TypeScript

- *Superset* of JavaScript (a.k.a. JavaScript++) to make it easier to program for large-scale JavaScript projects
  - New features: types, interfaces, decorators, . . .
  - All additional TypeScript features are *strictly optional* and are not required
  - Any JavaScript code is also a TypeScript code!
- *Transpilation*: TypeScript code is "compiled" to a JavaScript code using TypeScript compiler

```
// --- hello.ts ---
function hello(name: string): string
{
    return "Hello " + name;
}
console.log(hello("world!"));
```

```
$ tsc hello.ts
```

The above command runs the TypeScript compiler `tsc` on `hello.ts` and produces the `hello.js` file, which contains a standard JavaScript code.

```
$ node hello.js
Hello world!
```

Types

- Types can be added to functions and variables as an intended "contract"

```
function hello(name: string): string
{
    return "Hello " + name;
}
let user = [0, 1, 2];
hello(user);
```

- Compiler produces an error for the above code due to type mismatch

```
$ tsc hello.ts
hello.ts(6,33): error TS2345: Argument of type 'number[]' is
   not assignable to parameter of type 'string'.
```

  - Use `any` type to specifically indicate that any type is possible
  - Use `void` as the return type of a function with no return value

- Q: Why would anyone want this?
  - Compile-time error vs run-time error
  - Rigidity vs flexibility

Classes

- TypeScript allows explicit declaration of class properties, including `public`, `private`, `protected` access levels
  - JavaScript syntax

```
// JavaScript -- point.js
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
}
```

  - TypeScript syntax

```
class Point {
    x: number;
    private y: number;

    constructor (x, y) {
        this.x = x;
        this.y = y;
    }
}
let p = new Point(10, 20);
console.log(p.x);
```

    * Property with no access-level keyword becomes `public`

- Adding access-level keyword to constructor automatically adds the property

```
class Point {
    constructor (public x: number, private y: number) {}
}
let p = new Point(10, 20);
console.log(p.x);
```

  - This code is equivalent to the previous code

Interfaces

- Like Java, TypeScript supports interfaces
- Two types are compatible if their internal structure is compatible
  - We can implement an interface simply by having the needed structure of the interface, without an explicit `implements` clause

```
interface Person {
    firstName: string;
    lastName: string;
}
function hello(person: Person) {
    return "Hello, " + person.firstName + " " + person.
       lastName;
}
let user = { firstName: "Jane", lastName: "User" };
hello(user);
```

  - No error in the above example because `user` is compatible with `Person`

Generics

- Like Java generics, TypeScript allows creating generic functions/classes using parameterized types
- Example

```
class Pair<T> {
    x: T;
    y: T;
    constructor(x: T, y: T) {
        this.x = x;
        this.y = y;
    }
}
let p = new Pair<number>(1, 2);

function log<T>(arg: T) : void
{
    console.log(arg);
}
log<number>(1);
```

Decorators

- We can "decorate" classes, methods, properties, and parameters using a *decorator*
  - Syntax: `@decorator`
  - Example:

```
@sealed    // <- decorator
class Greeter {
    greeting: string;
    constructor(greeting: string) {
        this.greeting = greeting;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}
```

        - Interpretation: “objects of this class are *sealed*!”
        - In JavaScript, “sealing” means
            - no new property and method can be added and
            - their “attributes” (such as enumerable, writable) cannot be changed
    - Technically, decorators are functions that modify JavaScript classes, properties, methods, and parameters
- General syntax for decorator: `@expression`
    - `expression` must be (or evaluate to) a function, and it will be called at runtime with the decorated entity as its parameter(s)
        - Class decorators get the constructor of the class as its parameter
    - Example: possible implementation of the above `@sealed` decorator:

```
function sealed(constructor: Function) {
    Object.seal(constructor);    // seal the constructor
    Object.seal(constructor.prototype)  // seal its prototype
}
```

        - This example effectively seals any object of the class