Introduction

TypeScript

Info

TypeScript is a super-set of JavaScript and all plain JS is valid TypeScript that the TS compiler will compile any existing JS application without the need for adding any type annotations.

TypeScript compiles TS code into a target JS format. The target output can be configured with a tsconfig.json file in the root folder of the project. Targets include ES2015/E6+, commonjs and more.

Basic Types

Primitives: **string, number, boolean**

Special: **any, unknown, void, null, undefined**

Note on the 'any' type:

When you don't specify a type, and TypeScript can't infer it from context, the compiler will typically default to any.

You usually want to avoid this, though, because any isn't type-checked. Use the compiler flag 'noImplicitAny' to flag any implicit any as an error.

Type Annotation

Functions

Without TS:

```
function greet(name) {
    console.log('Hello ' + name + '!');
}

greet('Name') // Hello Name!
greet(42) // Hello 42!
greet({ a: 'apple' }) // Hello [Object object]!
```

With TS:

```
function greet(name: string): void {
    console.log('Hello ' + name + '!');
}

greet('Name') // Hello Name!
greet(42) // Argument of type 'number' is not assignable to parameter of type 'string'.
greet({ a: 'apple' }) // Argument of type '{ a: string; }' is not assignable to parameter of type 'string'.
```

Objects

Without TS:

```
const user = {
    name: 'Name',
    id: '1234',
};

user.address = '123 Fake St.';
user.email = 'e@mail.com';
```

With TS:

```
const user: {
    name: string;
    id: string;
    address?: string;
} = {
    name: 'Name',
    id: '1234',
}

user.address = '123 Fake St.';
user.email = 'e@mail.com';
```

Complex Types

Interfaces

```
interface User {
    id: string;
    created: Date;
    contact: {
        firstName: string;
        lastName: string;
        middleName?: string;
        address: {
            street: string;
            city: string;
            zip: string;
            country: string;
        };
    };
}
```

Types

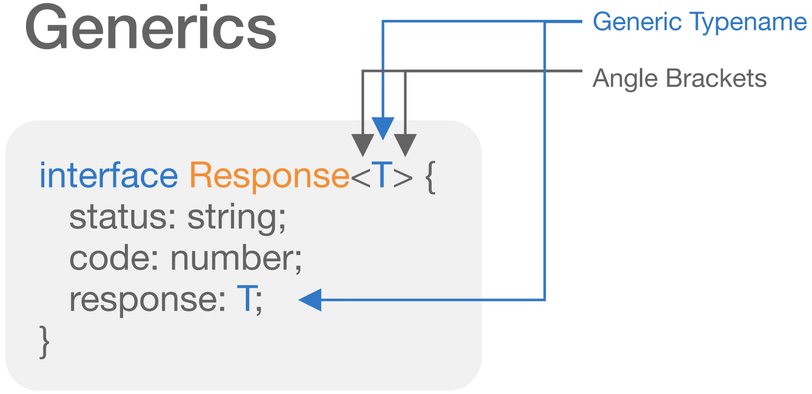
```
type UserResponse = User | null

type CallbackFunction =
    (user: UserResponse, error?: Error) => void;

function getUser(id: string, cb: CallbackFunction): void {
    const user = database.getUser(id);
    if (!user) {
        return cb(null, new Error('No user found'));
    }
    cb(user);
}
```

Classes Array, Date, URL, Promise, etc.

```
class Storage {

  private users: Array<User> = [];

  constructor(private readonly name: string) {}

  static toString(s: Storage): string {
    let out = s.name + ':\n';
    out += s.users.length + ' number of users';
    return out;
  }

  public addUser(u: User): void {
    this.users.push(u);
  }
}
```

Generic Types

Generics




```
interface Response<T> {
  status: string;
  code: number;
  response: T;
}
```

```
type UserResponse = Response<User>;

function generateOkayResponse<T>(response: T): Response<T> {
  return {
    status: ‘success’,
    code: 200,
    response,
  };
}

function generateOkayResponseUser(user: User): UserResponse {
  return {
    status: ‘success’,
    code: 200,
    response: user,
  };
}
```

Generic Class

```
class Container<T> {

  private items: Array<T> = [];

  public push(item: T): void {
    this.items.push(item);
  }

  public get(index: number): T | null {
    const item = this.items[index];
    if (item) {
      return item;
    }
    return null;
  }

}
```

Array

You can annotate arrays as such:

```
const users: Array<User> = [];

const users: User[] = [];
```

Both are valid

Enum Types

```
enum Keys {
  Up = ‘up’,
  Down = ‘down’,
  Left = ‘left’,
  Right = ‘right’,
  A = ‘a’,
  B = ‘b’,
  Start = ‘start’,
  Pause = ‘pause’,
  None = ‘none’,
}
```

```
enum Actions {
  Jump = ‘jump’,
  Crouch = ‘crouch’,
  MoveLeft = ‘move:left’,
  MoveRight = ‘move:right’,
  Punch = ‘punch’,
  PickUpItem = ‘pickup:item’,
  None = ‘none’,
}
```

```
class Player {

  private inventory: Array<InventoryItem> = [];

  public onKey(keyString: string): void {
    const key = this.getKey(keyString);
    const action = this.getAction(key);
    this.onAction(action);
  }

  private getKey(key: string): Keys {}

  private getAction(key: Keys): Action {}

  private onAction(action: Action): void {}

}
```

Enums are one of the few features TypeScript has which is not a type-level extension of JavaScript.

Enums allow a developer to define a set of named constants. Using enums can make it easier to document intent, or create a set of distinct cases. TypeScript provides both numeric and string-based enums.

Utility Types, Unions and Intersections

Builtin Utility Types

Record<Keys, Type>

Constructs an object type whose property keys are Keys and whose property values are Type. This utility can be used to map the properties of a type to another type.

```
type TemperatureMap = Record<string, Array<number>>;

const temps: TemperatureMap = {
  Denver: [5, 3, -2, 17],
  Tulsa: [63, 51, 43, 71],
  Chicago: [-5, -12, 2, 7],
}
```

Partial<Type>

Constructs a type with all properties of Type set to optional. This utility will return a type that represents all subsets of a given type.

```
interface Book { title: string; author: string; dates: Array<Date>; }

function updateBook(book: Book, update: Partial<Book>): Book {
  return { …book, …update };
}
```

Pick<Type, Keys>

Constructs a type by picking the set of properties Keys (string literal or union of string literals) from Type.

```
function bookPreview(book: Book): Pick<Book, 'author' | 'title'> {
  return { author: book.author, title: book.title };
}
```

Omit<Type, Keys>

Constructs a type by picking all properties from Type and then removing Keys (string literal or union of string literals).

```
function bookPreview(book: Book): Omit<Book, 'dates'> {
  return { author: book.author, title: book.title };
}
```

Unions and Intersections

Unions

```
interface Book { title: string; author: string; id: number; }
interface Album { title: string; artist: string; id: number; }
interface Painting { title: string; painter: string; id: number; }
type HasId = Book | Album | Painting; // { id: number }

function getItemById(item: HasId): Book | Album | Painting {
  return database.getItem(item.id);
}
```

Intersections

```
interface Book { title: string; author: string; }
interface Album { title: string; artist: string; }
interface Painting { title: string; painter: string; }
interface HasId { id: number; }
type BookWithId = Book & HasId;
// { title: string; author: string; id: number }
type Indexable<T> = T & HasId;

function index<T>(item: T): Indexable<T> {
  return { …item, id: generateId(), };
}
```