TypeScript

LUXOFT

LXFT LISTED NYSE

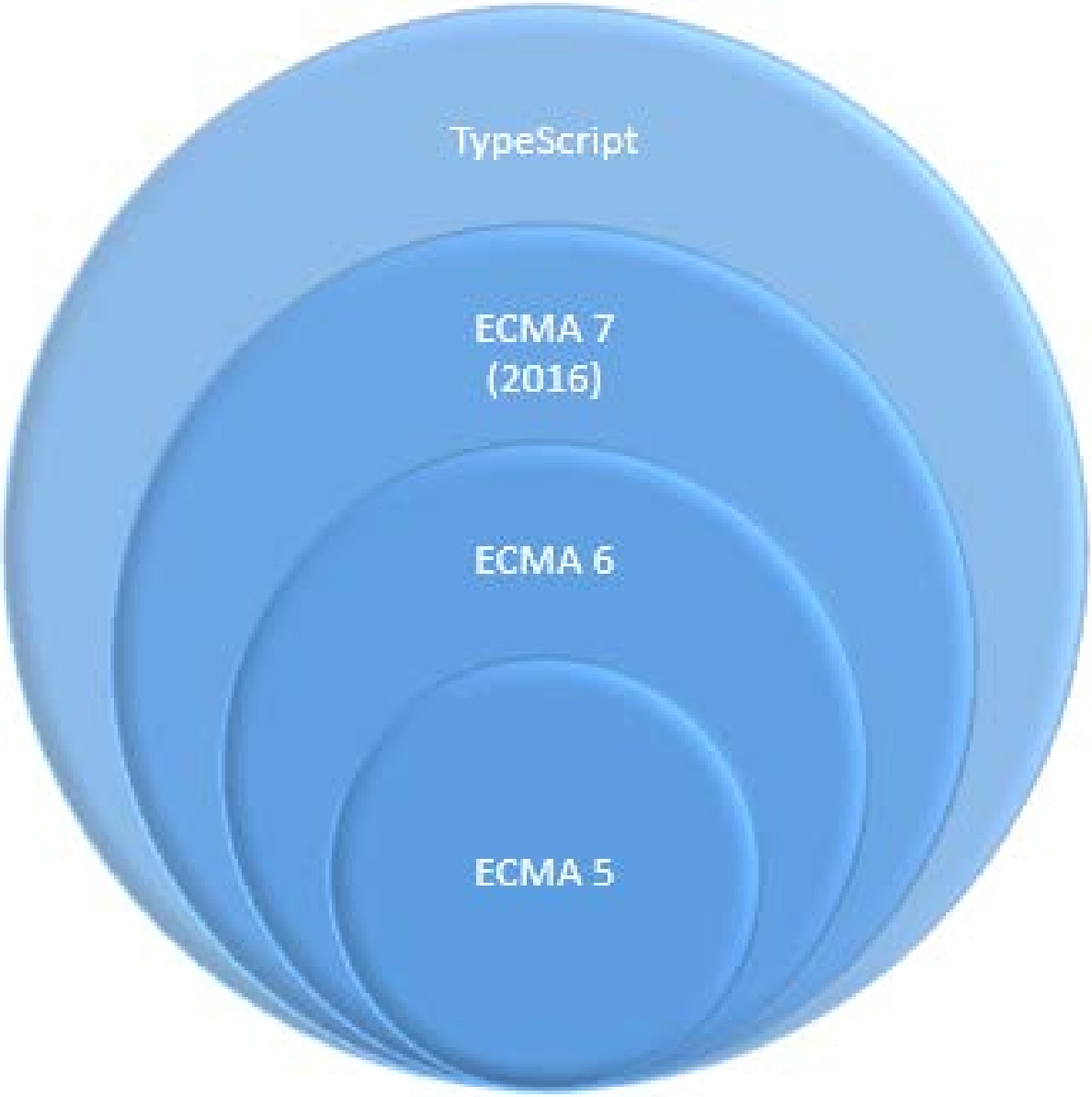
TypeScript

TypeScript is a free and open-source programming language developed and maintained by Microsoft. It is a strict superset of JavaScript, and adds optional static typing.

TypeScript is designed for development of large applications and transcompiles to JavaScript.

As TypeScript is a superset of JavaScript, any existing JavaScript programs are also valid TypeScript programs.

***Anders Hejlsberg**, lead architect of C#*
creator of Delphi and Turbo Pascal,
author of TypeScript


TypeScript
ECMA 7
(2016)
ECMA 6
ECMA 5

TypeScript: example

```
class Person {
    firstName: string;
    middleName: string;
    lastName: string;

    constructor(firstName: string, middleName: string, lastName:
string) {
        this.firstName = firstName;
        this.middleName = middleName;
        this.lastName = lastName;
    }

    getFullName() {
        let fullName = this.firstName + ' ' + this.middleName +
                ' ' + this.lastName;
        return fullName;
    }
}
```

TypeScript: example

```
class Employee extends Person {
    employeeId: string;
    salary: number;

    constructor(firstName: string, middleName: string, lastName: string,
                employeeId: string, salary: number)
    {
        super(firstName, middleName, lastName)
        this.employeeId = employeeId;
        this.salary = salary;
    }

    getSalary() {
        return this.salary;
    }
}
```

TypeScript: implementing class

```
interface BonusCalculator {
    getBonus()
}
class Employee extends Person implements BonusCalculator {
    employeeId: string;
    salary: number;
    constructor(firstName: string, middleName: string,
                lastName: string, employeeId: string, salary: number) {
        super(firstName, middleName, lastName)
        this.employeeId = employeeId;
        this.salary = salary;
    }
    getSalary() {
        return this.salary;
    }
    getBonus() {
        return this.salary * 0.3;
    }
}
```

TypeScript

Basics

LUXOFT

LXFT LISTED NYSE

Data types

```
Boolean isDone: boolean = false;

Number: height: number = 6;

String: name: string = "bob";

Array:  list:number[] = [1, 2, 3];
        list:Array<number> = [1, 2, 3];

Enum:   enum Color {Red, Green, Blue};
        c: Color = Color.Green;

Any:    notSure: any = 4;
        notSure = "maybe a string instead";
        notSure = false; // okay, definitely a boolean
        var list:any[] = [1, true, "free"];

Void:   function warnUser(): void {
            alert("This is my warning message");
        }
```

let operator

```
function func() {
    if (true) {
        let tmp: number = 123;
    }
    console.log(tmp); // ReferenceError: tmp is not defined
}


function func() {
    if (true) {
        var tmp = 123;
    }
    console.log(tmp); // 123
}
```

let operator

```
function func() {
    let foo: number = 5;
    if (…) {
        let foo:number = 10; // shadows outer `foo`
        console.log(foo); // 10
    }
    console.log(foo); // 5
}
```

const

```
let foo = 'abc';
foo = 'def';
console.log(foo); // def

const foo2 = 'abc';
foo2 = 'def'; // TypeError
```

Arrow function

```
f = v => v + 1;
```

```
var f = function (v) { return v + 1; })
```

Usage example:

```
var arr = [1,2,3];
arr.forEach(i=>console.log(i));
```

Arrow function with multiple parameters

```
f = (x,y) => x+y;
f(1,2) === 3;
```

Arrow function with function body

```
f = (x,y) => {
  console.log(x,y);
  return x+y;
}
```

Property Shorthand

```
obj = { x, y }
```

same as `obj = { x: x, y: y };`

Computed Property Names

```
obj = {
  foo: "bar",
  [ "prop_" + foo() ]: 42
}
```

```
obj = { foo: "bar" };
obj[ "prop_" + foo() ] = 42;
```

Method Properties

```
obj = {
  foo (a, b) { ... },
  bar (x, y) { ... },
  *quux (x, y) { ... }
}
```

```
obj = {
  foo: function (a, b) { ... },
  bar: function (x, y) { ... },
  // quux: no equivalent in ES5 ...
};
```

Array matching

```
var list = [ 1, 2, 3 ]
var [ a, , b ] = list
[ b, a ] = [ a, b ]
```

```
var list = [ 1, 2, 3 ];
var a = list[0], b = list[2];
var tmp = a; a = b; b = tmp;
```

Object matching

```
var { op, lhs, rhs } =
getASTNode()
```

```
var tmp = getASTNode();
var op = tmp.op;
var lhs = tmp.lhs;
var rhs = tmp.rhs;
```

Fail-soft matching

```
var list = [ 7, 42 ]
var [ a = 1, b = 2, c = 3, d ] = list
// a === 7 b === 42
// c === 3 d === undefined
```

```
var list = [ 7, 42 ];
var a = list[0] !== undefined ? list[0] : 1;
var b = list[1] !== undefined ? list[1] : 2;
var c = list[2] !== undefined ? list[2] : 3;
var d = list[3] !== undefined ? list[3]
      : undefined;
```

Array: new functions

```
[ 1, 3, 4, 2 ].find(x => x > 3) // 4
```

```
[ 1, 3, 4, 2 ].filter(function (x) {
  return x > 3; })[0]; // 4
```

Object assigning

```
var dst = { quux: 0 }
var src1 = { foo: 1, bar: 2 }
var src2 = { foo: 3, baz: 4 }
Object.assign(dst, src1, src2)

dst.quux === 0
dst.foo === 3
dst.bar === 2
dst.baz === 4
```

```
var dst = { quux: 0 };
var src1 = { foo: 1, bar: 2 };
var src2 = { foo: 3, baz: 4 };
Object.keys(src1).forEach(function(k) {
  dst[k] = src1[k]; });
Object.keys(src2).forEach(function(e) {
  dst[k] = src2[k]; });
```

String searching

```
"hello".startsWith("ello", 1) // true
"hello".endsWith("hell", 4) // true
"hello".includes("ell") // true
"hello".includes("ell", 1) // true
"hello".includes("ell", 2) // false
```

```
"hello".indexOf("ello") === 1; // true
"hello".indexOf("hell") === (4 - "hell".length);
"hello".indexOf("ell") !== -1; // true
"hello".indexOf("ell", 1) !== -1; // true
"hello".indexOf("ell", 2) !== -1; // false
```

Set

```
let s = new Set()
s.add("hello").add("goodbye").add("hello")
s.size === 2
s.has("hello") === true
for (let key of s.values()) // insertion order console.log(key)
```

Map

```
let m = new Map()
m.set("hello", 42)
m.set(s, 34)
m.get(s) === 34
m.size === 2
for (let [ key, val ] of m.entries()) console.log(key + " = " + val)
```

WeakSet/WeakMap

```
var weakSet = new WeakSet()
a = {}; // only objects allowed
weakSet.add(a);
weakSet.has(a); // true
a = null; // now a can be garbage collected
for (e in weakSet) console.log(e); // not working: WeakSet is not itarable
```

String Interpolation

```
var customer = { name: "Foo" }
var card = { amount: 7,
  product: "Bar",
  unitprice: 42 }
message = `Hello ${customer.name},
want to buy ${card.amount}
${card.product} for a total of
${card.amount * card.unitprice}
bucks?`
```

```
var customer = { name: "Foo" };
var card = { amount: 7,
    product: "Bar",
    unitprice: 42 };
message = "Hello " + customer.name + ",\n" +
"want to buy " + card.amount + " " +
card.product + " for\n" + "a total of " +
(card.amount * card.unitprice) + " bucks?";
```

New number functions

```
Number.isNaN(42) === false
Number.isNaN(NaN) === true

Number.isFinite(Infinity) === false
Number.isFinite(-Infinity) === false
Number.isFinite(NaN) === false
Number.isFinite(123) === true

Number.isSafeInteger(42) === true
Number.isSafeInteger(900719925474099 2) === false

console.log(0.1 + 0.2 === 0.3) // false
console.log(Math.abs((0.1 + 0.2) - 0.3) < Number.EPSILON)
// true
```

Default Parameter Values

```
function f (x, y = 7, z = 42) {
  return x + y + z
}
f(1) === 50
```

```
function f (x, y, z) {
  if (y === undefined) y = 7;
  if (z === undefined) z = 42;
  return x + y + z;
}
f(1) === 50;
```

Rest Parameters

```
function f (x, y, ...a) {
  return (x + y) * a.length
}

f(1, 2, "hello", true, 7) === 9
```

```
function f (x, y) {
  return (x + y) * (a.length-2);
}

f(1, 2, "hello", true, 7) === 9;
```

Spread Operator

```
var params = [ "hello", true, 7 ]
var other = [ 1, 2, ...params ] // [ 1, 2, "hello", true, 7 ]
f(1, 2, ...params) === 9
```

Trailing commas in function parameters and arrays/objects

```
let obj = {
 first: 'Jane',
 last: 'Doe',
};

let arr = [
        'red',
        'green',
        'blue',
    ];

console.log(arr.length); // 3
```

```
function foo(
        param1,
        param2,
) {}

foo(
    'abc',
    'def',
);
```

Using this in callbacks

```
arr = [1,2,3];
arr.summarize = function() {
  this.sum = 0;
  this.forEach(function(e) { this.sum = this.sum+e; } );
  // Callbacks are executed in their own context, this points to function, not arr
}
```

workaround:

```
arr.summarize = function() {
  this.sum = 0;
  var self = this;
  this.forEach(function(e) { self.sum = self.sum+e; } );
}
```

another workaround:

```
  this.forEach(function(e) { this.sum = this.sum+e; }.bind(this) );
```

lexical scoping “this”

```
arr.summarize = function() {
   this.sum = 0;
   this.forEach(e=>{ this.sum = this.sum+e; });
}
```

Static members

```
class Circle extends Shape {
    static defaultCircle () {
        return new Circle("default", 0, 0, 100)
    }
}
var defRectangle = Rectangle.defaultRectangle()
var defCircle    = Circle.defaultCircle()
```

Getters/setters

```
class Rectangle {
    constructor (width, height) {
        this._width  = width
        this._height = height
    }
    set width  (width)  { this._width = width             }
    get width  ()       { return this._width              }
    set height (height) { this._height = height           }
    get height ()       { return this._height             }
    get area   ()       { return this._width * this._height }
}
var r = new Rectangle(50, 20)
r.area === 1000
```

Modules import/export

```
//lib/math.js
export function sum (x, y) { return x + y }
export var pi = 3.141593

//  someApp.js
import * as math from "lib/math"
console.log("2π = " + math.sum(math.pi, math.pi))

//  otherApp.js
import { sum, pi } from "lib/math"
console.log("2π = " + sum(pi, pi))
```

Marking a value as the default exported value

```
//  lib/mathplusplus.js
export * from "lib/math"
export var e = 2.71828182846
export default (x) => Math.exp(x)

//  someApp.js
import exp, { pi, e } from "lib/mathplusplus"
console.log("e^{π} = " + exp(pi))
```

TypeScript

GENERATORS

LUXOFT

LXFT LISTED NYSE

Generators

```
function* range (start, end, step) {
    while (start < end) {
        yield start
        start += step
    }
}

for (let i of range(0, 10, 2)) {
    console.log(i) // 0, 2, 4, 6, 8
}

function* genFunc() {
        yield 'a';
        yield 'b';
        return 1;
    }
genObj = genFunc();
genObj.next() // {value: "a", done: false}
genObj.next() // {value: "b", done: false}
genObj.next() //  {value: 1, done: true}
arr = [...genFunc()]; // ['a', 'b']
```

Generators: example of use

```
function* objectEntries(obj) {
    // In ES6, you can use strings
    // or symbols as property keys,
    // Reflect.ownKeys() retrieves both
    let propKeys = Reflect.ownKeys(obj);

    for (let propKey of propKeys) {
        yield [propKey, obj[propKey]];
    }
}

let jane = { first: 'Jane', last: 'Doe' };
for (let [key,value] of objectEntries(jane)) {
    console.log(`${key}: ${value}`);
}
// Output:
// first: Jane
// last: Doe
```

Generators: recursion

```
function* foo() {
    yield 'a';
    yield 'b';
}
function* bar() {
    yield 'x';
    yield* foo();
    yield 'y';
}

// Collect all values yielded by bar() in an
array
let arr = [...bar()];
// ['x', 'a', 'b', 'y']
```

Generators: yielding arrays

```
function* bla() {
    yield 'sequence';
    yield* ['of', 'yielded'];
    yield 'values';
}

let arr = [...bla()];
// ['sequence', 'of', 'yielded', 'values']
```

Iterator with generator function

```
class IterableArguments {
    args:Array<any>;

    constructor(...args) {
        this.args = args;
    }

    *[Symbol.iterator]() {
        for (let arg of this.args) {
            yield arg;
        }
    }
}

let iterable = new IterableArguments("hello","world")
let arr = [...iterable]; // ["hello","world"]
for (a of iterable) console.log(a);
iterable.next()
```

TypeScript

Promises

Async/Await

LUXOFT

LXFT LISTED NYSE

Asynchronous function with callbacks

```
function add(x,y,f) {
    setTimeout(()=>f(x+y), 1000);
}

add(1,2,
    res=>add(res,3,
        res=>console.log(res)));
```

Promises

```
function add(x,y) {
    return new Promise<any>(function(resolve,reject) {
        setTimeout(()=>x>0?resolve(x+y):reject("x should be >0"),
1000);
    });
}


add(1,2)
    .then(x=>add(-5,x))
    .then(res=>console.log(`result = ${res}`))
    .catch(err=>console.log("ERROR:"+err))
```

Promise.all

```
Promise.all([add(1,2),add(2,3),add(5,5)])
    .then(res=>console.log(res))
```

> [3, 5, 10] (in a second)

async/await

```
function add(x,y) {
   return new Promise(function(resolve,reject) {
      setTimeout(()=>x>0?resolve(x+y):reject("x should be >0"), 1000);
   });
}

async function main() {
    var res = await add(1, 2);
    var res2 = await add (res, 3);
    console.log( res2 ); //6
}

main();
```

Example: fetchJson with async/await

```
async function fetchJson(url) {
    try {
        let request = await fetch(url);
        let text = await request.text();
        return JSON.parse(text);
    }
    catch (error) {
        console.log(`ERROR: ${error.stack}`);
    }
}
```

async declaration:

- Async function declarations: async function foo() {}
- Async function expressions: const foo = async function () {};
- Async method definitions: let obj = { async foo() {} }
- Async arrow functions: const foo = async () => {};

Async/await

```
// printDelayed is a 'Promise<void>'
async function printDelayed(elements: string[]) {
    for (const element of elements) {
        await delay(200);
        console.log(element);
    }
}

async function delay(milliseconds: number) {
    return new Promise<void>(resolve => {
        setTimeout(resolve, milliseconds);
    });
}

printDelayed(["Hello", "beautiful", "asynchronous", "world"]).then(() => {
    console.log();
    console.log("Printed every element!");
});
```

TypeScript

Decorators

LUXOFT

LXFT LISTED NYSE

Decorators: @readonly

```
function readonly(target, key, descriptor) {
    descriptor.writable = false;
    return descriptor;
}
class Meal {
    @readonly
    entree='salad';
}
// this is the same as
Object.defineProperty(Meal.prototype, 'entree',

    // this is descriptor:
    { value: 'salad',  enumerable: false, configurable: true,  writable: false  });


// let's check it!
var dinner = new Meal();
dinner.entree = 'soup'; // Cannot assign to read only property
```

Decorators: enrich class

```
function superhero(target) {
    target.isSuperhero = true;
    target.power = "flight";
}

@superhero
class MySuperHero {}
console.log(MySuperHero.isSuperhero); // true
```

Decorators: enrich class with parameter

```
function superhero(isSuperhero) {
   return function (target) {
      target.isSuperhero = isSuperhero
   }
}

@superhero(true)
class MySuperheroClass { }
console.log(MySuperheroClass.isSuperhero); // true

@superhero(false)
class MySuperheroClass { }
console.log(MySuperheroClass.isSuperhero); // false
```

Decorators: enrich class objects

```
@makesPhonecalls
class Cellphone {
    constructor() {
        this.model = "Samsung"
        this.storage = 16
    }
}
function makesPhonecalls(target) {
    let callNumber = function(number) {
        return `calling ${number}`
    }
    // Attach it to the prototype
    target.prototype.callNumber = callNumber
}
```

Decorators: limit access

```
function adminOnly(user) {
    return function (target) {
        if (!user.isAdmin) {
            log('You do not have sufficient privileges!');
            return false;
        }
    }
}
@adminOnly(currentUser)
function deleteAllUsers() {
    users.delete().then((response) => {
        log('You deleted everyone!');
    });
}
```

Decorator as a wrapper

```
@logger
function logMe() {
    console.log('I want to be logged');
}
// Decorator function for logging
function logger(target, name, descriptor) {
    // obtain the original function
    let fn = descriptor.value;
    // create a new function that wraps the original function
    let newFn  = function() {
        console.log('starting %s', name);
        fn.apply(target, arguments);
        console.log('ending %s', name);
    };
    // we then overwrite the origin descriptor value and return new
    descriptor.value = newFn;
    return descriptor;
}
```

Decorator as a wrapper – customization with parameters

```
@logger('custom message starting %s', 'custom message ending %s')
function logMe() {
   console.log('I want to be logged');
}
function logger(startMsg, endMsg) {
   return function(target, name, descriptor) {
      let fn = descriptor.value;
      let newFn  = function() {
         console.log(startMsg, name);
         fn.apply(target, arguments);
         console.log(endMsg, name);
      };
      descriptor.value = newFn;
      return descriptor;
   }
}
```

TypeScript

Types

LUXOFT

LXFT LISTED NYSE

Data types

```
Boolean isDone: boolean = false;

Number: height: number = 6;

String: name: string = "bob";

Array:  list:number[] = [1, 2, 3];
        list:Array<number> = [1, 2, 3];

Enum:   enum Color {Red, Green, Blue};
        c: Color = Color.Green;

Any:    notSure: any = 4;
        notSure = "maybe a string instead";
        notSure = false; // okay, definitely a boolean
        var list:any[] = [1, true, "free"];

Void:   function warnUser(): void {
            alert("This is my warning message");
        }
```

Tuples

```
// Declare a tuple type
let x: [string, number];
x = ["hello", 10]; // OK
x = [10, "hello"]; // Error

console.log(x[0].substr(1)); // OK
console.log(x[1].substr(1)); // Error, 'number' does not have 'substr'

// Return tuple from function
function f(): [string, number] {
   return [“cow”,3];
}
```

Type never

```
// Function returning never must have unreachable end point
function error(message: string): never {
    throw new Error(message);
}

// Inferred return type is never
function fail() {
    return error("Something failed");
}

// Function returning never must have unreachable end point
function infiniteLoop(): never {
    while (true) {
    }
}
```

Types: examples

Number

```
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
```

String

```
let fullName: string = `Bob Bobbington`;
let age: number = 37;
let sentence: string = `Hello, my name is ${
fullName }.
I'll be ${ age + 1 } years old next month.`
```

Array

```
let list: number[] = [1, 2, 3];
let list: Array<number> = [1, 2, 3];
```

Types: examples

Tuple

```
// Declare a tuple type
let x: [string, number];
// Initialize it
x = ["hello", 10]; // OK
// Initialize it incorrectly
x = [10, "hello"]; // Error
```

Enum

```
enum Color {Red, Green, Blue};
let c: Color = Color.Green;

enum Color {Red = 1, Green, Blue};
let colorName: string = Color[2];
```

Types: examples

Any

```
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a boolean
```

Void

```
function warnUser(): void {
    alert("This is my warning message");
}
```

Null and undefined

```
// Not much else we can assign to these variables!
let u: undefined = undefined;
let n: null = null;
```

Never

```
function error(message: string): never {
    throw new Error(message);
}
```

Type assertions

A type assertion is like a type cast in other languages, but performs no special checking or restructuring of data.

```
let someValue: any = "this is a string";

let strLength: number = (<string>someValue).length;
```

And the other is the as-syntax:

```
let someValue: any = "this is a string";

let strLength: number = (someValue as string).length;
```

Type aliases

```
type PrimitiveArray = Array<string|number|boolean>;
type MyNumber = number;
type Callback = () => void;

let f: Callback;
f = function() {
    console.log(“function");
}
```

TypeScript

Interfaces

LUXOFT

LXFT LISTED NYSE

Interfaces

Define right in place:

```
function printLabel(labelledObj: {label: string}) {
    console.log(labelledObj.label);
}

var myObj = {size: 10, label: "Size 10 Object"};
printLabel(myObj);
```

Using interface keyword:

```
interface LabelledValue {
    label: string;
}
function printLabel(labelledObj: LabelledValue) {
    console.log(labelledObj.label);
}
var myObj = {size: 10, label: "Size 10 Object"};
printLabel(myObj);
```

Interfaces: optional properties

```
interface SquareConfig {
    color?: string;
    width?: number;
}

function createSquare(config: SquareConfig):
    {color: string; area: number} {
    var newSquare = {color: "white", area: 100};
    if (config.color) {
        newSquare.color = config.color;
        // Type-checker can catch the mistyped name
here
    }
    if (config.width) {
        newSquare.area = config.width * config.width;
    }
    return newSquare;
}

var mySquare = createSquare({color: "black"});
```

Interfaces: function types

```
interface SearchFunc {
    (source: string, subString: string): boolean;
}

var mySearch: SearchFunc;
mySearch = function(source: string, subStr: string) {
    var result = source.search(subStr);
    if (result == -1) {
        return false;
    } else {
        return true;
    }
}
```

Interfaces: array types

```
interface StringArray {
        [index: number]: string;
}
var myArray: StringArray;
myArray = ["Bob", "Fred"];
```

Interfaces: class types

```
interface ClockInterface {
    currentTime: Date;
    setTime(d: Date);
}

class Clock implements ClockInterface {
    currentTime: Date;

    setTime(d: Date) {
        this.currentTime = d;
    }

    constructor(h: number, m: number) { }
}
```

Interfaces: static/instance side of class

```
interface ClockStatic {
    new (hour: number, minute: number);
}

class Clock {
    currentTime: Date;

    constructor(h: number, m: number) { }
}

var cs: ClockStatic = Clock;
var newClock = new cs(7, 30);

class Timer {
    constructor(h: number, m: number) { }
}
cs = Timer;
var newTimer = new cs(7, 30);
```

Extending Interfaces

```
interface Shape {
    color: string;
}

interface PenStroke {
    penWidth: number;
}

interface Square extends Shape, PenStroke {
    sideLength: number;
}

var square = <Square>{};
square.color = "blue";
square.sideLength = 10;
square.penWidth = 5.0;
```

Interfaces: Hybrid Types

```
interface Counter {
    (start: number): string;
    interval: number;
    reset(): void;
}

var c: Counter;
c(10);
c.reset();
c.interval = 5.0;
```

TypeScript

Classes

LUXOFT

LXFT LISTED NYSE

Classes

```
class Greeter {
    greeting: string;

    constructor(message: string) {
        this.greeting = message;
    }

    greet() {
        return "Hello, " + this.greeting;
    }
}

var greeter = new Greeter("world");
```

Private/Public/Protected: Public by default

```
class Animal {
    private name:string;

    constructor(theName: string) {
        this.name = theName;
    }

    move(meters: number) {
        alert(this.name + " moved " + meters + "m.");
    }
}
```

Parameter properties:

```
class Animal {
    constructor(private name: string) { }
    move(meters: number) {
        alert(this.name + " moved " + meters + "m.");
    }
}
```

Accessors

```
var passcode = "secret passcode";

class Employee {
    private _fullName: string;

    get fullName(): string { return this._fullName; }
    set fullName(newName: string) {
        if (passcode && passcode == "secret passcode") {
            this._fullName = newName;
        } else {
            alert("Error: Unauthorized update!");
        }
    }
}

var employee = new Employee();
employee.fullName = "Bob Smith";
if (employee.fullName) {
    alert(employee.fullName);
}
```

Static properties

```
class Grid {
    static origin = {x: 0, y: 0};

    calculateDistanceFromOrigin(point: {x: number; y: number;}) {
        var xDist = (point.x - Grid.origin.x);
        var yDist = (point.y - Grid.origin.y);
        return Math.sqrt(xDist * xDist + yDist * yDist) / this.scale;
    }

    constructor (public scale: number) { }
}

var grid1 = new Grid(1.0); // 1x scale
var grid2 = new Grid(5.0); // 5x scale
alert(grid1.calculateDistanceFromOrigin({x: 10, y: 10}));
alert(grid2.calculateDistanceFromOrigin({x: 10, y: 10}));
```

Constructor function

```
class Greeter {
    static standardGreeting = "Hello, there";
    greeting: string;

    greet() {
        if (this.greeting) { return "Hello, " + this.greeting; }
        else { return Greeter.standardGreeting; }
    }
}

var greeter1: Greeter;
greeter1 = new Greeter();
alert(greeter1.greet());

var greeterMaker: typeof Greeter = Greeter;
greeterMaker.standardGreeting = "Hey there!";
var greeter2:Greeter = new greeterMaker();
alert(greeter2.greet());
```

Using a class as an interface

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

interface Point3d extends Point {
        z: number;
}

var point3d: Point3d = {x: 1, y: 2, z: 3};
```

TypeScript

Functions

LUXOFT

LXFT LISTED NYSE

Functions

```
function add(x: number, y: number): number { return x+y; }

var myAdd = function(x: number, y: number): number { return x+y; };
```

Writing the function type:

```
var myAdd: (a:number, b:number)=>number =
    function(x: number, y: number): number { return x+y; };
```

Inferring the types:

```
// The parameters 'x' and 'y' have the type number
var myAdd: (baseValue:number, increment:number)=>
    number = function(x, y) { return x+y; };
```

Functions

Optional parameters:

```
function buildName(firstName: string, lastName?: string) {
    if (lastName) return firstName + " " + lastName;
    else return firstName;
}

var result1 = buildName("Bob"); //works correctly now
var result2 = buildName("Bob", "Adams", "Sr."); //error, too many params
var result3 = buildName("Bob", "Adams"); //ah, just right
```

Default parameters:

```
function buildName(firstName: string, lastName = "Smith") {
    return firstName + " " + lastName;
}

var result1 = buildName("Bob"); //works correctly now, also
var result2 = buildName("Bob", "Adams", "Sr."); //error, too many
params
var result3 = buildName("Bob", "Adams"); //ah, just right
```

Functions

Rest parameters:

```
function buildName(firstName: string, ...restOfName: string[]) {
    return firstName + " " + restOfName.join(" ");
}
var employeeName = buildName("Joseph", "Samuel", "Lucas", "MacKinzie");
```

Functions overloading

```
var suits = ["hearts", "spades", "clubs", "diamonds"];

function pickCard(x: {suit: string; card: number; }[]): number;
function pickCard(x: number): {suit: string; card: number; };
function pickCard(x): any {
    // Check to see if we're working with an object/array
    if (typeof x == "object") {
        var pickedCard = Math.floor(Math.random() * x.length);
        return pickedCard;
    } // Otherwise just let them pick the card
    else if (typeof x == "number") {
        var pickedSuit = Math.floor(x / 13);
        return { suit: suits[pickedSuit], card: x % 13 };
    }
}

var myDeck = [{ suit: "diamonds", card: 2 }, { suit: "spades", card: 10 }];
var pickedCard1 = myDeck[pickCard(myDeck)];
alert("card: " + pickedCard1.card + " of " + pickedCard1.suit);
var pickedCard2 = pickCard(15);
alert("card: " + pickedCard2.card + " of " + pickedCard2.suit);
```

TypeScript

Generics

LUXOFT

LXFT LISTED NYSE

Generics

```
function identity(arg: number): number { return arg; }

function identity(arg: any): any { return arg; }
```

Using generics:

```
function identity<T>(arg: T): T { return arg; }
```

Pass type in <>:

```
var output = identity<string>("myString"); // type of output will be 'string‘
```

Interfere type automatically:

```
var output = identity("myString"); // type of output will be 'string'
```

Generics

```
function loggingIdentity<T>(arg: T): T {
    console.log(arg.length); // Error: T doesn't have .length
    return arg;
}
```

We can define that we are using array:

```
function loggingIdentity<T>(arg: T[]): T[] {
    console.log(arg.length); // Array has a .length, so no more error
    return arg;
}
```

Alternatively:

```
function loggingIdentity<T>(arg: Array<T>): Array<T> {
    console.log(arg.length);
    // Array has a .length, so no more error
    return arg;
}
```

Generic types:

```
function identity<T>(arg: T): T {
    return arg;
}

var myIdentity: <T>(arg: T)=>T = identity;
```

Generic Classes

```
class GenericNumber<T> {
        zeroValue: T;
        add: (x: T, y: T) => T;
}

var myGenericNumber = new GenericNumber<number>();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function(x, y) { return x + y; };



var stringNumeric = new GenericNumber<string>();
stringNumeric.zeroValue = "";
stringNumeric.add = function(x, y) { return x + y; };

alert(stringNumeric.add(stringNumeric.zeroValue, "test"));
```

Generic constraints

```
function loggingIdentity<T>(arg: T): T {
    console.log(arg.length); // Error: T doesn't have .length
    return arg;
}
```

Solution using constraint:

```
interface Lengthwise {
    length: number;
}

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

loggingIdentity(3); // Error, number doesn't have a .length property
loggingIdentity({length: 10, value: 3}); // OK
```

Using class type in generics

```
function create<T>(c: {new(): T; }): T {
    return new c();
}
```

Example of using:

```
class BeeKeeper { hasMask: boolean; }
class ZooKeeper { nametag: string; }
class Animal { numLegs: number; }
class Bee extends Animal { keeper: BeeKeeper; }
class Lion extends Animal { keeper: ZooKeeper; }

function findKeeper<A extends Animal, K> (a: {new(): A;
    prototype: {keeper: K}}): K {
        return a.prototype.keeper;
}

findKeeper(Lion).nametag; // typechecks!
```

Merging interfaces

```
interface Box {
        height: number;
        width: number;
}

interface Box { scale: number; }

var box: Box = {height: 5, width: 6, scale: 10};
```

Type Inference

basic:

```
x = 3
```

best common type:

```
var x = [0, 1, null];
```

types share a common structure, but no one is the super type of all candidate types:

```
var zoo = [new Rhino(), new Elephant(), new Snake()];
```

to correct use :

```
var zoo: Animal[] = [new Rhino(), new Elephant(), new Snake()];
```

Contextual type

```
window.onmousedown = function(mouseEvent) {
    console.log(mouseEvent.button); //<- Error
};
```

Solution:

```
window.onmousedown = function(mouseEvent: any) {
    console.log(mouseEvent.button); //<- Now, no error is given
};
```

explicit type override the contextual type:

```
function createZoo(): Animal[] {
    return [new Rhino(), new Elephant(), new Snake()];
}
```

Type Compatibility

```
interface Named { name: string; }
class Person {
    name: string;
}
var p: Named; // OK, because of structural typing
p = new Person();
```

x is compatible with y if y has at least the same members as x:

```
interface Named {  name: string; }
var x: Named; // y's inferred type is { name: string; location: string; }
var y = { name: 'Alice', location: 'Seattle' };
x = y; // OK!
```

the same for checking function call arguments:

```
function greet(n: Named) {
    alert('Hello, ' + n.name);
}
greet(y); // OK
```

Class expressions (anonymous class type)

```
let Point = class {
    constructor(public x: number, public y: number) { }
    public length() {
        return Math.sqrt(this.x * this.x + this.y * this.y);
    }
};
var p = new Point(3, 4);  // p has anonymous class type
console.log(p.length());
```

Extending expressions

```
// Extend built-in types
class MyArray extends Array<number> { }
class MyError extends Error { }

// Extend computed base class
class ThingA {   getGreeting() { return "Hello from A"; }  }
class ThingB {   getGreeting() { return "Hello from B"; }  }
interface Greeter {   getGreeting(): string;   }
interface GreeterConstructor {    new (): Greeter;   }

function getGreeterBase(): GreeterConstructor {
    return Math.random() >= 0.5 ? ThingA : ThingB;
}
class Test extends getGreeterBase() {
    sayHello() {
        console.log(this.getGreeting());
    }
}
```

Abstract classes

```
abstract class Base {
    abstract getThing(): string;
    getOtherThing() { return 'hello'; }
}
let x = new Base(); // Error, 'Base' is abstract

// Error, must either be 'abstract' or implement concrete 'getThing'
class Derived1 extends Base { }
class Derived2 extends Base {
    getThing() { return 'hello'; }
    foo() {
        super.getThing();// Error: cannot invoke abstract members through 'super'
    }
}
var x = new Derived2(); // OK
var y: Base = new Derived2(); // Also OK
y.getThing(); // OK
y.getOtherThing(); // OK
```