TypeScript

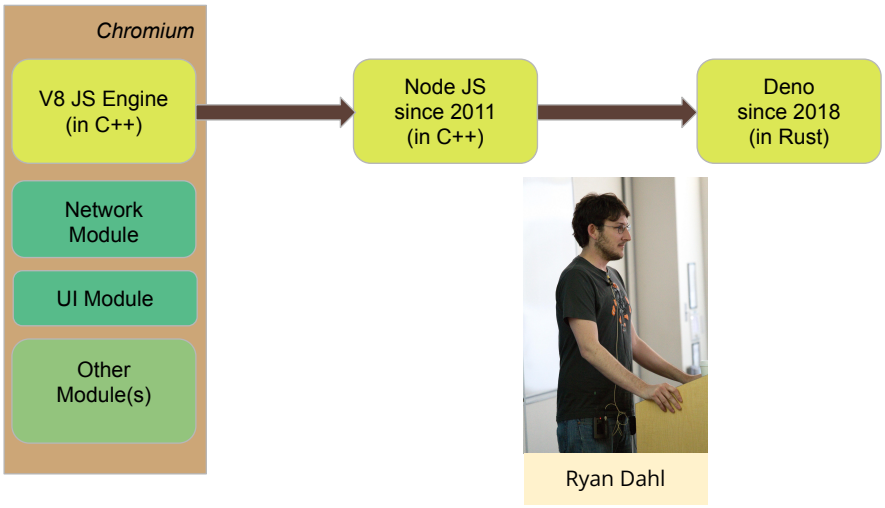
Transition from Java to TypeScript

Official Reference:
The TypeScript Handbook

JS Edition	Release Date	Code Name	TypeScript Version
1st-4th	June 1997, Jun 1998, Dec 1999		
5th	June 2011		TypeScript 0.x (2012-2013)
6th	June 2015	ECMAScript 6 or ES2015	
7th	June 2016	ECMAScript 2016	TypeScript 2.0
8th	June 2017	ECMAScript 2017	
9th	June 2018	ECMAScript 2018	TypeScript 3.0
10th	June 2019	ECMAScript 2019	
11th	June 2020	ECMAScript 2020	TypeScript 4.0

Prerequisites

- Proficient in Java
- Most concepts in TypeScript will be explained by comparisons with Java
- Software Required
  - NodeJS
    - node: for running JavaScript in a non-browser environment
    - npm (Node Package Manager): for installing JS/TS libraries
  - TypeScript
    - ts-node
    - tsc (TypeScript transpiler to JavaScript)



Initial Setup For NodeJS (Do This Once)

1. Download NodeJS (from https://nodejs.org)
   a. Pick the LTS version (v.14 or newer)
2. Verify your installation
   a. From the command prompt / console / PowerShell:

```
node -v  # version 14.x.x (or newer)
npm -v   # version 7.x.x (or newer)
npx -v   # version 7.x.x (or newer)
```

Topics

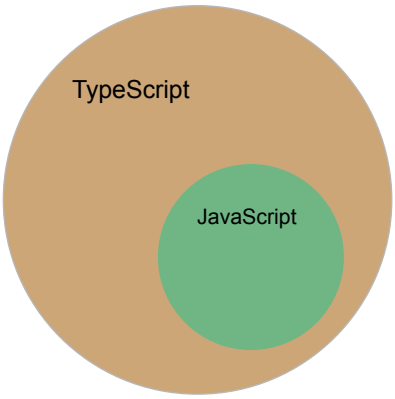
- General TypeScript (off-browser)
- TypeScript on Browser
  - Inspecting DOM trees
  - Manipulating DOM trees

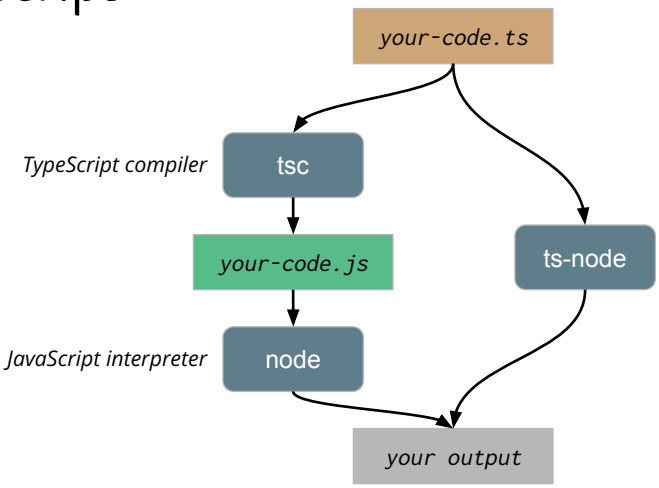
Warning

1. The “ and ‘ characters in the code snippets in this slide may not be a valid character for JS/TS transpiler
2. Copy-and-pasting the code snippet into your editor may/may not trigger compiler errors
3. Use the code snippet as a reference, but **type them manually** to try them out

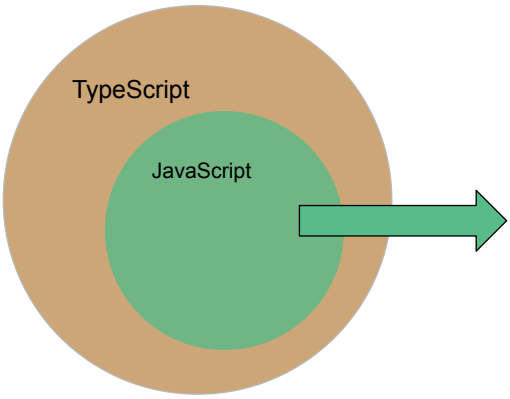
Part A: Off-Browser TypeScript

TypeScript vs. JavaScript





TypeScript vs. JavaScript: Online Reference



Use Mozilla Developer Network (MDN Web Docs) to lookup JS objects/classes:

- *Array,*
- *BigInt,*
- *Map,*
- *Number,*
- *String,*
- *Promise,*
- *etc.*

Initialize a NodeJS (and TypeScript) Project

```
# Create a new (sub) directory
mkdir my-first-node-project
cd my-first-node-project

npm init -y                    # this creates package.json
npm install -D typescript      # Add typescript as development dependencies
npm install -D ts-node
npx tsc -init                  # Creates tsconfig.json

# Create hello.ts

# Option 1: Use ts-node
npx ts-node hello.ts

# Option 2: Use tsc and node
npx tsc hello.ts
node hello.js
```

package.json

```
{
  “name”: “sample-project”,
  “version”: “1.0.0”,
  “author”: “Don Knuth <knuth@mail.org>”,
  “licence”: “MIT”,
  “dependencies”: {
    “@js-joda/core”: “3.0.1”,
    “axios”: “0.25.3”,
    . . .
  },
  “devDependencies”: {
    “ts-node”: “x.y.z”,
    “Nodemon”: “x.y.z”
  }
}
```

Libraries needed to run/deploy your app

Libraries needed only during development of your app

Demo: Project Setup

Hello World: Java vs. TypeScript

Demo.java

```
class Demo {
  public sayHello() {
    System.out.println("Hello, JS");
  }
}
```

hello.ts

```
console.log("Hello World");
console.error("Hello Again");
```

Say.java

```
class Say {
  public static void main(String[] argos) {
    System.out.println("Hello World!");
    System.err.println("Hello again");

    Demo d = new Demo();
    d.sayHello();
  }
}
```

say.ts

```
class Demo {
  sayHello(): void {
    console.log("Hello, TS");
  }
}

const d:Demo = new Demo();
d.sayHello();
```


Functions vs. Methods

function-demo.ts

```
// Stand-alone fn
function sayHello(): void {
  console.log("Hello, TS");
}
```

method-demo.ts

```
class Demo {
  // a method of a class
  sayHello(): void {
    console.log("Hello, TS");
  }
}
```

- Use `function` keyword for standalone functions
- No `function` keyword methods in a class

Data Types

Java	TypeScript
`boolean`	`boolean`
`char`	`string`
`String`	`string`
`float, double`	`number`
`short, int, long`	`number`
	`any` (no type checking)
	`unknown` (strict type checking)

Variable Declaration

Java	TypeScript
`boolean hasCoupon;`	`let hasCoupon: boolean;`
`boolean isHidden = false;`	`let isHidden = false;`
`final String month = "July";`	`const month = "July";`
`float taxRate;`	`let taxRate: number;`
`short distance;`	`let distance: number;`

- Use `let` for mutable variables
- Use `const` for immutable "variables"

Explicit type is not required when the compiler can infer the type from the surrounding context

Variable Declarations (uninitialized)

```
// Java (inside a class)
boolean isDarkMode;          // init to false
String lang;                 // init to null
float total;                 // init to 0.0f
```

```
// TypeScript (anywhere)
let isDarkMode: boolean;     // init to undefined
let lang: string;            // init to undefined
let total: number;           // init to undefined
```

`null` is different from `undefined`

TS Unions: **multiple types**

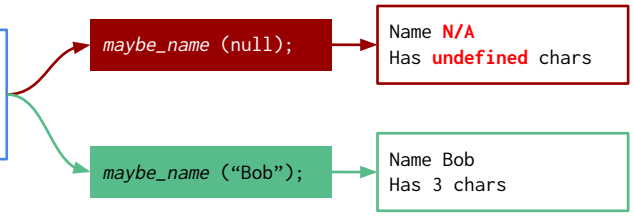
```
// TypeScript
let a: number | boolean;                  // init to undefined
let b: string | number | null = 6.5;      // current type is number


a = “can’t do this”;          // Error, can’t take a string type
a = false;                    // current type is boolean

console.log(typeof b);        // output “number”
b = “hello”;
console.log(typeof b);        // output “string”
```

Use this feature in conjunction with `typeof` test at runtime

Null: Chaining (?), Assertion (!), and Coalesce (??)

```
let noName: string | null;
noName = null;

console.log(noName?.length);    // Compile error: property “length” does not exist
console.log(noName!.length);    // Runtime CRASHED!

let airport: string | null;
airport = “GRR”;
console.log(airport?.length);  // Output 3
```

```
function maybe_name (name: string | null): void {
  console.debug(“Name”, name ?? “N/A”);
  console.debug(“Has“, name?.length, “chars”);
}
```



VSCode Demo: Multi Types and Nullables

Type Assertions (or Typecast)

```
function doOne (x: number | string | null): void {
  console.debug(x.toUpperCase());   // Compile ERROR: toUpperCase() does not exist for number
  console.debug(x * 10);            // Compile ERROR: incorrect type for arithmetic
}
```

```
function doOne (x: number | string | null): void {
  console.debug((x as string).toUpperCase());
}
```

```
doOne("file");    // Output FIVE
doOne(5);         // Runtime crash!
```

```
function doOne (x: number | string | null): void {
  console.debug((x as number) * 3);
}
```

```
doOne("file");    // Runtime crash!
doOne(5);         // Output 15
```

SmartCast

```
function doOne (x: number | string | null): void {
  if (typeof x === 'number') console.debug(x * 3);
  else if (typeof x === 'string') console.debug(x.toUpperCase());
}
```


== VS ===

5 == "5"	true
0.123 == "0.123"	true
1 == true	true
5 == true	false
0 == false	true
"0" == false	true
"1" == true	true

With internal type conversion

5 === "5"	false
0.123 === "0.123"	false
1 === true	false
5 === true	false
0 === false	false
"0" === false	false
"1" === true	false

No type conversion

Arrays

Arrays

```
// Create with initial values and capacity
const primes: number[] = [31, 43, 19];
const alsoPrimes: Array<number> = [31, 43, 19];

for (let k = 0; k < primes.length; k++) {
  console.debug("At", k, primes[k]);
}

// At 0 31
// At 1 43
// At 2 19
```

```
// Initialize with capacity
const values: number[] = new Array(5);
const nums = new Array<number>(7);

console.log(values.length);          // output 5
console.log(nums.length);            // output 7

values[2] = 73;
console.debug(values[0]);            // Output "undefined"
console.debug("OOB", values[13]);    // Output "OOB undefined"
```

Arrays: for, for-in vs. for-of

```
const fruits = [“Apple”, “Banana”, “Cherry”];
```

```
for (let k = 0; k < fruits.length; k++) {
  console.debug(“At”, k, fruits[k]);
}
```

```
for (let k in fruits) {
  console.debug(“At”, k, fruits[k]);
}
```

for-in

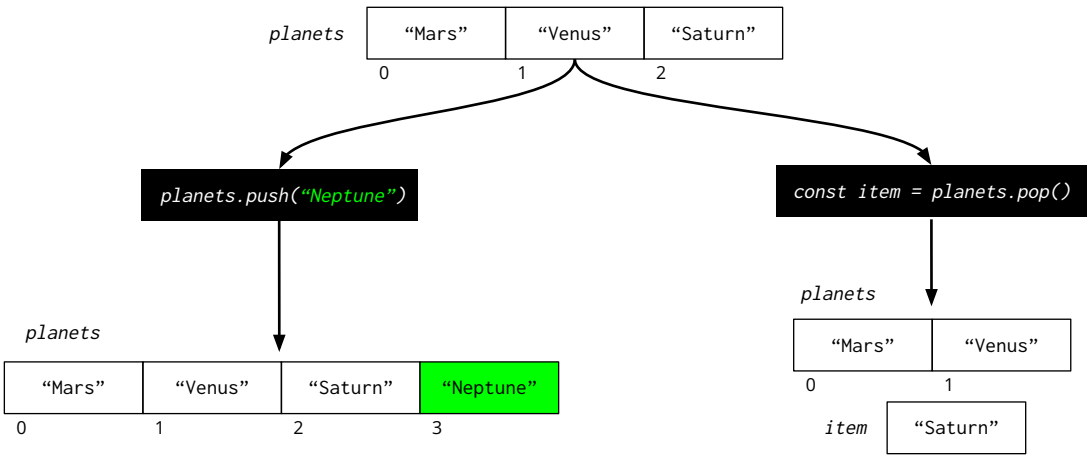
```
At 0 Apple
At 1 Banana
At 2 Cherry
```

```
for (let f of fruits) {
  console.debug(f);
}
```

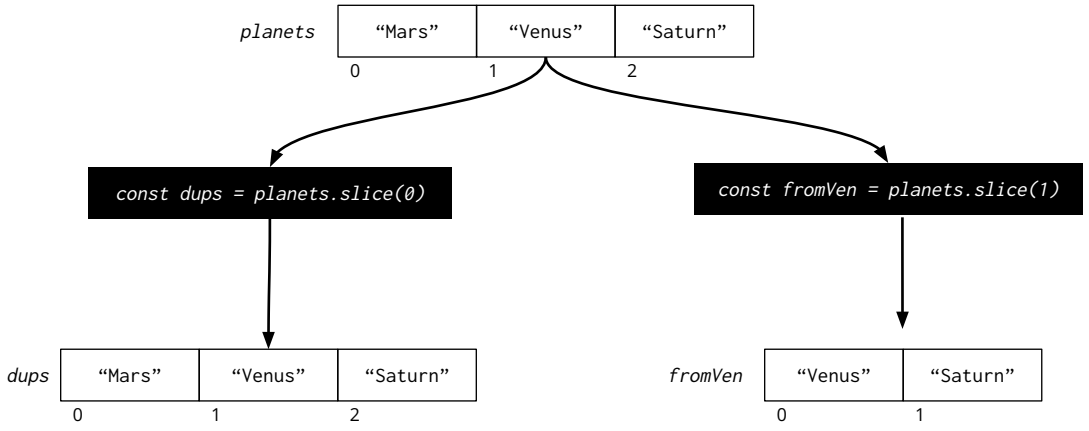
for-of

```
Apple
Banana
Cherry
```

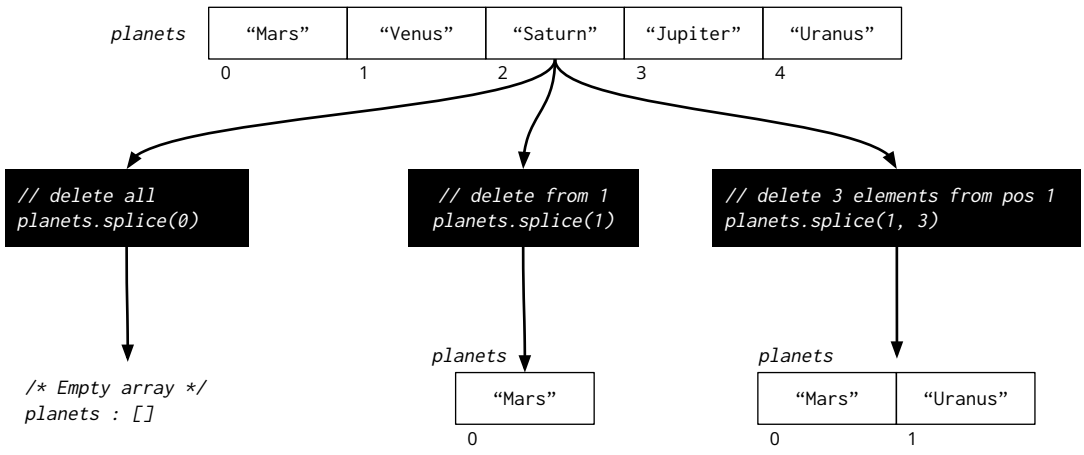

Array: .push() and .pop()



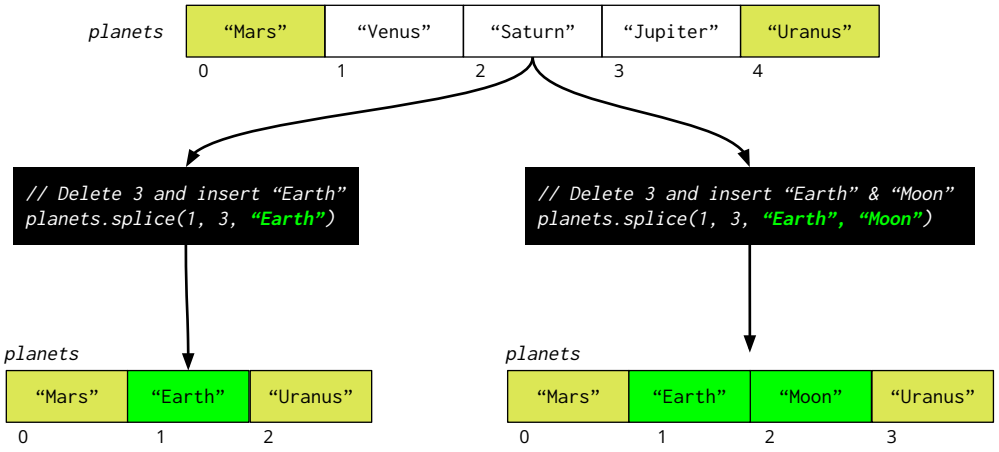
Array: `.slice()` creates a copy



Array: `.splice()` delete elements



Array: `.splice()` replaces elements



VSCode Demo: Arrays

Objects

Java Classes and Objects vs. TS Objects

```
// Java objects must be instantiated from a class
// In Sub.java
class Sub() {
  public String name;
  public int calorie;
}

// In AnotherFile.java
Sub my_order = new Sub();
my_order.name = “Spicy Turkey”;
my_order.calorie = 182;

my_order.price = 3.17;  // ERROR!
```

```
// TypeScript (no class needed)
const my_order = {
  name: “Spicy Turkey”,
  calorie: 182
}
```

Objects can be created without a class definition

Objects in TypeScript

```
// Typeless objects
const in_a_month = {
  name: “September”,
  days: 30
}

const employee_vacation = {
  name: “Bob”,
  days: 11
}
```

```
// Typed objects
type Monthly = {
  name: string,
  days: number
}

const in_a_month:Monthly = {
  name: “September”,
  days: 30
}
```

```
type VacationDays = {
  name: string,
  days: number
}

const employee_vacation:VacationDays = {
  name: “Bob”,
  days: 11
}
```


Objects with Sub-Objects & Array prop

```
type City = {
  name: string,
  population: number,
  geopos: {
    lat: number,
    lon: number
  },
  univs: Array<string>
}
```

```
const ours:City = {
  name: “Grand Rapids”,
  population: 198_400,
  geopos: {
    lat: 42.9633599,
    lon: -85.6680863
  },
  univs: [
    “Aquinas”, “Calvin”,
    “Cornerstone”, “GVSU”, “Kuiper”
  ]
}
```

```
const theirs:City = {
  name: “East Lansing”,
  population: 48_729,
  geopos: {
    lat: 42.737652,
    lon: -84.483788
  },
  univs: [
    “MSU”,
  ]
}
```

```
console.log(ours.name);
for (let u of ours.univs) console.log(u);

console.log(theirs.geopos.lat);
```

```
Grand Rapids
Aquinas
Calvin
. . .
42.737652
```

`for-in` to enumerate object properties

```
const theirs:City = {
  name: “East Lansing”,
  population: 48_729,
  geopos: {
    lat: 42.737652,
    lon: -84.483788
  },
  univs: [
    “MSU”,
  ]
}
```

```
for (let z in theirs) {
  console.debug(z)
}
```

```
name
population
geopos
univs
```

```
for (let z in theirs) {
  console.debug(z, theirs[z]);
}                  ^-------^ ERROR
```

```
const eLan = theirs as any;
for (let z in theirs) {
  console.debug(z, “==>”, eLan[z])
}
```

```
name ==> East Lansing
population ==> 48729
geopos ==> {lat: 42..., lon: -84..}
univs == > [“MSU”]
```


Array of Objects

```
// In Atom.java
class Atom {
  public String name;
  public weight double;
}

// In AnotherFile.java
ArrayList<Atom> atoms = new ArrayList<>();
Atom a = new Atom(“Carbon”, 12);
atoms.add(a);
Atom b = new Atom(“Oxygen”, 16);
atoms.add(b);
atoms.add(new Atom(“Natrium”, 23);
```

TS: option 1

```
// TypeScript (no class required, BUT)
const atoms = [];
atoms.push({ name: “Carbon”, weight: 12});
atoms.push({ name: “Oxygen”, weight: 16});
atoms.push({ name: “Natrium”, weight: 23});
```

TS: option 2

```
// Or initialize the array
const atoms = [
  { name: “Carbon”, weight: 12},
  { name: “Oxygen”, weight: 16},
  { name: “Natrium”, weight: 23}
];
```

Array of Typed Objects

Typeless array

```
const atoms = [];
atoms.push({ name: “Carbon”, weight: 12});
atoms.push({ namme: “Fluor”, weight: 12});                    // OK
atoms.push({ name: “Oxygen”});                                // OK
atoms.push({ name: “Natrium”, weight: 23, isMetal: false}); // OK
```

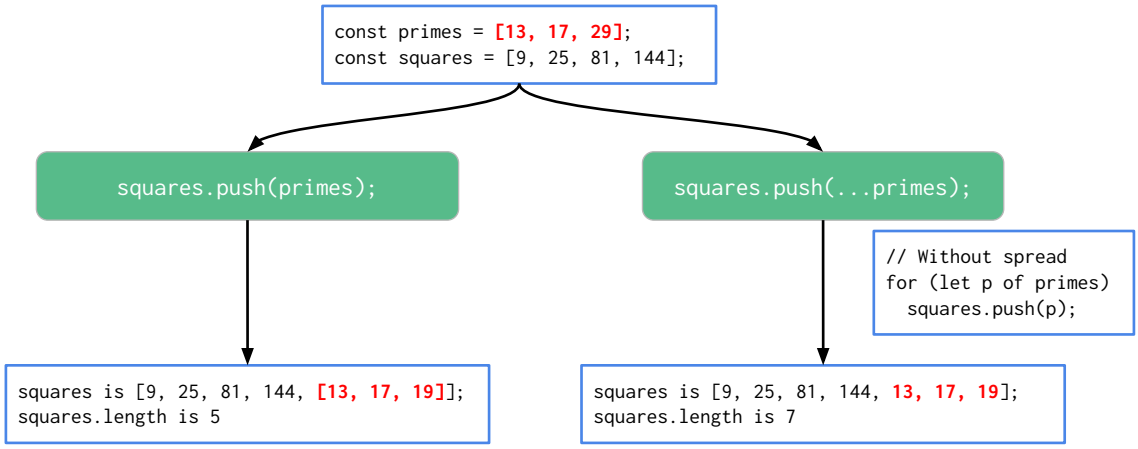
```
// Declare a type alias
type Atom = {
  name: string,
  weight: number
}
```

Typed array

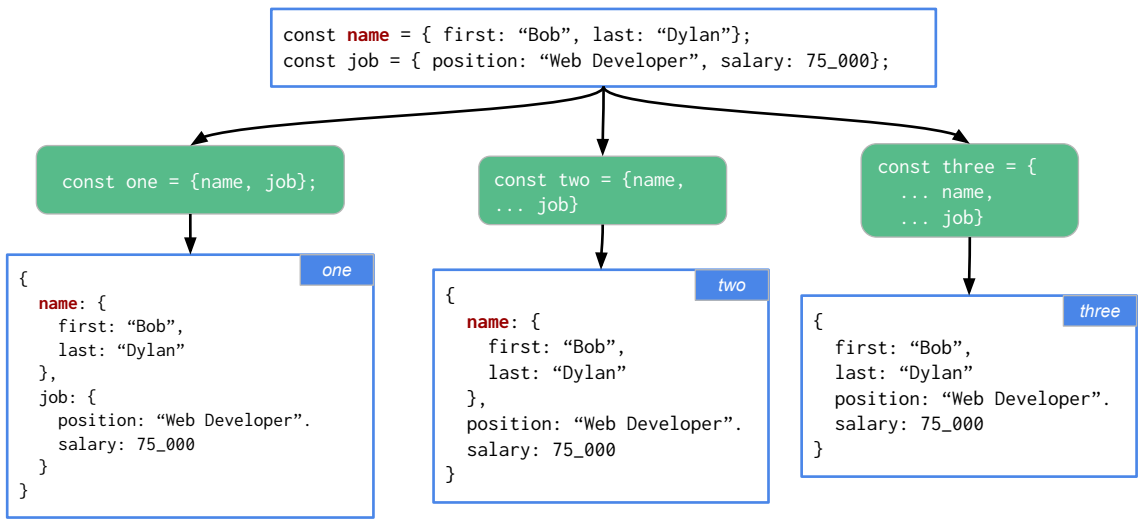
```
const atoms:Array<Atom> = [];
atoms.push({ name: “Carbon”, weight: 12});
atoms.push({ namme: “Fluor”, weight: 12}); // ERROR: “namme” does not exist
atoms.push({ name: “Oxygen”});             // ERROR: property “weight” is missing

atoms.push({
  name: “Natrium”,
  weight: 23,
  isMetal: false});                       // ERROR: “isMetal” does not exist
```


Spread Operator (Unpack)

Spreading an Array



Spreading an Object



Spread on Objects (with duplicate props)

Without spread

```
const prop1 = {name: "Carbon", abbrev: "Cb"}
const prop2 = {weight: 12, abbrev: "C"}

// without spread on prop1
const element = {prop1, ...prop2};
```

```
{
  prop1: {
    name: "Carbon", abbrev: "Cb"
  },
  weight: 12, abbrev: "C"
}
```

element

```
{
  isMetal: false,
  name: "Clue",
  abbrev: "C",
  weight: 12,
}
```

With spread

```
const prop1 = {name: "Carbon", abbrev: "Ca"}
const prop2 = {weight: 12, abbrev: "C", name: "Clue"}

// with spread
const element = {...prop1, ...prop2, isMetal: false};
const el2     = {...prop2, ...prop1, isMetal: false};
```

el2

```
{
  isMetal: false,
  name: "Carbon",
  abbrev: "Ca",
  weight: 12,
}
```

Later values overwrite previous values of the same key

Object spread: copy and modify

```
const bob = {
  first: "Bob",
  last: "Dylan"
  position: "Web Developer".
  salary: 75_000
}
```

```
const bob_now = {
  ...bob,
  workFromHome: true,
  position: "Cloud Data Egr.",
  salary: 78_000
}
```

bob_now

```
{
  first: "Bob",
  last: "Dylan",
  workFromHome: true,
  position: "Cloud Data Egr.",
  salary: 78_000
}
```

This won't work (no copy created).

```
const bob_now = bob;
bob_now.position = "Cloud Data Egr.";
bob_now.salary = 78_000;
```

Enum vs. Literal Types

```
enum CollegeYear {
  Freshman,
  Sophomore,
  Junior,
  Senior
}
```

```
let yr: CollegeYear;
yr = CollegeYear.Junior;
console.debug(yr);                    // Output 2
console.debug(CollegeYear[yr]);       // Output Junior
```

Sort order (enum order): Freshman < Sophomore < Junior < Senior

```
type CollegeLiteral =
  “Freshman” |
  “Sophomore” |
  “Junior” |
  “Senior”;
```

```
let yr: CollegeLiteral;
yr = “junior”;          // Compile error
yr = “Junior”
console.debug(yr);      // Output “Junior”
```

Sort order (alphabetical): “Freshman” < “Junior” < “Senior” < “Sophomore”

Literal Types: Narrowing

```
// TypeScript
let dayOfWeek: string;
dayOfWeek = “Munday”;   // No error

let strictDOW: “Mon” | “Tue” | “Wed” | “Thu”;
strictDOW = undefined;  // OK
strictDOW = “Fri”;      // Compile Error: invalid value

let dieValue: 1 | 2 | 3 | 4 | 5 | 6;
dieValue = undefined;   // OK
dieValue = 0;     // ERROR
```

- Use this for data with one a small number of valid values.
- Invalid values are detected at compile time (not at runtime)

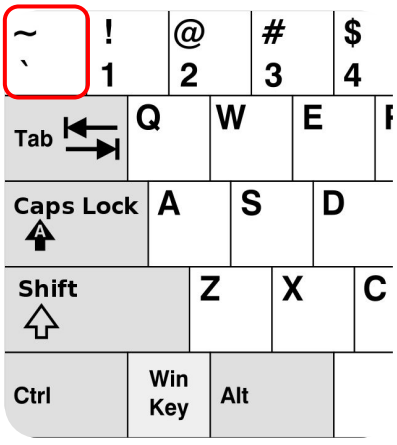
String Operations

(very similar to java.lang.String class)

String Interpolation (backquotes)

```
`Some text here ${var} and here`
`More text ${expression} also here`
```

```
const x = “Eleven”;
const arr = [3, 5, 13];

// Java-like string concatenation
let oldStore = (4 + arr[0]) + “-” + x;    // 7-Eleven

// Use backtick string interpolation
let store = `${4 + arr[0]}-${x}`;         // 7-Eleven
```



ES6 key/value Shortcut

```
let cityName = "Allendale";
let zipCode = "49401";

let location = {
  city: cityName,
  zip: zipCode;
};
```

When both key and value refer to the same name, you don't have to write them both. Only one is required

```
let city = "Allendale";
let zip = "49401";

let location = {
  city: city,
  zip: zip;
};
```

equivalent

```
let city = "Allendale";
let zip = "49401";

let location = {
  city,
  zip;
};
```


Array Destructuring

Without spread

```
const nums:number[] = [1,2,3,4,5];

const [fist,rest] = nums;
// first is 1  (number)
// rest is 2  (number)
```

With spread

```
const nums:number[] = [1,2,3,4,5];

const [first, ...rest] = nums;
// first is 1          (number)
// rest is [2,3,4,5]   (number[])
```

With spread on func args

```
function splitIt([f, ...r]: number[]): void {
  console.log("Front", f);  // 5             a number
  console.log("Rest", r);   // [20, 31, 19]  an ARRAY of numbers
}

splitIt([5, 20, 31, 19]);
```

Object Destructuring

```
const gvStudent = {
  name: “John Smith”,
  age: 22,
  email: “smithj@mail.gvsu.edu”,
  inStateTuition: true
}
```

```
const {name, inStateTuition} = gvStudent;
// name is “John Smith”
// inStateTuition is true

const {email, ...info} = gvStudent;
// email is “smithj@mail,.gvsu.edu”
// info is { name: “John Smith”,
             age: 22,
             inStateTuition: true}
```