postfix versions; for example, `if` can also be written after the conditional statement. Here is a code example for CoffeeScript:

```
# Assignment:
number = 42
opposite = true

# Conditions:
number = -42 if opposite

# Functions:
square = (x) -> x * x

# Arrays:
list = [1, 2, 3, 4, 5]

# Objects:
math =
  root: Math.sqrt
  square: square
  cube: (x) -> x * square x

# Splats:
race = (winner, runners...) ->
  print winner, runners

# Existence:
alert "I knew it!" if elvis?

# Array comprehensions:
cubes = (math.cube num for num in list)
```

TypeScript ([source](#))

TypeScript, like CoffeeScript, is a language that transcompiles into JavaScript. It adds static typing and class based object oriented programming to JavaScript. We will delve into TypeScript more later on. Here is an example of TypeScript:

```

interface GreetingSettings {
  greeting: string;
  duration?: number;
  color?: string;
}

declare function greet(setting: GreetingSettings): void;

function getGreeting() {
  return "howdy";
}

class MyGreeter extends Greeter { }

greet("hello");
greet(getGreeting);
greet(new MyGreeter());

```

Elm ([source](#))

Elm, like the before mentioned languages, transpiles into JavaScript. The biggest difference between elm and JavaScript is that there are no runtime exception in elm, instead elm uses type inference to detect problems during compilation. It also has excellent performance. Here is an example:

```

import Html exposing (text)
import String

capitalize1 word =
  String.toUpper (String.left 1 word) ++ String.dropLeft 1 word

capitalize2 word =
  let
    firstLetter =
      String.left 1 word

    otherLetters =
      String.dropLeft 1 word
  in
    String.toUpper firstLetter ++ otherLetters

main =
  text (capitalize2 "who ate all the pie?")

```

ClojureJS ([source](#))

Clojure is a dynamic, general-purpose programming language supporting interactive development. Clojure is a functional programming language featuring a rich set of immutable, persistent data structures similar to Lisp, as it is a dialect of Lisp. Here is an example of Clojure code:

```
(defn foo
  [& {:keys [bar baz]
      :or {bar "default1"
            baz "default2"}}]
  (str bar "-" baz))

(foo) ;; => "default1-default2"
(foo :bar 1) ;; => "1-default2"
(foo :bar 1 :baz 2) ;; => "1-2"
```

What is Transpiling ([source](#))

Even though not many people know about transpiling, the term and the process have been around for quite some time now. To avoid any confusion that there might be regarding the difference between the terms transpiling and compiling is we will firstly define the two terms. In short: Compiling is, in a general sense, transforming source code written in one language into code written in another language, with the main difference being that the two languages used differ in the level of abstraction that they hold. On the other hand, transpiling is a more specific term which means taking source code written in one language and transforming it into another language but with the same or similar level of abstraction.

For the sake of example, when we do

```
tsc index.ts
```

The TypeScript transpiler is actually taking our TypeScript code written in the file index.ts and transforming it into a JavaScript file with the same name but different extension, e.g. index.js.

Both compilers and transpilers can optimise the code as part of the process.

Other common combinations that can be dubbed as transpiling include C++ to C, CoffeeScript to JavaScript, Dart to JavaScript and PHP to C++.

Some of the more popular languages using transpilers are TypeScript, Babel and CoffeeScript, with all of these we can understand that everything that we write in JavaScript we can write in TypeScript, Babel or CoffeeScript. For example:

JavaScript:


```

"use strict";

// Good 'ol JS
function printSecret ( secret ) {
  console.log(`${secret}. But don't tell anyone.`);
}

printSecret("I don't like CoffeeScript");

```

CoffeeScript:

```

"use strict"

# CoffeeScript
printSecret (secret) =>
  console.log `${secret}. But don't tell anyone.`

printSecret "I don't like JavaScript."

```

TypeScript:

```

"use strict";

// TypeScript -- JavaScript, with types and stuff
function printSecret ( secret : string ) {
  console.log(`${secret}. But don't tell anyone.`);
}

printSecret("I don't like CoffeeScript.");

```

Trying those last two examples in your console will throw errors. As a matter of fact, if you try that pure JavaScript example in an older browser, you'll still get an error. That's where transpilers come in: They read CoffeeScript, TypeScript, and ES2015, and spit out JavaScript guaranteed to work anywhere.

Why learn new syntax and pick up new tools if all we get at the end of the day is the JavaScript we could have written in the first place? In the case of languages that target JavaScript, it's largely a matter of preference or background. Writing in a language that "thinks" the way you do makes you more productive. People with backgrounds in OOP often like TypeScript because it's familiar territory. Pythonistas like CoffeeScript. Clojurists write ClojureScript, and so on. ([source](#))

ECMAScript 6 (ES6)

ECMAScript 6, also known as ECMAScript 2015, is one of the versions of the ECMAScript standard. For the sake of brevity, we will only highlight the most important features that ES6 brings to the JavaScript world:

ES6 includes the following new features:

1. [arrows](#)
2. [classes](#)

3. [enhanced object literals](#)
4. [template strings](#)
5. [destructuring](#)
6. [default + rest + spread](#)
7. [let + const](#)
8. [iterators + for..of](#)
9. [generators](#)
10. [unicode](#)
11. [modules](#)
12. [module loaders](#)
13. [map + set + weakmap + weakset](#)
14. [proxies](#)
15. [symbols](#)
16. [subclassable built-ins](#)
17. [promises](#)
18. [math + number + string + array + object APIs](#)
19. [binary and octal literals](#)
20. [reflect api](#)
21. [tail calls](#)

ECMAScript 2017 (ES8)

To go into all of these features we would need further additional chapters perhaps for every single one of them, so we have decided to just highlight the features.

Major new features:

- [Async Functions \(Brian Terlson\)](#)
- [Shared memory and atomics \(Lars T. Hansen\)](#)

Minor new features:

- [Object.values/Object.entries \(Jordan Harband\)](#)
- [String padding \(Jordan Harband, Rick Waldron\)](#)
- [Object.getPrototypeOfDescriptors\(\) \(Jordan Harband, Andrea Giammarchi\)](#)
- [Trailing commas in function parameter lists and calls \(Jeff Morrison\)](#)

Babel

Babel is a transpiler for JavaScript best known for its ability to turn ES6 (the next version of JavaScript) into code that runs in your browser (or on your server) today. For example, the following ES6 code:

```
const input = [1, 2, 3];
console.log(input.map(item => item + 1)); // [2, 3, 4]
```

is compiled by Babel to:

```
var input = [1, 2, 3];
console.log(input.map(function (item) {
  return item + 1;
})); // [2, 3, 4]
```

Which runs in just about any JavaScript environment.

Babel was created by an extremely dedicated Australian developer called Sebastian McKenzie who has worked tirelessly to ensure that Babel can handle all of the new syntax that ES6 brings, along with built-in support for React's JSX extensions and Flow type annotations. (Babel also makes use of an excellent set of polyfills called core-js, written by a talented Russian software engineer called Denis Pushkarev, and is built upon the very fast acorn JavaScript parser by Marijn Haverbeke and Ingvar Stepanyan.)

Of all the ES6 transpilers, Babel has the greatest level of compatibility with the ES6 spec, even exceeding the much longer established Traceur by Google.

Babel lets you use virtually all of the new features that ES6 brings today, without sacrificing backwards compatibility for older browsers. It also has first class support for dozens of different build & test systems which makes integration with your current toolchain very simple.

If we stopped here, Babel would already be pretty compelling, unless perhaps you're already using an alternative compile-to-JS language, like CoffeeScript or TypeScript.

In fact, Babel is often compared to CoffeeScript. They are both transpilers after all, and since CoffeeScript influenced the design of ES6, they have a lot in common in terms of syntax (minus of course things like significant whitespace).

Typescript brings other benefits, like type safety and a whole host of other language features, but it's not quite JavaScript and doesn't play nicely with technologies like JSX. If you use TypeScript, you can't really use anything other than TypeScript, and if you do, you lose many of the benefits that TypeScript provides, e.g. type safety.

If you're already using CS or TS you might be tempted to stick with what you already know, however, the similarities between them are quite superficial, where CoffeeScript and TypeScript are merely languages, Babel is much more than that, fundamentally Babel is a platform.

Babel doesn't only track ES6, it also tracks the next version of JavaScript - ES7 (or ES2016). Amongst other things this brings support for async/await. This is the much-simplified code for recursively reading a directory.:

```
async function walk (dir, visitor) {
  const filenames = await fs.readdirAsync(dir);
  await* filenames.map(async _filename => {
    const filename = path.join(dir, _filename);
    const stat = await fs.statAsync(filename);
```

```

    if (stat && stat.isDirectory()) {
      await walk(filename, visitor);
    } else {
      visitor(filename, stat);
    }
  });
}

```

Babel's plugin system is its secret weapon, and this is where the ecosystem around Babel will really take off. Babel allows custom code transformers to be plugged in to the compilation process. These code transformers take an AST (Abstract Syntax Tree, a collection of nested objects which describes the structure of your program) and perform manipulations upon it before it is turned into executable JavaScript.

Compilation context

The compilation context refers to the grouping of the files that TypeScript will parse and analyze to determine what is valid and what isn't. Along with the information about which files, the compilation context contains information about which compiler options. A great way to define this logical grouping (we also like to use the term project) is using a `tsconfig.json` file.

You have two options when it comes to the `tsconfig.json` file, you can either write it yourself or you can generate it using the command `tsc --init` which will initialize the folder in which we are present as a TypeScript project and will generate the `tsconfig.json`. Of course, we can just have an open and closing bracket to serve as our `tsconfig.json` after all the is technically an empty JSON object.

```
{}
```

The compiler, or rather the transpiler will choose to follow its default options such as transpiling all TypeScript documents, meaning all files with the `.ts` extension, it will transpile them into the default target which is ECMAScript 5, it will allow plain old JavaScript to be seen as valid if written in the given TypeScript files et cetera.

To customize or specify the compiler options we can use the `compilerOptions` which should be written in the `tsconfig.json` file. For example:

```

{
  "compilerOptions": {

    /* Basic Options */
    "target": "es5", /* Specify ECMAScript target version: 'ES3'
    (default), 'ES5', 'ES2015', 'ES2016', 'ES2017', or 'ESNEXT'. */
    "module": "commonjs", /* Specify module code generation: 'commonjs',
    'amd', 'system', 'umd' or 'es2015'. */
    "lib": [], /* Specify library files to be included in the
    compilation: */
    "allowJs": true, /* Allow javascript files to be compiled. */
    "checkJs": true, /* Report errors in .js files. */
    "jsx": "preserve", /* Specify JSX code generation: 'preserve', 'react-
    native', or 'react'. */
    "declaration": true, /* Generates corresponding '.d.ts' file. */
  }
}

```

```

    "sourceMap": true,                /* Generates corresponding '.map' file. */
    "outFile": "./",                 /* Concatenate and emit output to single file. */
    "outDir": "./",                  /* Redirect output structure to the directory. */
    "rootDir": "./",                 /* Specify the root directory of input files. Use
to control the output directory structure with --outDir. */
    "removeComments": true,          /* Do not emit comments to output. */
    "noEmit": true,                   /* Do not emit outputs. */
    "importHelpers": true,            /* Import emit helpers from 'tslib'. */
    "downlevelIteration": true,       /* Provide full support for iterables in 'for-of',
spread, and destructuring when targeting 'ES5' or 'ES3'. */
    "isolatedModules": true,          /* Transpile each file as a separate module
(similar to 'ts.transpileModule'). */

    /* Strict Type-Checking Options */
    "strict": true,                   /* Enable all strict type-checking options. */
    "noImplicitAny": true,            /* Raise error on expressions and declarations with
an implied 'any' type. */
    "strictNullChecks": true,         /* Enable strict null checks. */
    "noImplicitThis": true,           /* Raise error on 'this' expressions with an
implied 'any' type. */
    "alwaysStrict": true,             /* Parse in strict mode and emit "use strict" for
each source file. */

    /* Additional Checks */
    "noUnusedLocals": true,           /* Report errors on unused locals. */
    "noUnusedParameters": true,       /* Report errors on unused parameters. */
    "noImplicitReturns": true,         /* Report error when not all code paths in function
return a value. */
    "noFallthroughCasesInSwitch": true, /* Report errors for fallthrough cases in switch
statement. */

    /* Module Resolution Options */
    "moduleResolution": "node",        /* Specify module resolution strategy: 'node'
(Node.js) or 'classic' (TypeScript pre-1.6). */
    "baseUrl": "./",                  /* Base directory to resolve non-absolute module
names. */
    "paths": {},                       /* A series of entries which re-map imports to
lookup locations relative to the 'baseUrl'. */
    "rootDirs": [],                    /* List of root folders whose combined content
represents the structure of the project at runtime. */
    "typeRoots": [],                  /* List of folders to include type definitions
from. */
    "types": [],                       /* Type declaration files to be included in
compilation. */
    "allowSyntheticDefaultImports": true, /* Allow default imports from modules with no
default export. This does not affect code emit, just typechecking. */

    /* Source Map Options */
    "sourceRoot": "./",               /* Specify the location where debugger should
locate TypeScript files instead of source locations. */
    "mapRoot": "./",                  /* Specify the location where debugger should
locate map files instead of generated locations. */

```

```

    "inlineSourceMap": true,                /* Emit a single file with source maps instead of
having a separate file. */
    "inlineSources": true,                 /* Emit the source alongside the sourcemaps within
a single file; requires '--inlineSourceMap' or '--sourceMap' to be set. */

    /* Experimental Options */
    "experimentalDecorators": true,        /* Enables experimental support for ES7 decorators.
*/
    "emitDecoratorMetadata": true          /* Enables experimental support for emitting type
metadata for decorators. */
  }
}

```

Additionally, we can be explicit when telling the transpiler which files we want to transpile by telling so in “files”, for example:

```

{
  "files": [
    "./some/file.ts"
  ]
}

```

Or we can use “include” and “exclude” to be even more specific in our file choice:

```

{
  "include": [
    "./folder"
  ],
  "exclude": [
    "./folder/**/*.spec.ts",
    "./folder/someSubFolder"
  ]
}

```

Default options are to transpile all TypeScript files.

TypeScript

Why use TypeScript? ([source](#))

[TypeScript](#) is ECMAScript 6 (ES6) with optional typing.

[ECMAScript](#) is the standardized specification for the JavaScript language. It’s sometimes referred as ECMAScript Harmony or ES.next. At the time of this writing, JavaScript is currently at ECMAScript 8, which is not yet finished ([source](#)). Since ES6 is backwardly compatible with the ES5 syntax, you can start writing TypeScript without changing any of your existing code. The good thing is that learning TypeScript doesn’t require that the programmer learn a whole new syntax, as one may would using CoffeeScript for example.

TypeScript Has Great Tools

The biggest selling point of TypeScript is tooling. It provides advanced autocompletion, navigation, and refactoring. Having such tools is almost a requirement for large projects. Without them the fear changing the code puts the code base in a semi-read-only state, and makes large-scale refactorings very risky and costly.

TypeScript is not the only typed language that compiles to JavaScript. There are other languages with stronger type systems that in theory can provide absolutely phenomenal tooling. But in practice most of them do not have anything other than a compiler. This is because building rich developer tools has to be an explicit goal from day one, which it has been for the TypeScript team. That is why they built language services that can be used by editors to provide type checking and autocompletion. If you have wondered why there are so many editors with great TypeScript supports, the answer is the language services.

The fact that intellisense and basic refactorings (e.g., rename a symbol) are reliable makes a huge impact on the process of writing and especially refactoring code.

TypeScript Makes Abstractions Explicit

A good design is all about well-defined interfaces. And it is much easier to express the idea of an interface in a language that supports them.

As you can see, both classes play the role of a purchaser. Despite being extremely important for the application, the notion of a purchaser is not clearly expressed in the code. There is no file named purchaser.js. And as a result, it is possible for someone modifying the code to miss the fact that this role even exists.

```
function processPurchase(purchaser, details){ }

class User { }

class ExternalSystem { }
```

It is hard, just by looking at the code to tell what objects can play the role of a purchaser, and what methods this role has. We do not know for sure, and we will not get much help from our tools. We have to infer this information manually, which is slow and error-prone.

Now, compare it with a version where we explicitly define the Purchaser interface.

```
interface Purchaser {id: int; bankAccount: Account;}

class User implements Purchaser {}

class ExternalSystem implements Purchaser {}
```

The typed version clearly states that we have the Purchaser interface, and the User and ExternalSystem classes implement it. So TypeScript interfaces allow us to define abstractions/protocols/roles.

It is important to realize that TypeScript does not force us to introduce extra abstractions. The Purchaser abstraction is present in the JavaScript version of the code, but it is not explicitly defined.

In a statically-typed language, boundaries between subsystems are defined using interfaces. Since JavaScript lacks interfaces, boundaries are not well expressed in plain JavaScript. Not being able to clearly see the boundaries, developers start depending on concrete types instead of abstract interfaces, which leads to tight coupling.

My experience of working on Angular before and after our transition to TypeScript reinforced this belief. Defining an interface forces me to think about the API boundaries, helps me define the public interfaces of subsystems, and exposes incidental coupling.

Optional Typing

Where TypeScript comes into its own is in its optional typing. A typed language like Java, C# or Objective-C requires you to specify the type of the variable when declaring it.

Declaring a string in Java would look like: `String name = "Andrew";`, in C# `string name = "Andrew";` and in Objective-C it's `NSString name = "Andrew"`.

In TypeScript it would be: `var name: string = "Andrew"`. But with it being optional, it can be just plain `var name = "Andrew"`. In strictly typed languages you have to declare everything all the time, with TypeScript you don't!

Having types in your script allows text editors and IDEs to give you intelligent hints quickly without having to run your code.

It also helps with autocompletion of your code too. Other text editors autocomplete based on the text written in your project files, it doesn't know anything about the type of each variable so you may end up passing a similarly named variable into a method call but it's the wrong type of object. However, with TypeScript, code editors can have a more intelligent approach and suggest more appropriate variables to pass into a function call. It's a real productivity boost. It's almost as if the code is writing itself.

It also reduces the need to look up documentation so frequently since the code is annotated with the types needed for a method call or what will be returned from a method call.

Why use Types?

The term dynamic typing refers to the practice where a programmer needn't declare the type of a variable when programming his code, because the variable will dynamically be assigned a type when needed. This is the practice that JavaScript follows as well as popular languages such as PHP and Ruby. On the other hand we have static typing, which is found in all major and famous languages such as Java, C++/C, C# etc. It is when the programmer is forced into declaring the type for each variable.

What does static typing offer?

Static typing most of the time leads the programmer to think along the lines of software structure, meaning he needs to understand what variable and more specifically what kind of variable can use a certain function and which cannot. Because of this reason TypeScript has

been adopted and used in major programming projects as mentioned. Some projects have completely been rewritten into TypeScript because TypeScript conveys the object-oriented programming concept much more clearly than using certain hacks or workarounds when typing plain old JavaScript. Of course, TypeScript also uses those same workarounds and hacks when transpiling the code to JavaScript but the main thing is that while the program is being created the programmer needs only to think about solving the problem and less about how to encapsulate or compose information. Of course the static typing in TypeScript is optional, but of course most programmers will use it since they want their abstractions to be explicit.

TypeScript Makes Code Easier to Read and Understand

Let's start off with an example using the function `jQuery.ajax()` from the JavaScript library jQuery. What kind of information can we get from its signature?

```
jQuery.ajax(url, settings)
```

The only thing we can tell for sure is that the function takes two arguments. We can guess the types. Maybe the first one is a string and the second one is a configuration object. But it is just a guess, and we might be wrong. We have no idea what options go into the settings object (neither their names nor their types), or what this function returns.

There is no way we can call this function without checking the source code or the documentation. Checking the source code is not a good option—the point of having functions and classes is to be able to use them without knowing how they are implemented. In other words, we should rely on their interfaces, not on their implementation. We can check the documentation, but it is not the best developer experience—it takes additional time, and the docs are often out-of-date.

So although it is easy to read `jQuery.ajax(url, settings)`, to really understand how to call this function we need to either read its implementation or its docs.

Now, contrast it with a typed version.

```
ajax(url: string, settings?: JQueryAjaxSettings): JQueryXHR;

interface JQueryAjaxSettings {
  async?: boolean;
  cache?: boolean;
  contentType?: any;
  headers?: { [key: string]: any; };
  //...
}

interface JQueryXHR {
  responseJSON?: any; //...
}
```

It gives us a lot more information.

1. The first argument of this function is a string.
2. The settings argument is optional. We can see all the options that can be passed into the function, and not only their names, but also their types. Note: the question mark

operator specifies that the given field is optional, e.g. `async?: boolean`; meaning this field isn't specifically required.

3. The function returns a `jQueryXHR` object, and we can see its properties and functions.

The typed signature is certainly longer than the untyped one, but `:string`, `:jQueryAjaxSettings`, and `jQueryXHR` are not clutter. They are important documentation that improves the comprehensibility of the code. We can understand the code to a much greater degree without having to dive into the implementation or reading the docs.

One thing that is different about TypeScript comparing to many other languages compiled to JavaScript is that its type annotations are optional, and `jQuery.ajax(url, settings)` is still valid in TypeScript. So instead of an on-off switch, TypeScript's types are more of a dial. If you find that the code is trivial to read and understand without type annotations, do not use them. Use types only when they add value.

A few words and examples regarding the typing

Dynamically-typed languages have inferior tooling, but they are more malleable and expressive. We think using TypeScript makes your code more rigid, but to a much lesser degree than people think. For example:

```
const PersonRecord = Record({name:null, age:null});

function createPerson(name, age) {
  return new PersonRecord({name, age});
}

const p = createPerson("Jim", 44);

expect(p.name).toEqual("Jim");
```

How do we type the record? Let's start with defining an interface called `Person`:

```
interface Person { name: string, age: number };

If we try to do the following:
function createPerson(name: string, age: number): Person {
  return new PersonRecord({name, age});
}
```

the TypeScript compiler will complain. It does not know that `PersonRecord` is actually compatible with `Person` because `PersonRecord` is created reflectively. TypeScript's type system is not the most advanced one. But its goal is different. It is not here to prove that the program is 100% correct. It is about giving you more information and enable greater tooling. So it is alright to take shortcuts when the type system is not flexible enough. So we can just cast the created record, as follows:

```
function createPerson(name: string, age: number): Person {
  return <any>new PersonRecord({name, age});
}
```

The typed example:

```
interface Person { name: string, age: number };

const PersonRecord = Record({name:null, age:null});

function createPerson(name: string, age: number): Person {
    return <any>new PersonRecord({name, age});
}

const p = createPerson("Jim", 44);

expect(p.name).toEqual("Jim");
```

The reason why it works is because the type system is structural. As long as the created object has the right fields, name and age, we are good.

You need to embrace the mindset that it is alright to take shortcuts when working with TypeScript. Only then you will find using the language enjoyable. For instance, don't try to add types to some funky metaprogramming code most likely you won't be able to express it statically. Type everything around that code, and tell the typechecker to ignore the funky bit. In this case you will not lose a lot of expressiveness, and the bulk of your code will remain toolable and analyzable.

The optional type system preserves the JavaScript development workflow. Large parts of your application's code base can be "broken", but you can still run it. TypeScript will keep generating JavaScript, even when the type checker complains. This is extremely useful during development.

When to use TypeScript

We believe that TypeScript should be used on projects which benefit from its use as that is the most rational choice. That would refer to projects that would be much better off with structure in the object-oriented sense. For smaller projects which use small scripts only to change a few css properties of HTML elements TypeScript is unneeded and even too verbose for small tasks such as that. Of course writing plain old JavaScript in a TypeScript file is completely alright and one can do that, but in essence TypeScript's utilities are never really used. But they are used when we have some hierarchy of classes, dependencies between information et cetera and TypeScript is the perfect candidate to sort out and structure such things.

In conclusion, we believe that small projects which use shorts scripts which do not require structure, do not require types as well and therefore would not have the specific benefit of being written in TypeScript which is the opposite for bigger projects with plenty lines of code which would greatly benefit by having structure and thus would benefit from having their programmers write TypeScript code instead of plain old JavaScript code.

Installation Process

There are two main ways to get the TypeScript tools:

1. Via npm (the Node.js package manager)
2. By installing TypeScript's Visual Studio plugins

For option 1, you will need to have NodeJS installed and alongside it npm, a package manager. You can easily install them both by following this [link](#), where we recommend that you choose to install the LTS version, recommended for most users. After NodeJS is installed you can verify your installation by typing

```
code** node -v
```

Or

```
code** node --version
```

For which you will get something like “v6.10.0” as output. You can also run “npm -v” or “npm --version” to verify that the package manager has also been installed.

Assuming you have node and npm and you’ve chosen to take on the first installation option (using npm) simply write this line in your command line:

```
code** npm install -g typescript
```

Where the -g is the global flag meaning that it’ll install TypeScript on the whole machine, if we didn’t have the -g global flag then npm will automatically assume that we want TypeScript just in the current project that we’re working in, which is actually the folder we are in at that moment inside the command line.

For option 2, Visual Studio 2017 and Visual Studio 2015 Update 3 include TypeScript by default. If you didn’t install TypeScript with Visual Studio, you can still [download it](#). In other IDEs such as WebStorm or PhpStorm TypeScript comes built-in, so you can easily just incorporate TypeScript into your projects assuming your project is suited for its use.

Making your first TypeScript file

Assuming you’ve went through with the installation process, using the editor of your choice type in the following JavaScript code in a file named “greeter.ts”, notice that the extensions is “.ts” instead of “.js”.

```
function greeter(person) {
  return "Hello, " + person;
}

var user = "Jane User";

document.body.innerHTML = greeter(user);
```

At the command line, run the TypeScript compiler:

```
code** tsc greeter.ts
```

The result will be a file greeter.js which contains the same JavaScript that we fed in.

Additional notes on transpiling, joining files and watcher. ([source](#))

The following command will compile a single .ts file into a .js file:

```
tsc app.ts
```

This will result in an app.js file being created.

To compile multiple .ts files:

```
tsc app.ts another.ts someMore.ts
```

This will result in 3 files, app.js, another.js and someMore.js.

You can also use wildcards too. The following command will compile all TypeScript files in the current folder.

```
tsc *.ts
```

All TypeScript files will compile to their corresponding JavaScript files.

You can also compile all your TypeScript files down to a single JavaScript file. This can reduce the number of HTTP requests a browser has to make and improve performance on HTTP 1.x sites. To do this use the --out option like so:

```
tsc *.ts --out app.js
```

Instead of running the tsc command all the time you can use the option --watch.

```
tsc *.ts --out app.js --watch
```

Every time there's an update to a TypeScript file it'll recompile the source files to JavaScript.

If you're using a wildcard like this, any new files created since running the tsc command won't get compiled, you need to stop the watcher and start again.

Now we can start taking advantage of some of the new tools that TypeScript offers. Add a : string type annotation to the 'person' function argument as shown here:

```
function greeter(person: string) {  
    return "Hello, " + person;  
}
```

```
var user = "Jane User";

document.body.innerHTML = greeter(user);
```

Basic Types in TypeScript [\(source\)](#)

Programming is all about data, and manipulating the simplest, most primitive forms of data is what makes the program function. TypeScript has all the same datatypes as JavaScript but with an addition of a useful enumeration type.

Boolean

The boolean type is the most datatype and it's a simple false or true value.

```
let boolean_var : boolean = true;
```

Number

All numbers in TypeScript are floating point numbers and their type in TypeScript is `number`. TypeScript supports decimal, hexadecimal, binary and octal literals.

```
let decimal: number = 10;
let hex: number = 0x00F0;
let binary: number = 0b1010;
let octal: number = 0o744;
```

String

TypeScript, like in many other languages, handles textual data using the type `string`. TypeScript can use either double quotes (") or single quotes (') to surround string data.

```
let name: string = "John";
let color: string = 'blue';
```

You could also use backtick/backquote to surround what are called **template strings** which can span multiple lines and have embedded expressions, and these expressions have the form `${ expression }`.

```
let age: number = 34;
let name: string = "John Doe";
let line: string = `Hello, I am ${ name } and I will be ${ age + 1 } years old next week`;
```

Array

TypeScript, like most languages, allows you to use arrays of multiple values. Arrays can be written in two ways, you can either write the type of the array followed by square brackets `[]` or you can use a generic array type, i.e. `Array<type>`.

```
let list1 : number[] = [1,2,3];
let list2: Array<number> = [1,2,3];
```

Tuple

Tuples are arrays with a fixed number of elements, however you can use different types in the same tuple.

```
let tuple1: [string, number];
tuple1 = ["typescript", 6]; //correct
tuple1 = [6, "typescript"]; //incorrect
```

When accessing an element with an index that is bigger than the index given when initializing the tuple, you create a union type.

```
tuple1[3] = "javascript"; //correct, since the type is either string or number.
tuple1[4] = false; //incorrect, since the type is not number or string.
```

Union types are an advanced types and will be covered later on.

Enum

Enum is a type in TypeScript that is not found in JavaScript, and is an array of numeric values which have specific names.

```
enum Color {Red, Green, Blue}
let c: Color = Color.Green;
```

By default, enums will number their elements starting with 0. You can change this by manually changing the value of one of the members. We can change the starting value to 1 by doing this:

```
enum Color {Red = 1, Green, Blue}
let c: Color = Color.Green;
```

You can even set all the values to a specific number:

```
enum Color {Red = 1, Green = 2, Blue = 4}
let c: Color = Color.Green;
```

Any

The any type is a self-explanatory type, as in it can hold any other type in itself. What this means is that we do not check the type of the data, rather we let the compile-time check handle it.

```
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay
```

With any, you can have an array which contains many variables of different data types, like so:

```
let list: any[] = [1, true, "free"];
```

Contrary to the expected, Object does not play a similar role to any, rather it only allows you to assign any value to them - you can't call arbitrary methods on them, even ones that actually exist:

```
let notSure: any = 4;
notSure.exists(); // okay, exists might exist at runtime
notSure.toFixed(); // okay, toFixed exists (but the compiler doesn't check)

let prettySure: Object = 4;
prettySure.toFixed(); // Error: Property 'toFixed' doesn't exist on type 'Object'.
```

Void

Void is the opposite of any, as in it denotes an absence of any type, this is oftenly used to signal that a function doesn't return a value:

```
function warnUser(): void {
    alert("This is my warning message");
}
```

You can declare variables of type void, but it is not very practical since you can only assign undefined or null to them:

```
let unusable: void = undefined;
```

Null and Undefined

Null and undefined actually are both separate types in TypeScript, however, much like void, they're not very useful; on their own, as you cannot assign anything else to a variable once it has been assigned an undefined or null type:

```
let u: undefined = undefined;
```



```
let n: null = null;
```

You can add undefined or null to any other type e.g. number, which means that null and undefined are subtypes of all types. There is an exception however, and that is when you use the `--strictNullChecks` flag where null and undefined are only assignable to void types. This is very helpful in debugging, as you can avoid many common errors. But, if you want to pass either a string, null, or undefined, you can use a `string | null | undefined` union type. More on union types later on.

Never

Never is a type that denotes values that never occur. For example a never return type can be assigned to a function that always throws an exception or a function that never returns. Variables also acquire the type never when narrowed by any type guards that can never be true. Never is assignable to every type, however no type is assignable to never:

```
// Function returning never must have unreachable end point
function error(message: string): never {
    throw new Error(message);
}

// Inferred return type is never
function fail() {
    return error("Something failed");
}

// Function returning never must have unreachable end point
function infiniteLoop(): never {
    while (true) {
    }
}
```

Type Assertions

A type assertion allows you to bypass TypeScript's checking of data types. This is used when you want to do a specific task, and you know that you won't run into any problems, but would otherwise not be allowed to do it by the compiler. Type assertions act similarly to type casts in other languages, however, as stated before, the compiler assumes that you have done all the necessary checks and does not do any checks or data restructuring of its own. You can type assert in two ways, firstly, using angle brackets:

```
let someValue: any = "this is a string";

let strLength: number = (<string>someValue).length;
```

Secondly, using the keyword `as`:

```
let someValue: any = "this is a string";

let strLength: number = (someValue as string).length;
```

The above examples both accomplish the exact same thing, and using one over the other is merely an act of preference, however when using TypeScript with JSX, only as type assertions are allowed.

Keywords and Features in TypeScript

let

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:

```
var foo = 123;
if (true) {
    var foo = 456;
}
console.log(foo); // 456
```

This is because { does not create a new variable scope. The variable foo is the same inside the if block as it is outside the if block. This is a common source of errors in JavaScript programming. This is why TypeScript (and ES6) introduces the let keyword to allow you to define variables with true block scope. That is if you use let instead of var you get a true unique element disconnected from what you might have defined outside the scope. The same example is demonstrated with let:

```
let foo = 123;
if (true) {
    let foo = 456;
}
console.log(foo); // 123
```

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

```
var index = 0;
var array = [1, 2, 3];
for (let index = 0; index < array.length; index++) {
    console.log(array[index]);
}
```

```
}  
console.log(index); // 0
```

In all sincerity, we find it better to use `let` whenever possible as it leads to lesser surprises for new and existing multi-lingual developers.

Functions create a new scope

Since we mentioned it, we'd like to demonstrate that functions create a new variable scope in JavaScript. Consider the following:

```
var foo = 123;  
function test() {  
    var foo = 456;  
}  
test();  
console.log(foo); // 123
```

This behaves as you would expect. Without this it would be very difficult to write code in JavaScript.

The Generated JS

The JS generated by TypeScript is simple renaming of the `let` variable if a similar name already exists in the surrounding scope. E.g. the following is generated as is with a simple replacement of `var` with `let`:

```
if (true) {  
    let foo = 123;  
}  
  
// becomes //  
  
if (true) {  
    var foo = 123;  
}
```

However, if the variable name is already taken by the surrounding scope then a new variable name is generated as shown (notice `_foo`):

```
var foo = '123';  
if (true) {  
    let foo = 123;  
}
```

```
// becomes //

var foo = '123';
if (true) {
    var _foo = 123; // Renamed
}
```

let in Closures

A common programming interview question for a JavaScript developer is what is the log of this simple file:

```
var funcs = [];
// create a bunch of functions
for (var i = 0; i < 3; i++) {
    funcs.push(function() {
        console.log(i);
    })
}
// call them
for (var j = 0; j < 3; j++) {
    funcs[j]();
}
```

One would have expected it to be 0,1,2. Surprisingly it is going to be 3 for all three functions. Reason is that all three functions are using the variable `i` from the outer scope and at the time we execute them (in the second loop) the value of `i` will be 3 (that's the termination condition for the first loop).

A fix would be to create a new variable in each loop specific to that loop iteration. As we've learn before we can create a new variable scope by creating a new function and immediately executing it (i.e. the IIFE pattern from classes `(function() { /* body */ })();`) as shown below:

```
var funcs = [];
// create a bunch of functions
for (var i = 0; i < 3; i++) {
    (function() {
        var local = i;
        funcs.push(function() {
            console.log(local);
        })
    })();
}
// call them
for (var j = 0; j < 3; j++) {
    funcs[j]();
}
```

```
}
```

Here the functions close over (hence called a closure) the local variable (conveniently named `local`) and use that instead of the loop variable `i`.

Note that closures come with a performance impact (they need to store the surrounding state).

The ES6 `let` keyword in a loop would have the same behavior as the previous example:

```
var funcs = [];  
// create a bunch of functions  
for (let i = 0; i < 3; i++) { // Note the use of let  
  funcs.push(function() {  
    console.log(i);  
  })  
}  
// call them  
for (var j = 0; j < 3; j++) {  
  funcs[j]();  
}
```

Using a `let` instead of `var` creates a variable `i` unique to each loop iteration.

To summarize, `let` is extremely useful to have for the vast majority of code. It can greatly enhance your code readability and decrease the chance of a programming error

const

`const` is a very welcomed 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;
```

The syntax is much better (IMHO) than other languages that force the user to type something like `let constant foo` i.e. a variable + behavior specifier.

`const` is a good practice for both readability and maintainability and avoids using *magic literals* e.g.

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

```
const maxRows = 10;
if (x > maxRows) {
}
```

const declarations must be initialized

The following is a compiler error:

```
const foo; // ERROR: const declarations must be initialized
```

Left hand side of assignment cannot be a constant

Constants are immutable after creation, so if you try to assign them to a new value it is a compiler error:

```
const foo = 123;
foo = 456; // ERROR: Left-hand side of an assignment expression cannot be a constant
```

Block Scoped

A const is block scoped like we saw with [let](#):

```
const foo = 123;
if (true) {
  const foo = 456; // Allowed as its a new variable limited to this `if` block
}
```

Deep immutability

A const works with object literals as well, as far as protecting the variable reference is concerned:

```
const foo = { bar: 123 };
foo = { bar: 456 }; // ERROR : Left hand side of an assignment expression cannot be a constant
```

However it still allows sub properties of objects to be mutated, as shown below:

```
const foo = { bar: 123 };
foo.bar = 456; // Allowed!
console.log(foo); // { bar: 456 }
```

For this reason I recommend using const with literals or immutable data structures.

for...of

A common error experienced by beginning 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 below example. Here you would expect 9,2,5 but you get the indexes 0,1,2:

```
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:

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

Similarly, TypeScript has no trouble going through a string character by character using `for...of`:

```
var hello = "is it me you're looking for?";
for (var char of hello) {
    console.log(char); // is it me you're looking for?
}
```

JS Generation

For pre ES6 targets TypeScript will generate the standard `for (var i = 0; i < list.length; i++)` kind of loop. For example here's what gets generated for our previous example:

```
var someArray = [9, 2, 5];
for (var item of someArray) {
    console.log(item);
}

// becomes //

for (var _i = 0; _i < someArray.length; _i++) {
    var item = someArray[_i];
    console.log(item);
}
```

You can see that using `for...of` makes intent clearer and also decreases the amount of code you have to write (and variable names you need to come up with).

Limitations

If you are not targeting ES6 or above, the generated code assumes the property length exists on the object and that the object can be indexed via numbers e.g. `obj[2]`. So it is only supported on string and array for these legacy JS engines.

If TypeScript can see that you are not using an array or a string it will give you a clear error "is not an array type or a string type";

```
let articleParagraphs = document.querySelectorAll("article > p");
// Error: Nodelist is not an array type or a string type
for (let paragraph of articleParagraphs) {
  paragraph.classList.add("read");
}
```

Use `for...of` only for stuff that you know to be an array or a string. Note that this limitation might be removed in a future version of TypeScript.

To summarize, you would be surprised at how many times you will be iterating over the elements of an array. The next time you find yourself doing that, give `for...of` a go. You might just make the next person who reviews your code happy.

Multiline Strings

Ever wanted to put a newline in a JavaScript string? Perhaps you wanted to embed some lyrics? You would have needed to escape the literal newline using our favorite escape character `\`, and then put a new line into the string manually `\n` at the next line. This is shown below:

```
var lyrics = "Never gonna give you up \
\nNever gonna let you down";
```

With TypeScript you can just use a template string:

```
var lyrics = `Never gonna give you up
Never gonna let you down`;
```

Arrow Functions

Lovingly called 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 motivation for a fat arrow is:

1. You don't need to keep typing `function`
2. It lexically captures the meaning of `this`
3. 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 has traditionally been a pain point in JavaScript. As a wise man once said "I hate JavaScript as it tends to lose the meaning of this all too easily". Fat arrows fix it by capturing the meaning of this from the surrounding context. Consider this pure JavaScript class:

```
function Person(age) {
  this.age = age;
  this.growOld = function() {
    this.age++;
  }
}
var person = new Person(1);
setTimeout(person.growOld, 1000);

setTimeout(function() { console.log(person.age); }, 2000); // 1, should have been 2
```

If you run this code in the browser this `this` within the function is going to point to `window` because `window` is going to be what executes the `growOld` function. Fix is to use an arrow function:

```
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
```

The reason why this works is the reference to `this` is captured by the arrow function from outside the function body. This is equivalent to the following JavaScript code (which is what you would write yourself if you didn't have TypeScript):

```
function Person(age) {
```

```

    this.age = age;
    var _this = this; // capture this
    this.growOld = function() {
        _this.age++; // use the captured this
    }
}
var person = new Person(1);
setTimeout(person.growOld,1000);

setTimeout(function() { console.log(person.age); },2000); // 2

```

Note that since you are using TypeScript you can be even sweeter in syntax and combine arrows with classes:

```

class Person {
    constructor(public age:number) {}
    growOld = () => {
        this.age++;
    }
}
var person = new Person(1);
setTimeout(person.growOld,1000);

setTimeout(function() { console.log(person.age); },2000); // 2

```

[A video about this pattern](#)

A few helpful Tips

Tip: Arrow Function Need

Beyond the terse syntax, you only need to use the fat arrow if you are going to give the function to someone else to call. Effectively:

```

var growOld = person.growOld;
// Then later someone else calls it:
growOld();

```

If you are going to call it yourself, i.e.

```

person.growOld();

```

then this is going to be the correct calling context (in this example person).

Tip: Arrow Function Danger

In fact if you want this to be the calling context you should not use the arrow function. This is the case with callbacks used by libraries like jquery, underscore, mocha and others. If the documentation mentions functions on this then you should probably just use a function instead of a fat arrow. Similarly if you plan to use arguments don't use an arrow function.

Tip: Arrow functions with libraries that use this

Many libraries do this e.g. jQuery iterables (one example <http://api.jquery.com/jquery.each/>) will use this to pass you the object that it is currently iterating over. In this case if you want to access the library passed this as well as the surrounding context just use a temp variable like `_self` like you would in the absence of arrow functions.

```
let _self = this;
something.each(function() {
  console.log(_self); // the lexically scoped value
  console.log(this);  // the library passed value
});
```

Tip: Arrow functions and inheritance

If you have an instance method as an arrow function then it goes on this. Since there is only one `this` such functions cannot participate in a call to `super` (super only works on prototype members). You can easily get around it by creating a copy of the method before overriding it in the child.

```
class Adder {
  constructor(public a: number) {}
  // This function is now safe to pass around
  add = (b: number): number => {
    return this.a + b;
  }
}

class ExtendedAdder extends Adder {
  // Create a copy of parent before creating our own
  private superAdd = this.add;
  // Now create our override
  add = (b: number): number => {
    return this.superAdd(b);
  }
}
```

Code in JavaScript vs code in TypeScript

To examine how TypeScript actually transpiles to JavaScript we have offered a few examples in order to see some of the ways that the TS transpiler decided to write to JavaScript version of our TS input code.

Here, we simply define a class named Planet which holds two variables namely, mass and moons and also within our Planet class we add a reportMoons() functions to log the information contained in our variables. We can see that the JavaScript transformation of the TS code is still relatively similar to the original but with added wrappers and functions to achieve that OOP feel.

TypeScript:

```
class Planet {
  let mass: string;
  let moons: number;

  constructor (mass: string, moons: number) {
    this.mass = mass;
    this.moons = moons || 0;
  }

  function reportMoons () {
    console.log(`I have ${this.moons} moons.`)
  }
}

// Yeah, Jupiter really does have (at least) 67 moons.
const jupiter = new Planet('Pretty Big', 67);
jupiter.reportMoons();
```

JavaScript:

```
var Planet = (function () {
  function Planet() {
  }
  return Planet;
})();
var mass;
var moons;
constructor(mass, string, moons, number);
{
  this.mass = mass;
  this.moons = moons || 0;
}
function reportMoons() {
  console.log("I have " + this.moons + " moons.");
}
// Yeah, Jupiter really does have (at least) 67 moons.
var jupiter = new Planet('Pretty Big', 67);
```

```
jupiter.reportMoons();
```

In this next example we can see how TypeScript changes when we write code implementing class inheritance, as you can see the JavaScript version is rather longer and more complex, but when examined a little more intensively we can see that TypeScript utilises the JavaScripts prototypes to actually define certain features of the object from a certain class

TypeScript:

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

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 the Python");
let tom: Animal = new Horse("Tommy the Palomino");

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

JavaScript:

```
var __extends = (this && this.__extends) || (function () {
    var extendStatics = function (d, b) {
        d.__proto__ = b;
    };
    return function (d, b) {
        extendStatics(d, b);
        function __() {
            this.constructor = b;
        }
        d.prototype = b === null ? Object.create(b) : (__.prototype = b.prototype, new __());
    };
})();

var Animal = (function () {
    function Animal(name) {
        this.name = name;
    }
    Animal.prototype.move = function (distanceInMeters = 0) {
        console.log(`${this.name} moved ${distanceInMeters}m.`);
    };
    return Animal;
})();
```

```

        this.name = name;
    }
    Animal.prototype.move = function (distanceInMeters) {
        if (distanceInMeters === void 0) { distanceInMeters = 0; }
        console.log(this.name + " moved " + distanceInMeters + "m.");
    };
    return Animal;
})();
var Snake = (function (_super) {
    __extends(Snake, _super);
    function Snake(name) {
        return _super.call(this, name) || this;
    }
    Snake.prototype.move = function (distanceInMeters) {
        if (distanceInMeters === void 0) { distanceInMeters = 5; }
        console.log("Slithering...");
        _super.prototype.move.call(this, distanceInMeters);
    };
    return Snake;
})(Animal);
var Horse = (function (_super) {
    __extends(Horse, _super);
    function Horse(name) {
        return _super.call(this, name) || this;
    }
    Horse.prototype.move = function (distanceInMeters) {
        if (distanceInMeters === void 0) { distanceInMeters = 45; }
        console.log("Galloping...");
        _super.prototype.move.call(this, distanceInMeters);
    };
    return Horse;
})(Animal);
var sam = new Snake("Sammy the Python");
var tom = new Horse("Tommy the Palomino");
sam.move();
tom.move(34);

```

Pitfalls of Javascript ([source](#))

In this segment, we will analyze the quirky and strange, unexpected parts of JavaScript

Equality Operator

JavaScript has 2 types of equality operators, “==” and “===”. They are both essentially the same, however the operator “===” does no type conversion, and the types must be the same to be considered equal.

```
alert(1 == "1");//true
```

```
alert(1 === "1");//false
```

Object Usage and Properties

Everything in JavaScript acts like an object, with the only two exceptions being null and undefined.

```
false.toString(); // 'false'  
[1, 2, 3].toString(); // '1,2,3'  
  
function Foo(){}  
Foo.bar = 1;  
Foo.bar; // 1
```

A common misconception is that number literals cannot be used as objects. That is because a flaw in JavaScript's parser tries to parse the dot notation on a number as a floating point literal.

```
2.toString(); // raises SyntaxError
```

There are a couple of workarounds that can be used to make number literals act as objects too.

```
2..toString(); // the second point is correctly recognized  
2 .toString(); // note the space left to the dot  
(2).toString(); // 2 is evaluated first
```

Objects as a Data Type

Objects in JavaScript can also be used as Hashmaps; they mainly consist of named properties mapping to values.

Using an object literal - {} notation - it is possible to create a plain object. This new object inherits from Object.prototype and does not have own properties defined.

```
var foo = {}; // a new empty object  
  
// a new object with a 'test' property with value 12  
var bar = {test: 12};
```

Accessing Properties

The properties of an object can be accessed in two ways, via either the dot notation or the square bracket notation.

```
var foo = {name: 'kitten'}
```

```

foo.name; // kitten
foo['name']; // kitten

var get = 'name';
foo[get]; // kitten

foo.1234; // SyntaxError
foo['1234']; // works

```

The notations work almost identically, with the only difference being that the square bracket notation allows for dynamic setting of properties and the use of property names that would otherwise lead to a syntax error.

Deleting Properties

The only way to remove a property from an object is to use the delete operator; setting the property to undefined or null only removes the value associated with the property, but not the key.

```

var obj = {
  bar: 1,
  foo: 2,
  baz: 3
};
obj.bar = undefined;
obj.foo = null;
delete obj.baz;

for(var i in obj) {
  if (obj.hasOwnProperty(i)) {
    console.log(i, ' ' + obj[i]);
  }
}

```

The above outputs both bar undefined and foo null - only baz was removed and is therefore missing from the output.

Notation of Keys

```

var test = {
  'case': 'I am a keyword, so I must be notated as a string',
  delete: 'I am a keyword, so me too' // raises SyntaxError
};

```

Object properties can be both notated as plain characters and as strings. Due to another mis-design in JavaScript's parser, the above will throw a SyntaxError prior to ECMAScript 5.

This error arises from the fact that `delete` is a keyword; therefore, it must be notated as a string literal to ensure that it will be correctly interpreted by older JavaScript engines.

The Prototype

JavaScript does not feature a classical inheritance model; instead, it uses a prototypal one.

While this is often considered to be one of JavaScript's weaknesses, the prototypal inheritance model is in fact more powerful than the classic model. It is, for example, fairly trivial to build a classic model on top of a prototypal model, while the other way around is a far more difficult task.

JavaScript is the only widely used language that features prototypal inheritance, so it can take time to adjust to the differences between the two models.

The first major difference is that inheritance in JavaScript uses prototype chains.

Note: Simply using `Bar.prototype = Foo.prototype` will result in both objects sharing the same prototype. Therefore, changes to either object's prototype will affect the prototype of the other as well, which in most cases is not the desired effect.

```
function Foo() {
    this.value = 42;
}
Foo.prototype = {
    method: function() {}
};

function Bar() {}

// Set Bar's prototype to a new instance of Foo
Bar.prototype = new Foo();
Bar.prototype.foo = 'Hello World';

// Make sure to list Bar as the actual constructor
Bar.prototype.constructor = Bar;

var test = new Bar(); // create a new bar instance

// The resulting prototype chain
test [instance of Bar]
  Bar.prototype [instance of Foo]
    { foo: 'Hello World' }
  Foo.prototype
    { method: ... }
  Object.prototype
    { toString: ... /* etc. */ }
```

In the code above, the object test will inherit from both Bar.prototype and Foo.prototype; hence, it will have access to the function method that was defined on Foo. It will also have access to the property value of the one Foo instance that is its prototype. It is important to note that new Bar() does not create a new Foo instance, but reuses the one assigned to its prototype; thus, all Bar instances will share the same value property.

Property Lookup

While the prototype property is used by the language to build the prototype chains, it is still possible to assign any given value to it. However, primitives will simply get ignored when assigned as a prototype.

```
function Foo() {}  
Foo.prototype = 1; // no effect
```

Assigning objects, as shown in the example above, will work, and allows for dynamic creation of prototype chains.

Performance

The lookup time for properties that are high up on the prototype chain can have a negative impact on performance, and this may be significant in code where performance is critical. Additionally, trying to access non-existent properties will always traverse the full prototype chain.

Also, when iterating over the properties of an object every property that is on the prototype chain will be enumerated.

Extension of Native Prototypes

One mis-feature that is often used is to extend Object.prototype or one of the other built in prototypes.

This technique is called monkey patching and breaks encapsulation. While used by popular frameworks such as Prototype, there is still no good reason for cluttering built-in types with additional non-standard functionality.

The only good reason for extending a built-in prototype is to backport the features of newer JavaScript engines; for example, Array.forEach.

hasOwnProperty

To check whether an object has a property defined on itself and not somewhere on its prototype chain, it is necessary to use the `hasOwnProperty` method which all objects inherit from `Object.prototype`.

Note: It is not enough to check whether a property is undefined. The property might very well exist, but its value just happens to be set to undefined.

`hasOwnProperty` is the only thing in JavaScript which deals with properties and does not traverse the prototype chain.

```
// Poisoning Object.prototype
Object.prototype.bar = 1;
var foo = {goo: undefined};

foo.bar; // 1
'bar' in foo; // true

foo.hasOwnProperty('bar'); // false
foo.hasOwnProperty('goo'); // true
```

Only `hasOwnProperty` will give the correct and expected result. See the section on for in loops for more details on when to use `hasOwnProperty` when iterating over object properties.

Array type

JavaScript is usually called a functional programming language, however it is actually a multi-paradigm language with functional features and a little bit of a very roundabout object oriented system. It lacks specificity, i.e. it has a very “loose” typing style. For example it doesn’t have an integer type, nor does it have a proper array type, instead it uses the “number” type, and as for the arrays, to put it bluntly, they are very strange, see this example in JavaScript:

```
var arr = [];
alert(arr.length); // this outputs "0"
alert(arr[3]); // this outputs "undefined"
arr[3] = "hi";
alert(arr.length); // this outputs "4"
alert(arr.3); // this line throws a SyntaxError
alert(arr["3"]); // this outputs "hi"
delete arr[3];
alert(arr.length); // this outputs "4"
alert(arr[3]); // this outputs "undefined"
```

As we can see from the code, the first two lines make sense, however, as we go on, the strangeness and loose typing of JavaScript really shows. The third line should, in theory, give an `ArrayBoundsException`, however instead it gives “undefined”. Since the array `arr` is an empty array, line 4 should not work at all, however, unexpectedly, it doesn’t give any sort of error. Then if you print out the length of the array again, it says that it’s 4, even though we’ve only added 1

element. So, this would make us think that arrays in JavaScript are object with numerical properties, however, if that were true line 6 should work, however it gives a syntax error. But, If that line doesn't work, then also line 7 shouldn't work, since line 6 should prove that arrays aren't object, however line 7 works, so if you pass a number, with quotes surrounding it to an array, it works. Then, after we delete index 3 in line 8 and we want to print out the length of the array, it still outputs 4, even though we deleted the third index, and also when we try to access the third index again after deleting it, it still gives undefined just like before, however in the previous lines, when we accessed the length, it gave us 0, as it should be.

Optional Semicolons

From the previous paragraph we can only conclude that array types in JavaScript aren't arrays, but are some sort of amalgamated array-list-dictionary multi-typed objects. Now let's take a look at JavaScript's object oriented code:

```
function returnObj(){
    return {
        x:42
    };
}
alert(JSON.stringify(returnObj())); // this outputs {"x": 42}
function returnObj2(){
    return
    {
        x:42
    };
}
alert(JSON.stringify(returnObj2())); // this outputs "undefined"
```

There are 2 functions here that only return an object, however the only difference between `returnObj()` and `returnObj2()` is that the latter has a newline after the `return` keyword. This shouldn't change anything, however we can see from the last line that it prints `undefined`. This is because JavaScript has so-called "optional semicolons". Basically, in JavaScript you can usually omit the semicolon between two statements if those statements are written on separate lines, but JavaScript does not treat every line break as a semicolon, rather it usually treats linebreaks as semicolons only if it can't parse the code without the semicolons, however if a line break appears after `return`, `break` or `continue`, JavaScript will always interpret that line break as a semicolon. This is a very strange functionality from JavaScript working on seemingly random principles.

Paradigms in Programming

Functional Programming (JavaScript)

There are many types of paradigms in programming, including imperative, declarative, structured, procedural, object-oriented, functional etc. A paradigm simply means a way or style of programming, some languages make it easier to write in a certain paradigms rather than others.

Specifically, we are going to focus on the functional programming paradigm which is a style of coding that treats all computation as an evaluation of mathematical functions and avoids changing-state and mutable data, which means that it doesn't use data which can be altered after its creation and also the state of the program cannot change. Functional programming is a type of declarative programming, which uses expressions or declarations to program instead of statements.[\(source\)](#)

It's helpful and important to develop a functional style of programming that aims to simplify your own libraries and applications, and helps tame the wild beast of JavaScript complexity.

As a bare-bones introduction, functional programming can be described in a single sentence: Functional programming is the use of functions that transform values into units of abstraction, subsequently used to build software systems

The major evolution that is still going on for me is towards a more functional programming style, which involves unlearning a lot of old habits, and backing away from some OOP directions. —John Carmack

If you're familiar with object-oriented programming, then you may agree that its primary goal is to break a problem into parts, smaller parts which in turn will be combined to represent a bigger part.

By comparison, a strict functional programming approach to solving problems also breaks a problem into parts (namely, functions).

Whereas the object-oriented approach tends to break problems into groupings of "nouns," or objects, a functional approach breaks the same problem into groupings of "verbs," or functions.

As with object-oriented programming, larger functions are formed by "gluing" or "composing" other functions together to build high-level behaviors.

Regarding encapsulation, sometimes it [encapsulation] is used to restrict the visibility of certain elements, and this act is known as data hiding. By using functional techniques involving closures, you can achieve data hiding that is as effective as the same capability offered by most object-oriented languages.

What is a closure?

A closure is an inner function that has access to the outer (enclosing) function's variables—scope chain. The closure has three scope chains: it has access to its own scope (variables

defined between its curly brackets), it has access to the outer function's variables, and it has access to the global variables.

The inner function has access not only to the outer function's variables, but also to the outer function's parameters. Note that the inner function cannot call the outer function's arguments object, however, even though it can call the outer function's parameters directly.

You create a closure by adding a function inside another function.

```
function showName (firstName, lastName) {  
    var nameIntro = "Your name is ";  
    // this inner function has access to the outer function's variables, including  
    the parameter  
    function makeFullName () {  
        return nameIntro + firstName + " " + lastName;  
    }  
    return makeFullName ();  
}  
  
showName ("Michael", "Jackson"); // Your name is Michael Jackson
```

Closures are used extensively in Node.js; they are workhorses in Node.js' asynchronous, non-blocking architecture. Closures are also frequently used in jQuery and just about every piece of JavaScript code you read.

A little bit about Closures ([source](#))

Closures have access to the outer function's variable even after the outer function returns:

One of the most important and ticklish features with closures is that the inner function still has access to the outer function's variables even after the outer function has returned. Yep, you read that correctly. When functions in JavaScript execute, they use the same scope chain that was in effect when they were created. This means that even after the outer function has returned, the inner function still has access to the outer function's variables. Therefore, you can call the inner function later in your program. This example demonstrates:

```
function celebrityName (firstName) {  
    var nameIntro = "This celebrity is ";  
    // this inner function has access to the outer function's variables, including the  
    parameter  
    function lastName (theLastName) {  
        return nameIntro + firstName + " " + theLastName;  
    }  
}
```

```

    return lastName;
}

var mjName = celebrityName ("Michael"); // At this juncture, the celebrityName outer
function has returned.

// The closure (lastName) is called here after the outer function has returned above
// Yet, the closure still has access to the outer function's variables and parameter
mjName ("Jackson"); // This celebrity is Michael Jackson

```

Closures store references to the outer function's variables

They do not store the actual value. Closures get more interesting when the value of the outer function's variable changes before the closure is called. And this powerful feature can be harnessed in creative ways, such as this private variables example first demonstrated by Douglas Crockford:

```

function celebrityID () {
    var celebrityID = 999;
    // We are returning an object with some inner functions
    // All the inner functions have access to the outer function's variables
    return {
        getID: function () {
            // This inner function will return the UPDATED celebrityID variable
            // It will return the current value of celebrityID, even after the
            changeTheID function changes it
            return celebrityID;
        },
        setID: function (theNewID) {
            // This inner function will change the outer function's variable anytime
            celebrityID = theNewID;
        }
    }
}

var mjID = celebrityID (); // At this juncture, the celebrityID outer function has
returned.
mjID.getID(); // 999
mjID.setID(567); // Changes the outer function's variable
mjID.getID(); // 567: It returns the updated celebrityId variable

```

Closures Gone Awry

Because closures have access to the updated values of the outer function's variables, they can also lead to bugs when the outer function's variable changes with a for loop.

```
// This example is explained in detail below (just after this code box).
function celebrityIDCreator (theCelebrities) {
    var i;
    var uniqueID = 100;
    for (i = 0; i < theCelebrities.length; i++) {
        theCelebrities[i]["id"] = function () {
            return uniqueID + i;
        }
    }

    return theCelebrities;
}

var actionCelebs = [{name:"Stallone", id:0}, {name:"Cruise", id:0}, {name:"Willis", id:0}];

var createIdForActionCelebs = celebrityIDCreator (actionCelebs);

var stalloneID = createIdForActionCelebs [0];console.log(stalloneID.id()); // 103
```

In the preceding example, by the time the anonymous functions are called, the value of *i* is 3 (the length of the array and then it increments). The number 3 was added to the uniqueID to create 103 for ALL the celebritiesID. So every position in the returned array get id = 103, instead of the intended 100, 101, 102.

The reason this happened was because, as we have discussed in the previous example, the closure (the anonymous function in this example) has access to the outer function's variables by reference, not by value. So just as the previous example showed that we can access the updated variable with the closure, this example similarly accessed the *i* variable when it was changed, since the outer function runs the entire for loop and returns the last value of *i*, which is 103.

To fix this side effect (bug) in closures, you can use an Immediately Invoked Function Expression (IIFE), such as the following:

```
function celebrityIDCreator (theCelebrities) {
    var i;
    var uniqueID = 100;
    for (i = 0; i < theCelebrities.length; i++) {
        theCelebrities[i]["id"] = function (j) { // the j parametric variable is the i
            // passed in on invocation of this IIFE
            return function () {
                return uniqueID + j; // each iteration of the for loop passes the
                // current value of i into this IIFE and it saves the correct value to the array
            } () // BY adding () at the end of this function, we are executing it
            // immediately and returning just the value of uniqueID + j, instead of returning a
            // function.
        } (i); // immediately invoke the function passing the i variable as a parameter
    }
}
```



```

    return theCelebrities;
}

var actionCelebs = [{name:"Stallone", id:0}, {name:"Cruise", id:0}, {name:"Willis", id:0}];

var createIdForActionCelebs = celebrityIDCreator (actionCelebs);

var stalloneID = createIdForActionCelebs [0];
console.log(stalloneID.id); // 100

var cruiseID = createIdForActionCelebs [1];console.log(cruiseID.id); // 101

```

Reason why Closures are awesome

It allows you to compose objects easily e.g. the revealing module pattern:

```

function createCounter() {
  let val = 0;
  return {
    increment() { val++ },
    getVal() { return val }
  }
}

let counter = createCounter();
counter.increment();
console.log(counter.getVal()); // 1
counter.increment();
console.log(counter.getVal()); // 2

```

At a high level it is also what makes something like nodejs possible (don't worry if it doesn't click in your brain right now. It will eventually):

```

// Pseudo code to explain the concept
server.on(function handler(req, res) {
  loadData(req.id).then(function(data) {
    // the `res` has been closed over and is available
    res.send(data);
  })
});

```

Functions as Units of Behavior

Hiding data and behavior (which has the side effect of providing a more agile change experience) is just one way that functions can be units of abstraction. Another is to

provide an easy way to store and pass around discrete units of basic behavior. Take, for example, JavaScript's syntax to denote looking up a value in an array by index:

```
var letters = ['a', 'b', 'c'];
letters[1];
//=> 'b'
```

While array indexing is a core behavior of JavaScript, there is no way to grab hold of the behavior and use it as needed without placing it into a function. Therefore, a simple example of a function that abstracts array indexing behavior could be called `nth`. The naive implementation of `nth` is as follows:

```
function naiveNth(a, index) {
  return a[index];
}
As you might suspect, nth operates along the happy path perfectly fine:
naiveNth(letters, 1);
//=> "b"
However, the function will fail if given something unexpected:
naiveNth({}, 1);
//=> undefined
```

Therefore, if I were to think about the abstraction surrounding a function `nth`, I might devise the following statement: `nth` returns the element located at a valid index within a data type allowing indexed access. A key part of this statement is the idea of an indexed data type. To determine if something is an indexed data type, I can create a function `isIndexed`, implemented as follows:

```
function isIndexed(data) {
  return _.isArray(data) || _.isString(data);
}
```

The function `isIndexed` is also a function providing an abstraction over checking if a piece of data is a string or an array. Building abstraction on abstraction leads to the following complete implementation of `nth`:

```
function nth(a, index) {
  if (!_isNumber(index)) fail("Expected a number as the index");
  if (!isIndexed(a)) fail("Not supported on non-indexed type");
  if ((index < 0) || (index > a.length - 1))
    fail("Index value is out of bounds");
  return a[index];
}
```

The completed implementation of `nth` operates as follows:

```
nth(letters, 1);
//=> 'b'
nth("abc", 0);
//=> "a"
nth({}, 2);
// Error: Not supported on non-indexed type
nth(letters, 4000);
// Error: Index value is out of bounds
```

The point is to make independent functions with very specific tasks which they perform very well, in the case that they do not perform well or cannot perform at all they should give out an appropriate response, such as an exception, like this, the function as well as the program is more robust and can perhaps continue to run even though it has encountered a few mishaps. For the sake of example:

```
function lessOrEqual(x, y) {  
    return x <= y;  
}
```

To add on, functions that always return a Boolean value (i.e., true or false only), are called predicates. So, instead of an elaborate comparator construction, lessOrEqual is simply a “skin” over the built-in <= operator:

```
[2, 3, -1, -6, 0, -108, 42, 10].sort(lessOrEqual);  
//=> [100, 10, 1, 0, -1, -1, -2]
```

This functional way of programming is in a basic sense a way of thinking. Thinking about a complex problem, dividing it into small actions and combining those into one big process which shall tackle the big problem. For example, to make things more readable and suitable for other programmers which might use our code we can do something like this:

We define two useful functions that we might often find a need for: existy and truthy.

The function existy is meant to define the existence of something. JavaScript has two values—null and undefined—that signify nonexistence. Thus, existy checks that its argument is neither of these things, and is implemented as follows:

```
function existy(x) { return x != null };
```

Using the loose inequality operator (!=), it is possible to distinguish between null, undefined, and everything else. It's used as follows:

```
existy(null);  
//=> false  
existy(undefined);  
//=> false  
existy({}.notHere);  
//=> false  
existy((function(){})( ));  
//=> false  
existy(0);  
//=> true  
existy(false);  
//=> true
```

The use of existy simplifies what it means for something to exist in JavaScript. Minimally, it collocates the existence check in an easy-to-use function. The second function mentioned, truthy, is defined as follows:

```
function truthy(x) { return (x !== false) && existy(x) };
```

The truthy function is used to determine if something should be considered a synonym for true, and is used as shown here

```

truthy(false);
//=> false
truthy(undefined);
//=> false
truthy(0);
//=> true
truthy('');
//=> true

```

In JavaScript, it's sometimes useful to perform some action only if a condition is true and return something like undefined or null otherwise. The general pattern is as follows:

```

{
  if(condition)
    return _.isFunction(doSomething) ? doSomething() : doSomething;
  else
    return undefined;
}

```

Using truthy, we can encapsulate this logic in the following way:

```

function doWhen(cond, action) {
  if(truthy(cond))
    return action();
  else
    return undefined;
}

```

Thus we have created simpler-to-understand functions which when implemented result in shorter and more readable code. The goal is to create function on which we as programmers can rely on, functions which we can comfortably expect to do their tasks well and appropriately as their name would suggest. This helps the programmer to get a better feel of what he has, to understand what he can do regarding a certain problem. It is important to get acquainted with your toolbox whatever the task or profession is.

Object-oriented Programming in TypeScript [\(source\)](#)

Object-Oriented Programming(OOP) is a type of programming paradigm which focuses more on problem solving, rather than coding itself. OOP is generally used with classes and object which are meant to represent real-life problems. It stores data in the form of attributes, while it codes in the form of procedures, and these procedures often interact with other objects and access and modify the data within those objects. One big advantage of OOP is encapsulation. Encapsulation is a concept that binds data together with the functions that manipulate said data and keep them safe from outside interference and misuse. In Typescript, this is simply done by using the keywords `public`, `private` and `protected` before the name of the variable, for example: `private name: type.`

TypeScript allows for a much simple Object-oriented style of programming as opposed to JavaScript because it doesn't use the strange syntax of prototyping in JavaScript. In TypeScript defining classes is immediately familiar and straightforward as you can see from the following code example:

```
class A{
    private x: number;
    private y: string;
    private z: any;
    constructor(x: number, y: string, z: any){
        this.x = x;
        this.y = y;
        this.z = z;
    }
    function foo(){
        alert(x + y + z);
    }
}
```

While in contrast, the same code, but in JavaScript, would be translated as:

```
var A = (function () {
    function A(x, y, z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }
    return A;
})();
function foo() {
    alert(x + y + z);
}
```

As we can see, JavaScript creates classes using various functions, and does so in a fairly roundabout way because of its functional nature and the fact that JavaScript was never meant to be an Object-Oriented programming language.

Creating members of a certain class in TypeScript is pretty simple, we need only to specify whether it's private, protected or public, then typing the name of the member, and finally, specify the type of the member preceded by colons. For example:

```
class class_name{
    private name: string;
    protected name2: number;
    public name3: any;
    ...
}
```

In TypeScript, as shown in the example above, classes are made by using the keyword `class` followed by the name of said class e.g. code** `class A{}`. Then, the class must have a

constructor which is used every time an object of the class is instantiated in order to construct said object. The syntax for creating a constructor is the following:

```
constructor(arg1: type, arg2: type, arg3: type...){
    this.member1 = arg1;
    this.member2 = arg2;
    this.member3 = arg3;
    ...
}
```

You may notice the conspicuous absence of any destructor in the first example of classes in TypeScript, that is because it doesn't have destructors, because any object that is no longer referenced will be deleted by the garbage collector that JavaScript uses.

Another important concept of OOP is inheritance and polymorphism, and TypeScript handles in its typical succinct style. Consider the following example

```
class Person{
    private name: string;
    private age: number;
    constructor(name:string, age:number){
        This.name = name;
        This.age = age;
    }
    function greet(){
        alert("Hello, my name is " + this.name + ", and I am " +
this.age.toString() + " years old.");
    }
}

class Employee extends Person{
    private title:string;
    private salary: number;
    constructor(name: string, age: number, title: string, salary: number){
        super(name, age);
        This.title = title;
        This.salary = salary;
    }
    Function greet(){
        super.greet();
        alert("And also I am a " + this.title + " and make $" + this.salary.toString() +
" per month");
    }
}
```

The above example represents one parent class: Person, and one inherited, child class: Employee. The inherited class has access to all, except the private data members of the parent class. A child class is defined in TypeScript by using the keyword `extends` followed by the name of its parent class i.e `class Child extends Parent`. By using the function `super()` you can access the functions and non-private data members of the parent class as well as its constructor. We

can see an example of polymorphism by examining the function `greet()` in both of the classes. In the child class, the function `greet()` is overridden and has a different definition than the `greet()` function in the parent class. This means that the `greet()` function can have different outputs depending on which class object it's called from.

Type annotations

Type annotations in TypeScript are lightweight ways to record the intended contract of the function or variable. In this case, we intend the `greeter` function to be called with a single string parameter. We can try changing the call `greeter` to pass an array instead:

```
function greeter(person: string) {
    return "Hello, " + person;
}

var user = [0, 1, 2];

document.body.innerHTML = greeter(user);
```

Re-compiling, you'll now see an error:

```
greeter.ts(7,26): error TS2345: Argument of type 'number[]' is not assignable to parameter of type 'string'.
```

Similarly, try removing all the arguments to the `greeter` call. TypeScript will let you know that you have called this function with an unexpected number of parameters. In both cases, TypeScript can offer static analysis based on both the structure of your code, and the type annotations you provide.

Notice that although there were errors, the `greeter.js` file is still created. You can use TypeScript even if there are errors in your code. But in this case, TypeScript is warning that your code will likely not run as expected.

By doing this example we understand that TypeScript is still in essence JavaScript but with added safety. By using types, by forcing the programmer to be more explicit we achieve a more strict way of coding, in which the chances of making faults or bugs is reduced. This forces even other programmers which examine or use the code to use it properly because they are forced to follow convention and input appropriate objects in functions that only accept those given types of objects, thus TypeScript is in a sense JavaScript with a debugging safety net. TypeScript in this manner allows for greater safety when working on projects with twenty, thirty or more people, it allows for division of labour amongst the workers in which every person can safely contribute and safely use the work of others. Clearly we see that TypeScript buys in the whole concept of object-orientated programming. But there will be other segments in which we will elaborate more on certain points introduced here. For now let's develop our sample a little further. Here we use an interface that describes objects that have a `firstName` and `lastName` field. In TypeScript, two types are compatible if their internal structure is compatible. This allows us to implement an interface just by having the shape the interface requires, without an explicit implements clause.

```

interface Person {
    firstName: string;
    lastName: string;
}

function greeter(person: Person) {
    return "Hello, " + person.firstName + " " + person.lastName;
}

var user = { firstName: "Jane", lastName: "User" };

document.body.innerHTML = greeter(user);

```

Classes

Finally, let's extend the example one last time with classes. TypeScript supports new features in JavaScript, like support for class-based object-oriented programming.

Here we're going to create a Student class with a constructor and a few public fields. Notice that classes and interfaces play well together, letting the programmer decide on the right level of abstraction.

Also of note, the use of public on arguments to the constructor is a shorthand that allows us to automatically create properties with that name.

```

class Student {
    fullName: string;
    constructor(public firstName, public middleInitial, public lastName) {
        this.fullName = firstName + " " + middleInitial + " " + lastName;
    }
}

interface Person {
    firstName: string;
    lastName: string;
}

function greeter(person : Person) {
    return "Hello, " + person.firstName + " " + person.lastName;
}

var user = new Student("Jane", "M.", "User");

document.body.innerHTML = greeter(user);

```

Re-run `tsc greeter.ts` and you'll see the generated JavaScript is the same as the earlier code. Classes in TypeScript are just a shorthand for the same prototype-based OO that is frequently used in JavaScript. We will dwell deep into classes in another segment.

Running your TypeScript web app

Now type the following in a file greeter.html:

```
<!DOCTYPE html>
<html>
  <head><title>TypeScript Greeter</title></head>
  <body>
    <script src="greeter.js"></script>
  </body>
</html>
```

Open greeter.html in the browser to run your first simple TypeScript web application!

We believe that it's important to note that in the greeter.html file when we add our script we are actually adding the compiled JavaScript version of our code and not the actual code that we typed in written in TypeScript, additionally we recommend that you add all of your scripts above the ending body tag </body> the reason being that when the page is being loading in the browser the person viewing the webpage actually gets to see the visual part of the web page first, its appearance. While the person viewing actually clicks or uses our page the scripts will have already loaded.

Libraries and frameworks that use TypeScript

Because of these productivity wins from cleaner ES6 code, autocompletion, and hinting from optional typing, TypeScript is being adopted into major projects, like the dynamic web application framework, which can also be used on mobile, AngularJS 2.0 and the open source framework for building amazing mobile applications, Ionic Framework 2.0, the text editor Visual Studio Code etc..

You can also leverage TypeScript in JavaScript projects not written in TypeScript in the first place. TypeScript has definition files that allow you to get these productivity boosts when coding almost in any environment from the Browser to Node.js. You get all the autocompletion and hinting as if they were written in TypeScript.

Angular ([source](#))

AngularJS is by far the most popular JavaScript framework available today for creating web applications. And now Angular 2 and TypeScript are bringing true object oriented web development to the mainstream, in a syntax that is strikingly close to Java 8.

This template consists of a mix of standard and custom HTML tags that represent respective components. In the example we in-lined the HTML. If we prefer to store the markup in a separate file (in this case application.html), we'll would use the property templateUrl instead of template, and the code of the AppComponent might look like this:

```

import {Component} from 'angular2/core';
import {Route, RouteConfig, RouterOutlet} from 'angular2/router';
import HomeComponent from '../home/home';
import NavbarComponent from '../navbar/navbar';
import FooterComponent from '../footer/footer';
import SearchComponent from '../search/search';
import ProductDetailComponent from '../product-detail/product-detail';

@Component({
  selector: 'auction-application',
  templateUrl: 'app/components/application/application.html',
  directives: [
    RouterOutlet,
    NavbarComponent,
    FooterComponent,
    SearchComponent,
    HomeComponent
  ]
})
@RouteConfig([
  {path: '/', component: HomeComponent, as: 'Home'},
  {path: '/products/:id', component: ProductDetailComponent, as: 'ProductDetail'}
])
export default class ApplicationComponent {}

```

Visual Studio Code ([source](#))

Visual Studio Code is a lightweight but powerful source code editor written in TypeScript, JavaScript and CSS which runs on the desktop and is available for Windows, MacOS and Linux. It comes with built-in support for JavaScript, TypeScript and Node.js and has a rich ecosystem of extensions for other languages (such as C++, C#, Python, PHP, Go) and runtimes (such as .NET and Unity). Here is an example of the source code written in TypeScript (source code taken from [here](#)):

```

'use strict';

import 'vs/editor/browser/editor.all';
import 'vs/editor/contrib/quickOpen/browser/quickOutline';
import 'vs/editor/contrib/quickOpen/browser/gotoLine';
import 'vs/editor/contrib/quickOpen/browser/quickCommand';
import 'vs/editor/contrib/inspectTokens/browser/inspectTokens';

import { createMonacoBaseAPI } from 'vs/editor/common/standalone/standaloneBase';
import { createMonacoEditorAPI } from 'vs/editor/browser/standalone/standaloneEditor';
import { createMonacoLanguagesAPI } from 'vs/editor/browser/standalone/standaloneLanguages';
import { EDITOR_DEFAULTS, WrappingIndent } from 'vs/editor/common/config/editorOptions';

```

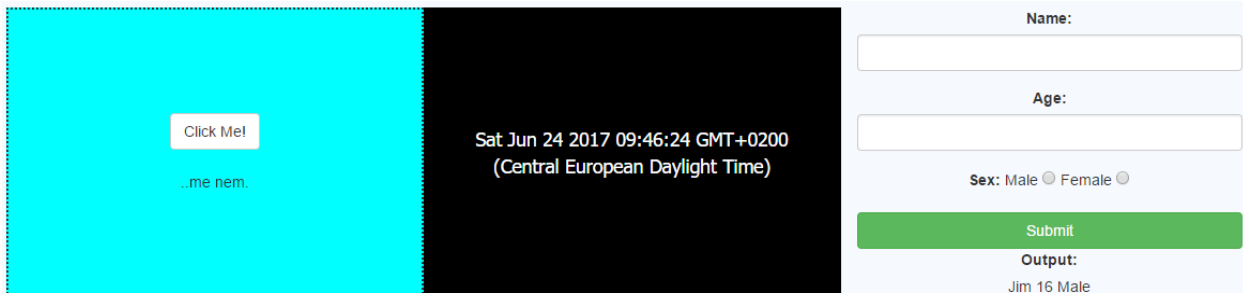
```
// Set defaults for standalone editor
(<any>EDITOR_DEFAULTS).wrappingIndent = WrappingIndent.None;
(<any>EDITOR_DEFAULTS.contribInfo).folding = false;
(<any>EDITOR_DEFAULTS.viewInfo).glyphMargin = false;

let base = createMonacoBaseAPI();
for (let prop in base) {
    if (base.hasOwnProperty(prop)) {
        exports[prop] = base[prop];
    }
}
exports.editor = createMonacoEditorAPI();
exports.languages = createMonacoLanguagesAPI();

var global: any = self;
global.monaco = exports;

if (typeof global.require !== 'undefined' && typeof global.require.config ===
'function') {
    global.require.config({
        ignoreDuplicateModules: [
            'vscode-languageserver-types',
            'vscode-languageserver-types/main',
        ]
    });
}
```

Our own implementation of TypeScript



The screenshot shows a web application interface. On the left, there is a cyan box containing a button labeled "Click Me!" and the text "...me nem.". In the center, there is a black box displaying the date and time: "Sat Jun 24 2017 09:46:24 GMT+0200 (Central European Daylight Time)". On the right, there is a form with the following elements: a "Name:" label above an input field, an "Age:" label above an input field, a "Sex:" label with radio buttons for "Male" and "Female", a green "Submit" button, and an "Output:" label above the text "Jim 16 Male".

How does it work?

First, for the black box containing the date, if you examine the `jef.html` file you can see that it contains only a `<p>` which holds only the word "Hello!" which means that by using the top three lines in `jef.ts` we have managed to changed "Hello!" to the current date.

Secondly, for the cyan box containing the button "Click me!". This example is similar to the first but it differs in that it allows for the user

to control when he wants the text to change, and upon a second click it does, before that only the background changes from cyan to pink. If you examine the `jef.ts` file you can notice a function called `changeBg()` which allows for this feature and another function named `changeText()` which allows for the change of text.

Lastly, the form contains three input fields, one for the name, age and sex of the user. Upon completion of the form's input fields and pressing the green button the text under the **Output:** label is changed to what the user has entered. What happens is that within the `jef.ts` file, more precisely in the `class Person` an object is created holding the appropriate information depending on the user's input. Afterwards similarly to the previous two example but with an added twist of using a in-class function `public print()` which returns a `string` we are able to see our output (which is of course, dependant on our input).

`jef.html`

```
<div class="row">
  <!-- MeNemJef Example -->
  <div id="btn-bg" class="col-md-4 bgBlue centeredElement jefBox">
    <div class="blockCenter">
      <button onclick="button()" id="button" class="btn btn-default center-block">Click
Me!</button>
      <br>
      <div id="textBox" class="block text-center">..me nem.</div>
    </div>
  </div>
  <!-- Banner Example -->
  <div class="col-md-4 banner text-center jefBox">
    <p id="banner" class="blockCenter text-center">Hello!</p>
  </div>
  <!-- Form Example -->
  <div class="col-md-4 text-center jefBox">
    <div class="form-group">
      <label for="name">Name:</label>
      <input type="text" class="form-control" id="name">
    </div>
    <div class="form-group">
      <label for="age">Age:</label>
```

```

        <input                type="age"                class="form-control"                id="age">
    </div>
    <div                                class="form-group">
        <label>Sex:</label>  Male <input id="sexMale" type="radio" name="sex">  Female
    <input                id="sexFemale"                type="radio"                name="sex">
    </div>
    <input  onclick="Submit()"  class="btn  btn-success  btn-block"  type="submit"
value="Submit">
    <label                for="outputBox">Output:                                </label>
    <div                id="outputBox"                class="block                text-center">..I                am.</div>
    </div>
</div>

```

jef.ts

```

let                banner:                HTMLElement;
banner
    =                document.getElementById("banner");
banner.innerText                =                Date();

let                i:                number                =                0;

function                button()                {
    if                (i                %                2                ==                0)                {
        changeBg();
        i++;
    }                else                {
        changeText();
        i++;
    }
}

let                btn_bg                =                document.getElementById("btn-bg");

function                changeBg()                {
    btn_bg.style.backgroundColor                =                "pink";
}

let                textBox:                HTMLElement                =                document.getElementById("textBox");

function                changeText()                {
    let                text:                string                =                textBox.innerText;
    textBox.innerText                =                text                +                ".jef.";
}

```

```

class Person {
    private name: string;
    private age: number;
    private sex: boolean;

    constructor(name: string, age: number, sex: boolean) {
        this.name = name;
        this.age = age;
        this.sex = sex;
    }

    public print() {
        let gender: string;
        if (this.sex == true) {
            gender = "Male";
        } else {
            gender = "Female";
        }
        return this.name + " " + this.age.toString() + " " + gender;
    }
}

let outputBox: HTMLElement = document.getElementById("outputBox");
let person: any = new Person("Jim", 16, true);
outputBox.innerText = person.print();

function Submit() {
    let Name: string = (<HTMLInputElement>document.getElementById("name")).value;
    let Age: string = (<HTMLInputElement>document.getElementById("age")).value;
    let Sex: boolean = (<HTMLInputElement>document.getElementById("sexMale")).checked;

    let inPerson: any = new Person(Name, +Age, Sex);
    outputBox.innerText = inPerson.print();
}

```

JavaScript which is transpiled from a part of our TypeScript file:

```

var Person = (function () {
    function Person(name, age, sex) {
        this.name = name;
        this.age = age;
    }
}

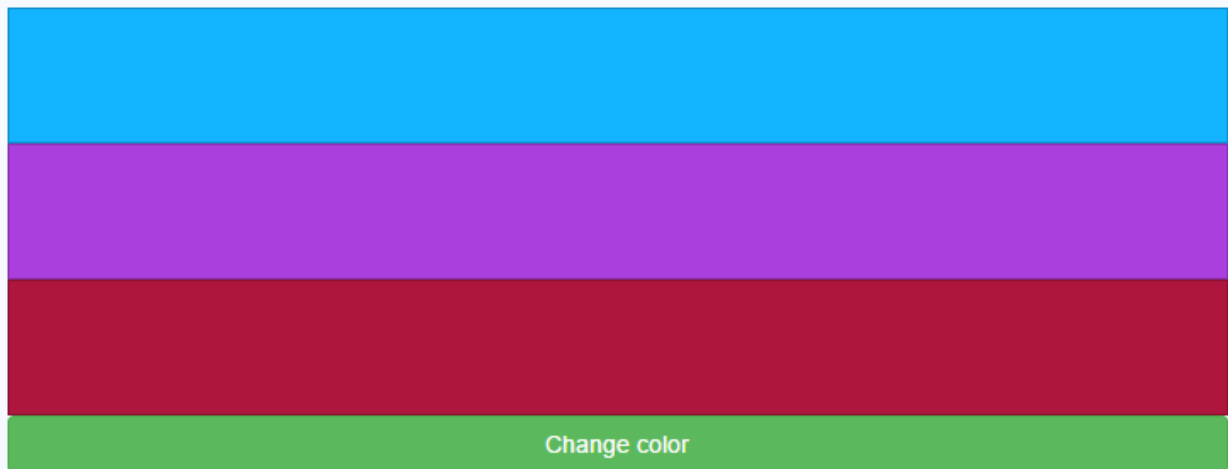
```

```

        this.sex = sex;
    }
    Person.prototype.print = function () {
        var gender;
        if (this.sex == true) {
            gender = "Male";
        }
        else {
            gender = "Female";
        }
        return this.name + " " + this.age.toString() + " " + gender;
    };
    return Person;
})();
var outputBox = document.getElementById("outputBox");
var person = new Person("Jim", 16, true);
outputBox.innerText = person.print();
var outputBox = document.getElementById("outputBox");
function Submit() {
    var Name = document.getElementById("name").value;
    var Age = document.getElementById("age").value;
    var Sex = document.getElementById("sexMale").checked;
    var inPerson = new Person(Name, +Age, Sex);
    outputBox.innerText = inPerson.print();
}

```

Click the green button



How does it work?

Notice that in the `src` attribute in the `<script>` tag we are adding a `.js` file called `flag.js` which is transpiled from the TypeScript file called `flag.ts`. Upon clicking the button we call the function `change)color()` which uses a for-loop to set the attributes of the three `<div>`s using an array.

flag.html

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <title>TypeScript Example</title>
  <link rel="stylesheet" type="text/css" href="flag.css" />
  <script type="text/javascript" src="flag.js"></script>
</head>

<body>
  <div id="line1" class="foo" style="background-color: blue; text-align: center;"></div>
  <div id="line2" class="foo" style="background-color: purple; text-align: center;"></div>
  <div id="line3" class="foo" style="background-color: wine; text-align: center;"></div>
  <hr>
  <button class="btn" onclick="change_color()">Change color</button>
</body>

</html>
```

flag.ts

```
function change_color() {
  let lines: any[] = document.getElementsByClassName("foo");
  let attributes: Array<string> =
    ["background: red;", "background: white;", "background: blue;"];
  console.log(lines);
  for (let i: number = 0; i < lines.length; i++) {
    console.log(i);
    console.log(lines[i]);
    lines[i].setAttribute("style", attributes[i]);
  }
}
```

Conclusion

From all that we have gathered about JavaScript and Typescript, we can come to the conclusion that JavaScript is ubiquitous in the web development world and because of that a web developer must adapt and learn how to code JavaScript. However JavaScript was never meant to be a disciplined, concise and strict language used to create big projects, and because of that it has many counter-intuitive quirks that have to be avoided by using some coding principles. But that will only mitigate the problems that arise with the aforementioned quirks, and using certain principles when coding with JavaScript can still lead to productivity loss and other problems. That is why we should use TypeScript, to reap the benefits of JavaScript, while solving most of its oddities while making it stable for constructing much bigger projects and from

the rise of TypeScript's popularity, we can see that TypeScript is indeed helping a lot of programmers to code websites more efficiently and with more stability.

In the end, we should always come back to the initial goal of creation and with that in mind we should choose our tools depending on the situation and needs. As we have hopefully successfully shown TypeScript is worthy of being used in major projects with many programmers, it relatively successfully solves and improves certain areas of JavaScript and where it doesn't improve at least it doesn't make it worse.

Literature

Additional Links:

1. [Book:](#)
2. [Functional JavaScript](#),
3. [Book:TypeScript](#),
4. [Blog:GettingStarted](#),
5. [Blog:TypeScriptHot](#),
6. [Blog:AngularTS](#),
7. [Blog:ClosuresinJS](#)
8. https://en.wikipedia.org/wiki/Functional_programming