Typescript

Types let developers write more explicit "contracts". In other words, things like function signatures are more explicit.

Without TS:

```
function add(a, b) {
  return a + b;
}

add(1, 3);   // 4
add(1, '3'); // '13'
```

With TS:

```
function add(a: number, b: number) {
  return a + b;
}

add(1, 3);   // 4
// compiler error before JS is even produced
add(1, '3'); // '13'
```

Types

Many people do not realize it, but JavaScript *does* in fact have types, they're just "duck typed", which roughly means that the programmer does not have to think about them. JavaScript's types also exist in TypeScript:

- `boolean` (true/false)
- `number` integers, floats, `Infinity` and `NaN`
- `string` characters and strings of characters
- `[]` Arrays of other types, like `number[]` or `boolean[]`
- `{}` Object literal
- `undefined` not set

TypeScript also adds

- `enum` enumerations like `{ Red, Blue, Green }`
- `any` use any type
- `void` nothing

```
let isDone: boolean = false;
let height: number = 6;
let name: string = "bob";
let list: number[] = [1, 2, 3];
let list: Array<number> = [1, 2, 3];
enum Color {Red, Green, Blue};
let c: Color = Color.Green;
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a boolean

function showMessage(data: string): void {
  alert(data);
}
showMessage('hello');
```

TypeScript Classes

TypeScript also treats `class`es as their own type:

```
class Foo { foo: number; }
class Bar { bar: string; }

class Baz {
  constructor(foo: Foo, bar: Bar) { }
}

let baz = new Baz(new Foo(), new Bar()); // valid
baz = new Baz(new Bar(), new Foo());     // tsc errors
```

Like function parameters, `class`es sometimes have optional members. The same `?:` syntax can be used on a `class` definition:

```
class Person {
  name: string;
  nickName?: string;
}
```

Shapes

Underneath TypeScript is JavaScript, and underneath JavaScript is typically a JIT (Just-In-Time compiler). Given JavaScript's underlying semantics, types are typically reasoned about by "shapes". These underlying shapes work like TypeScript's interfaces, and are in fact how TypeScript compares custom types like `class`es and `interface`s.

Consider an expansion of the previous example:

```
interface Action {
  type: string;
}

let a: Action = {
    type: 'literal'
}

class NotAnAction {
  type: string;
  constructor() {
    this.type = 'Constructor function (class)';
  }
}

a = new NotAnAction(); // valid TypeScript!
```

Type Keyword

The `type` keyword defines an alias to a type.

```
type str = string;
let cheese: str = 'gorgonzola';
let cake: str = 10; // Type 'number' is not assignable to type 'string'
```

Union Types

Union types allow type annotations to specify that a property should be one of a set of types (either/or).

```
function admitAge (age: number|string): string {
  return `I am ${age}, alright?!`;
}

admitAge(30); // 'I am 30, alright?!'
admitAge('Forty'); // 'I am Forty, alright?!'
```

Intersection Types

Intersection types are the combination of two or more types. Useful for objects and params that need to implement more than one interface.

```
interface Kicker {
  kick(speed: number): number;
}

interface Puncher {
  punch(power: number): number;
}
// assign intersection type definition to alias KickPuncher
type KickPuncher = Kicker & Puncher;

function attack (warrior: KickPuncher) {
  warrior.kick(102);
  warrior.punch(412);
  warrior.judoChop(); // Property 'judoChop' does not exist on type 'KickPuncher'
}
```

Function Type Definitions

Function type annotations can get much more specific than typescripts built-in `Function` type. Function type definitions allow you to attach a function signature to it's own type.

```
type MaybeError = Error | null;
type Callback = (err: MaybeError, response: Object) => void;

function sendRequest (cb: Callback): void {
  if (cb) {
    cb(null, {});
  }
}
```