TypeScript Notes

• TypeScript detects errors before runtime. (static checking)

• “tsc fileName” command compiles TypeScript files and outputs Javascript with .js extension.

Type Annotation

let myVar: type = value
type: string, number, boolean

• Once you define the type, you can’t reassign it to a different type.

• Suggests proper methods based on the type. Ex. toUpperCase( ) for string.

• Only one number type. No distinctions like float, int, etc.

• Any is a unique type to TS. Turns off type checking for variables. Its use is not recommended because it is contrary to the intended use of the TS.

• Never is the unique type to TS. Used for functions that should never return.

For example:

1. A function always throws an exception.

2. A function has an infinite loop and hence never finishes executing.

• Type annotation is not necessary because of type inference. TS compiler can infer types based on the values and later enforce that type.

TypeScript Notes

• Sometimes TS cannot infer types, and adding an annotation is good practice.

```
const fruits = ["apple", "cherry", "strawberry", "banana"];
// cannot infer type
let found: boolean;
for (let fruit of fruits) {
    if (fruit === "apple") {
        found = true;
    }
}
```

Functions

• Can specify the type of function parameters.

```
function f(myVar: type)
const f = (myVar: type) => {};
```

• To specify the default value of the function along with its type:

```
function f(myVar: string = “hi there”)
```

In this case, you may not do the type annotation, TS can determine the type from the default value.

• Can specify the type returned by a function. TS can often infer this.

```
function f(myVar: paramType): returnType
const f = (myVar: paramType): returnType => {}
```

• The return type is void if no type is defined. Therefore, it will not throw an error if the type is not explicitly specified and the return keyword is forgotten. It is good practice to add a return type to avoid such situations.

TypeScript Notes

• In anonymous functions TS can infer the type of the parameter based on the context it is called in.

```
const days = ["monday", "tuesday", "wednesday"];

// infers day as string
days.map(day => {
    return day.toUpperCase();
});
```

• In anonymous functions TS can infer the type of the parameter based on the context it is called in.

```
const numbers = [1, 2, 3];

// ERROR: Property 'toUpperCase' does not exist on type 'number'.
numbers.forEach(num => {return num.toUpperCase();});
```

Objects

• There are various usages where type annotation for objects is very helpful.

If the function accepts an object as a parameter, TS can check whether the user provided the correct format. (Wrong type, missing parameter)

```
const greet = (name: {first: string; last: string}) => {
    return `Hello ${name.first} ${name.last}`;
}

// ERROR: Type 'number' is not assignable to type 'string'.
greet({first: 1, last: 2});

// ERROR: Property 'last' is missing in type.
greet({first: "Mel"});
```

TypeScript Notes

• You can declare variables in objects as optional. And errors about missing properties will no longer be shown. In the example above last is an optional property.

```
const greet = (name: {first: string; last?: string}) => {
    return `Hello ${name.first} ${name.last}`;
}

// NO ERROR
greet({first: "1"});
```

• To declare objects as a variable:

```
let myName: {first: string, last: string} = {first: "X", last: "Y"};
```

• Index signatures can be used for objects without a defined list of properties.

```
const groceryList: { [index: string]: number } = {};

// no error
groceryList.Tomato = 25;

// Error: Type 'string' is not assignable to type 'number'.
groceryList.Potato = "Fifty";
```

• There is one interesting fact about excess properties in TS, it does not check for errors unless excess properties are explicitly defined.

```
const greet = (name: {first: string; last: string}) => {
    return `Hello ${name.first} ${name.last}`;
}

// ERROR: Object literal may only specify known properties,
// and 'age' does not exist in type
greet({first: "X", last: "Y", age:25});

const person = {first: "X", last: "Y", age:25};

// NO ERROR
greet(person);
```

TypeScript Notes

• Can declare values as read-only, this way values cannot be changed after initialization.

```
type Invitation = {
    readonly place: string;
    isAccepted: boolean;
}

// no error when setting initial values
const invitation: Invitation = {
    place: "Conference Room",
    isAccepted: false
};

// ERROR: Cannot assign to 'place' because it is a read-only property.
invitation.place = "Meeting Room";
```

Type Alias

Allows us to reuse types.

In the functions greet, goodbye and randomNameGenerator type `{first: string; last: string}` is used and each time the type definition is made again.

```
const greet = (name: {first: string; last: string}) => {
    return `Hello ${name.first} ${name.last}`;
}

const goodbye = (name: {first: string; last: string}) => {
    return `Bye ${name.first} ${name.last}`;
}

const randomNameGenerator = (): {first: string; last: string} => {
    return {first: Math.random().toString(36), last:
Math.random().toString(36) };
}
```

TypeScript Notes

With type alias same code can be written like this:

```
type fullName = {
    first: string;
    last: string;
};


const greet = (name: fullName) => {
    return `Hello ${name.first} ${name.last}`;
}

const goodbye = (name: fullName) => {
    return `Bye ${name.first} ${name.last}`;
}


const randomNameGenerator = (): fullName => {
    return {first: Math.random().toString(36), last:
Math.random().toString(36) };
}
```

Arrays

• To annotate arrays that can allow data of only one type write type and empty brackets:

```
let cities: string[] = ["Paris", "New York"];

let cities: Array<string> = ["Paris", "New York"];
```

• To create a type for multidimensional arrays add more brackets, each bracket corresponds to a dimension.

```
let pixels: number[][] = [[2, 1], [2, 4]];
```

TypeScript Notes

Union Types

• Union types are used when a value can be more than a single type. Accessing type-specific methods when union types are being used to will result in type errors. To solve this be sure that you narrow it to this specific type.

```
type value = string | number;
```

• Using union types with literal types can offer very useful uses.

```
type MouseEvent = 'click' | 'mouseup' | 'mousedown';
let mouseEvent: MouseEvent;
mouseEvent = 'click'; // valid
mouseEvent = 'mouseup'; // valid
mouseEvent = 'mousedown'; // valid
mouseEvent = 'mouseover'; // compiler error
```

Tuples

• Exclusive type to TS, doesn't exist in JS.

• A *tuple type* is another sort of Array type that knows exactly how many elements it contains, and exactly which types it contains at specific positions.

```
let myTuple: [number, string, boolean, string];
myTuple = [10, "X", true, "Y"];
myTuple = ["X", true, "Y"]; // ERROR
myTuple = [10, "X", true, 2]; // ERROR
```

• Can be useful for defining RGB.

TypeScript Notes

Enums

• Exclusive type to TS, doesn't exist in JS.

• Enums allow us to define a set of named constants.

```
enum OrderStatus {
    PREPARING,
    SHIPPED,
    DELIVERED,
}

let currentStatus = OrderStatus.DELIVERED;
```

• TS specific pieces of code are generally not included in the JS file. But this is different with enums. Adds code like the below in JS file:

```
var OrderStatus;
(function (OrderStatus) {
    OrderStatus[OrderStatus["PENDING"] = 0] = "PENDING";
    OrderStatus[OrderStatus["SHIPPED"] = 1] = "SHIPPED";
    OrderStatus[OrderStatus["DELIVERED"] = 2] = "DELIVERED";
})(OrderStatus || (OrderStatus = {}));
```

• Complicating JS files with this kind of code is not a good practice. For this reason, a cleaner code is obtained when defining the enum as follows:

```
const enum OrderStatus {
    PENDING,
    SHIPPED,
    DELIVERED
}
```

TypeScript Notes

Interfaces

- Very similar to type aliases.

- Helps to create reusable types that describe the shapes of objects.

```
interface name {
    first: string;
    last: string;
}
```

- Can create read-only and optional properties just like with types.

```
interface name {
    readonly first: string;
    last?: string;
}
```

- Can be used to specify the format of objects.

```
interface name {
    first: string;
    last: string;
    getInitials(): string;
}

const myName: name = {
    first: "Mel",
    last: "Codes",
    getInitials() {return "MC";}
}
```

TypeScript Notes

• Interfaces can be extended. (inheritance)

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

interface Employee extends Person {
   empCode:number
}
```

• Interface can extend multiple interfaces.

```
interface Manager extends Person, Employee {
    manage(): void
}
```

Types vs. Interfaces

1. Interfaces can be reopened and new properties can be added to them.

• Code with the interface:

```
interface name {
    first: string;
}

interface name {
    last: string;
}

// myName has both first and last properties (both interfaces are combined)
const myName: name = {
    first: "X",
    last: "Y"
}
```

TypeScript Notes

• Same structure of code with type:

```
type name = {
    first: string;
}

// ERROR: Duplicate identifier 'name'.
type name = {
    last: string;
}

// ERROR: 'last' does not exist in type 'name'
const myName: name = {
    first: "X",
    last: "Y"
}
```

2. Unlike an interface that is only describing objects, the type alias can also be used for other types such as primitives, unions, and tuples.

```
// primitive
type name = string;

// object
type coordinates = {
    x: number;
    y: number;
};

// union
type PartialPoint = coordinates| name;

// tuple
type Data = [number, string];
```

3. Both can be extended. Also they can extend each other.

```
interface myInterface { x: number; }

type myType = { x: number; }

// interface extends interface

interface extendedInterface extends myInterface {
    y: number;
}

// interface extends type alias
interface extendedInterface extends myType {
    y: number;
}

// type alias extends type alias
type extendedType = myType & {
    y: number;
}


// type alias extends interface
type extendedType = myInterface & {
    y: number;
}
```

4. A class can implement an interface or type alias, both in the same way. But class can not implement type alias that consists of union types.

```
interface Point {
  x: number;
  y: number;
}

class SomePoint implements Point {
  x = 1;
  y = 2;
}
```

TypeScript Notes

```ts
type Point2 = {
  x: number;
  y: number;
};

class SomePoint2 implements Point2 {
  x = 1;
  y = 2;
}

type PartialPoint = { x: number; } | { y: number; };

// ERROR: can not implement a union type
class SomePartialPoint implements PartialPoint {
  x = 1;
  y = 2;
}
```

Classes

Defining class fields: It was sufficient to say this.variable = variable in JS, but in TS we need to define the fields separately with their types.

JS

```js
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
}
```

TS

```ts
class Person {
  name: string;
  age: number;

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

TypeScript Notes

There is also a shorter version for parameter properties, with this version even initialization like “this.name = name” is not required.

```
class Person {
  constructor(public name: string, public age: number) { }
}
```

Getters and setters: TS makes getter methods by default read-only and will produce an error when we try to set it.

```
class Employee {
  constructor(private _id: number) {
  }
  get id() {
    return this._id;
  }
  set id(newId: number) {
    this._id = newId;
  }
}
```

To access getters and setters, the property access syntax is used instead of the method call syntax.

```
// accessing getter
emp.id;

// accessing setter
emp.id = 2;
```

Modifiers: There are three types of access modifiers in TS: public, private and protected. Default modifier is public.

• The protected access modifier is similar to the private access modifier, except that protected members can be accessed using their deriving classes.

TypeScript Notes

Interfaces and classes: Classes can implement interfaces.

```
interface GPS {
    x: number;
    y: number;
    startGPS(): void;
}

class Smartphone implements GPS {
  constructor(public name: string, public x: number, public y: number) {}
  startGPS() {}
}
```

Abstract classes: Classes whose instances cannot be created.

• Defines the methods and fields for derived classes to extend.

• Similar to interfaces, but while interfaces only define type, abstract classes also have methods that work.

```
abstract class Employee {
  constructor(public name: string, public age: number) {}
  abstract work(): string;
  greet() { console.log("Welcome!"); }
}

class Manager extends Employee {
  constructor(name: string, age: number, public department: string) {
    super(name, age);
  }
  work() {
    return "Manage the project...";
  }
}
```

TypeScript Notes

Generics

• Defines reusable functions and classes that work with multiple types.

```
function f<Type>(item: Type): Type {}
```

• TS can infer type based on given parameter value.

```
function getRandomElement<T>(list: T[]): T {
  const rand = Math.floor(Math.random() * list.length);
  return list[rand];
}

// function getRandomElement<number>(list: number[]): number
getRandomElement<number>([1, 2, 5, 3, 8, 9, 6]);

// function getRandomElement<number>(list: number[]): number
getRandomElement([1, 2, 5, 3, 8, 9, 6]);
```

• In React < > symbols are used to create JSX tags. Since the same symbols are used when defining generics in TS, React automatically tries to complete it as <T></T>. To prevent this and to use generics in anonymous functions, a comma is added next to T.

```
const getRandomElement = <T, >(list: T[]): T ⇒ { }
```

• There is no problem with the normal function definition, so there is no need to add commas.

```
function getRandomElement<T>(list: T[]): T {}
```

• To define generics with multiple types:

```
function f<T, U>(obj1: T, obj2: U) {}
```

TypeScript Notes

• The extends keyword is used to avoid assigning any type to generics and to make type constraints.

1. Create an interface that describes constraints.
2. Extend this interface.

```
interface Lengthwise {
  length: number;
}

function loggingIdentity<Type extends Lengthwise>(arg: Type): Type {
  // Now we know it has a .length property, so no more error
  console.log(arg.length);
  return arg;
}
```

• Setting default type:

```
function f<Type = defaultType>(item: Type): Type {}
```

Type Narrowing

Typeof: With primitive types, typeof works as expected but with other types, it will not give all information about types.

```
// string
typeof "hello"

// number
typeof 123

// object
typeof [1,2,3]
```

TypeScript Notes

in operator: Checks if a certain property exists in an object.

```
interface Person {
  firstName: string;
  surname: string;
}
interface Company {
  name: string;
}
type Contact = Person | Company;

function sayHello(contact: Contact) {
  if ("firstName" in contact) {
    // (parameter) contact: Person
    console.log(contact.firstName);
  }
  if ("name" in contact) {
    // (parameter) contact: Company
    console.log(contact.name);
  }
}
```

instanceof: Checks if one thing is an instance of another. It is a JS operator. In the example above, if the Person and Company interfaces were defined as classes, we would not be able to use "typeof". Instead, we would have to use "instanceof".

TypeScript Notes

type predicates: Specific to TS.

• Writing custom functions that can narrow the type of a value.

• A common naming convention for these functions is to start with is.

• If this function returns true then the return part helps narrow the type.

```
function isString(s) {
  return typeof s === 'string';
}

function toUpperCase(x: unknown) {
  if(isString(x)) {
    x.toUpperCase(); // x is unknown
  }
}

// if return is true then s is string narrows the type
function isString(s): s is string {
  return typeof s === 'string';
}

function toUpperCase(x: unknown) {
  if(isString(x)) {
    x.toUpperCase(); // x is string
  }
}
```