Getting Started

Terms

Dynamically-typed Languages
IntelliSense
Refactoring
Source maps
Statically-typed Languages
Transpiling
Type safety

Summary

- Programming languages divide into two categories: statically-typed and dynamically-typed.
- In statically-typed languages (eg C++, C#, Java, etc), the type of variables is set at compile-time and cannot change later.
- In dynamically-typed languages (eg Python, JavaScript, Ruby), the type of variables is determined at run-time and can change later.
- TypeScript is essentially JavaScript with static typing and some additional features that help us write more concise and robust code.
- Most IDEs and code editors supporting TypeScript provide incredible IntelliSense and auto-completion. So we get active hints as we code. A great productivity booster!

- By providing type information in our code, we get better refactoring support in most IDEs and code editors.
- Refactoring means changing the structure of the code without changing its behavior.
- With TypeScript we can catch more bugs at compile time.
- Browsers don't understand TypeScript code. So we need to use the TypeScript compiler to compile and translate (or transpile) our TypeScript code into regular JavaScript for execution by browsers.
- Source maps are files that represent the mapping between TypeScript and JavaScript code. They're used for debugging.
- We can configure the TypeScript compiler by enabling/disabling various settings in tsconfig.json.

Fundamentals

Summary

- Since TypeScript is a superset of JavaScript, it includes all the built-in types in JavaScript (eg number, string, boolean, object, etc) as well as additional types (eg any, unknown, never, enum, tuple, etc).

- In TypeScript, we set the type of our variables by annotating them.

- The **any** type can represent any kind of value. It's something we should avoid as much as possible because it defeats the purpose of using TypeScript in the first place. A variable of type **any** can take any kind of value!

- Tuples are fixed-length arrays where each element has a specific type. We often use them for representing two or three related values.

- Enums represent a list of related constants.

Cheat Sheet

Annotation

```
let sales: number = 123_456_789;
let numbers: number[] = [1, 2, 3];
```

Tuples

```
let user: [number, string] = [1, 'Mosh'];
```

Enums

```
enum Size { Small = 1, Medium, Large };
```

Functions

```
function calculateTax(income: number): number {
  return income * .2;
}
```

Objects

```
let employee: {
    id: number;
    name: string;
    retire: (date: Date) => void
} = {
   id: 1,
   name: 'Mosh',
   retire: (date: Date) => {},
};
```

Compiler Options

Option	Description
`noImplicitAny`	When enabled, the compiler will warn you about variables that are inferred with the **any** type. You'll then have to explicitly annotate them with **any** if you have a reason to do so.
`noImplicitReturns`	When enabled, the compiler will check all code paths in a function to ensure they return a value.
`noUnusedLocals`	When enabled, the compiler will report unused local variables.
`noUnusedParameters`	When enabled, the compiler will report unused parameters.

Advanced Types

Summary

- Using a type alias we can create a new name (alias) for a type. We often use type aliases to create custom types.
- With union types, we can allow a variable to take one of many types (eg number | string).
- With intersection types, we can combine multiple types into one (eg Draggable & Resizable).
- Using optional chaining (?.) we can simplify our code and remove the need for null checks.
- Using the Nullish Coalescing Operator we can fallback to a default value when dealing with null/undefined objects.
- Sometimes we know more about the type of a variable than the TypeScript compiler. In those situations, we can use the **as** keyword to specify a different type than the one inferred by the compiler. This is called type assertion.
- The **unknown** type is the type-safe version of **any**. Similar to **any**, it can represent any value but we cannot perform any operations on an **unknown** type without first narrowing to a more specific type.
- The **never** type represents values that never occur. We often use them to annotate functions that never return or always throw an error.

Cheat Sheet

Type alias

```
type Employee = {
   id: number;
   name: string;
   retire: (date: Date) => void
```

Union types

```
let weight: number | string = 1;
weight = '1kg';
```

Intersection types

```
type UIWidget = Draggable & Droppable;
```

Literal types

```
type Quantity = 50 | 100;
```

Nullable types

```
let name: string | null = null;
```

Optional chaining (?.)

```
customer?.birthdate?.getFullYear();
customers?.[0];
log?.('message');
```


Nullish coalescing operator

```
someValue ?? 30
```

Type assertion

```
obj as Person
```

The unknown type

```
function render(document: unknown) {
  // We have to narrow down to a specific
  // type before we can perform any operations
  // on an unknown type.
  if (typeof document === 'string') {

  }
}
```

The never type

```
function processEvents(): never {
  // This function never returns because
  // it has an infinite loop.
  while (true) {}
}
```

Compiler Options

Option	Description
`strictNullChecks`	When enabled, null and undefined will not be acceptable values for variables unless you explicitly declare them as nullable. So, you'll get an error if you set a variable to null or undefined.
`allowUnreachableCode`	When set the false, reports error about unreachable code.

Classes and Interfaces

Summary

- Object-oriented programming is one of the many programming paradigms (styles of programming) in which objects are the building blocks of applications.

- An object is a unit that contains some data represented by properties and operations represented by methods.

- A class is a blueprint for creating objects. The terms class and object are often used interchangeably.

- We use access modifiers (public, private, protected) to control access to properties and methods of a class.

- A constructor is a special method (function) within a class that is called when instances of that class are created. We use constructors to initialize properties of an object.

- Static members are accessed using the class name. We use them where we need a single instance of a class member (property or method) in memory.

- Inheritance allows a class to inherit and reuse members of another class. The providing class is called the *parent*, *super* or *base* class while the other class is called the *child*, *sub* or *derived* class.

- An abstract class is a class with partial implementation. Abstract classes cannot be instantiated and have to be inherited.

- We use interfaces to define the shape of objects.

Cheat Sheet

Classes and constructors

```
class Account {
  id: number;

  constructor(id: number) {
    this.id = id;
  }
}

let account = new Account(1);
```

Accessing properties and methods

```
account.id = 1;
account.deposit(10);
```

Read-only and optional properties

```
class Account {
  readonly id: number;
  nickname?: string;
}
```

Access modifiers

```
class Account {
  private _balance: number;

  // Protected members are inherited.
  // Private members are not.
  protected _taxRate: number;
}
```

Parameter properties

```
class Account {
  // With parameter properties we can
  // create and initialize properties in one place.
  constructor(public id: number, private _balance: number) {
  }
}
```

Getters and setters

```
class Account {
  private _balance = 0;

  get balance(): number {
    return this._balance;
  }

  set balance(value: number) {
    if (value < 0)
      throw new Error();
    this._balance = value;
  }
```

Index signatures

```
class SeatAssignment {
  // With index signature properties we can add
  // properties to an object dynamically
  // without losing type safety.
  [seatNumber: string]: string;
}

let seats = new SeatAssignment();
seats.A1 = 'Mosh';
seats.A2 = 'John';
```


Static members

```
class Ride {
  static activeRides = 0;
}

Ride.activeRides++;
```

Inheritance

```
class Student extends Person {
}
```

Method overriding

```
class Student extends Person {
  override speak() {
    console.log('Student speaking');
  }
}
```

Abstract classes and methods

```
abstract class Shape {
  // Abstract methods don't have a body
  abstract render();
}

class Circle extends Shape {
  override render() {
    console.log('Rendering a circle');
  }
}
```


Interfaces

```
interface Calendar {
  name: string;
  addEvent(): void;
}

class GoogleCalendar implements Calendar {
}
```

Compiler Options

Option	Description
`noImplicitOverride`	When enabled, then compiler will warn us if we try to override a method without using the override keyword.

Generics

Summary

- Generics allow us to create reusable classes, interfaces and functions.
- A generic type has one or more generic type parameters specified in angle brackets.
- When using generic types, we should supply arguments for generic type parameters or let the compiler infer them (if possible).
- We can constrain generic type arguments by using the **extends** keyword after generic type parameters.
- When extending generic classes, we have three options: can pass on generic type parameters, so the derived classes will have the same generic type parameters. Alternatively, we can restrict or fix them.
- The **keyof** operator produces a union of the keys of the given object.
- Using type mapping we can create new types based off of existing types. For example, we can create a new type with all the properties of another type where these properties are readonly, optional, etc.
- TypeScript comes with several utility types that perform type mapping for us. Examples are: **Partial<T>**, **Required<T>**, **Readonly<T>**, etc.
- See the complete list of utility types:

  https://www.typescriptlang.org/docs/handbook/utility-types.html

Cheat Sheet

Generic classes

```
class KeyValuePair<K, V> {
  constructor(public key: K, public value: V) {}
}

let pair = new KeyValuePair<number, string>(1, 'a');

// The TypeScript compiler can sometimes infer
// generic type arguments so we don't need to specify them.
let other = new KeyValuePair(1, 'a');
```

Generic functions

```
function wrapInArray<T>(value: T) {
  return [value];
}

let numbers = wrapInArray(1);
```

Generic interfaces

```
interface Result<T> {
  data: T | null;
}
```


Generic constraints

```
function echo<T extends number | string>(value: T) {}

// Restrict using a shape object
function echo<T extends { name: string }>(value: T) {}

// Restrict using an interface or a class
function echo<T extends Person>(value: T) {}
```

Extending generic classes

```
// Passing on generic type parameters
class CompressibleStore<T> extends Store<T> { }

// Constraining generic type parameters
class SearchableStore<T extends { name: string }> extends Store<T> { }

// Fixing generic type parameters
class ProductStore extends Store<Product> { }
```

The keyof operator

```
interface Product {
  name: string;
  price: number;
}

let property: keyof Product;
// Same as
let property: 'name' | 'price';

property = 'name';
property = 'price';
property = 'otherValue'; // Invalid
```


Type mapping

```
type ReadOnly<T> = {
  readonly [K in keyof T]: T[K];
};

type Optional<T> = {
  [K in keyof T]?: T[K];
};

type Nullable<T> = {
  [K in keyof T]: T[K] | null;
};
```

Utility types

```
interface Product {
  id: number;
  name: string;
  price: number;
}

// A Product where all properties are optional
let product: Partial<Product>;

// A Product where all properties are required
let product: Required<Product>;

// A Product where all properties are read-only
let product: Readonly<Product>;

// A Product with two properties only (id and price)
let product: Pick<Product, 'id' | 'price'>;

// A Product without a name
let product: Omit<Product, 'name'>;
```

Decorators

Summary

- Decorators are often used in frameworks (eg Angular, Vue) to chance and enhance classes and how they behave.
- We can apply decorators on classes, properties, methods, parameters, and accessors (getters and setters).
- A decorator is just a function that gets called by the JavaScript runtime. In that function, we have a chance to modify a class and its members.
- To use decorators, we have to enable the **experimentalDecorators** setting in tsconfig.
- We can apply more than one decorator to a class or its members. Multiple decorators are applied in the reverse order.

Cheat Sheet

Class decorators

```
function Component(constructor: Function) {
  // Here we have a chance to modify members of
  // the target class.
  constructor.prototype.uniqueId = Date.now();
}

@Component
class ProfileComponent { }
```

Parameterized decorators

```
function Component(value: number) {
  return (constructor: Function) => {
    // Here we have a chance to modify members of
    // the target class.
    constructor.prototype.uniqueId = Date.now();
  };
}

@Component(1)
class ProfileComponent {}
```

Decorator composition

```
// Multiple decorators are applied in reverse order.
// Pipe followed by Component.
@Component
@Pipe
class ProfileComponent {}
```

Method decorators

```
function Log(target: any, methodName: string, descriptor: 
PropertyDescriptor) {
  // We get a reference to the original method
  const original = descriptor.value as Function;
  // Then, we redefine the method
  descriptor.value = function(...args: any) {
    // We have a chance to do something first
    console.log('Before');
    // Then, we call the original method
    original.call(this, ...args);
    // And we have a chance to do something after
    console.log('After');
  }
}

class Person {
  @Log
  say(message: string) {}
}
```

Accessor decorators

```
function Capitalize(target: any, methodName: string, descriptor: 
PropertyDescriptor) {
  const original = descriptor.get;
  descriptor.get = function() {
    const result = original.call(this);
    return 'newResult';
  }
}

class Person {
  @Capitalize
  get fullName() {}
}
```

Property decorators

```
function MinLength(length: number) {
  return (target: any, propertyName: string) => {
    // We use this variable to hold the value behind the
    // target property.
    let value: string;

    // We create a descriptor for the target property.
    const descriptor: PropertyDescriptor = {
      // We're defining the setter for the target property.
      set(newValue: string) {
        if (newValue.length < length)
          throw new Error();
        value = newValue;
      }
    }

    // And finally, we redefine the property.
    Object.defineProperty(target, propertyName, descriptor);
  }
}

class User {
  @MinLength(4)
  password: string;
}
```

Modules

Summary

- We use modules to organize our code across multiple files.
- Objects defined in a module are private and invisible to other modules unless exported.
- We use **export** and **import** statements to export and import objects from various modules. These statements are part of the ES6 module format.
- Over years, many module formats have been developed for JavaScript. Examples are CommonJS (introduced by Node), AMD, UMD, etc.
- We can use the **module** setting in tsconfig to specify the module format the compiler should use when emitting JavaScript code.

Cheat Sheet

Exporting and importing

```
// shapes.ts
export class Circle {}
export class Square {}

// app.ts
import { Circle, Square as MySquare } from './shapes';
```

Default exports

```
// shapes.ts
export default class Circle {}


// app.ts
import Circle from './shapes';
```

Wildcard imports

```
// app.ts
import * as Shapes from './shapes';

let circle = new Shapes.Circle();
```

Re-exporting

```
// /shapes/index.ts
export { Circle } from './circle';
export { Square } from './square';

// app.ts
import { Circle, Square } from './shapes';
```


Integration with JavaScript

Summary

- To include JavaScript code in a TypeScript project, we need to enable the **allowJs** setting in tsconfig.
- JavaScript code included in TypeScript projects is not type-checked by default.
- We can enable type checking by enabling the **checkJs** setting in tsconfig.
- We can optionally turn off compiler errors on a file-by-file basis by applying **`// @ts-nocheck`** once on top of JavaScript files.
- When migrating a large JavaScript project to TypeScript, we might face numerous errors. In such cases, it's easier to disable **checkJs** and apply **`// @ts-check`** (the opposite of **@ts-nocheck**) on individual files to migrate them one by one.
- We have two ways to describe type information for JavaScript code: using JSDoc and declaration (type definition files).
- Type definition files are similar to header files in C. They describe the features of a module.
- We don't need to create type definition files for third-party JavaScript libraries. We can use type definition files from the Definitely Typed GitHub repository (@types/<package>).
- Newer JavaScript libraries come with type definition files. So there's no need to install type definition files separately.

Additional Reading

Converting a React Project to Use TypeScript

Follow the exact steps outlined on React docs:

https://reactjs.org/docs/static-type-checking.html#adding-typescript-to-a-project

This is the official source for converting a React JavaScript project to a React TypeScript project. If the above link is broken, just Google "React TypeScript" and find the corresponding page on React docs.

React TypeScript Cheat Sheet

Here you can find some additional details involved when using TypeScript to build complex React applications.

https://react-typescript-cheatsheet.netlify.app/docs/basic/setup/

Remember: Just because this website includes several complex examples doesn't mean you have to use them all! React is a tool. TypeScript is another tool. You use these tools to solve real-world problems. Sometimes you need very little of a given tool to solve a real-world problem. It's okay not to know all the features of a given tool. As you build more and more applications, you'll discover new features of the tools you use everyday.