TypeScript

slides at https://github.com/mvolkmann/talks

R. Mark Volkmann
Object Computing, Inc.
http://objectcomputing.com
Email: mark@objectcomputing.com
Twitter: @mark_volkmann
GitHub: mvolkmann

OCI | TRAINING

What Is TypeScript?

- A superset of JavaScript
- Adds types and more
  - types can be added gradually
  - specifying more types allows TypeScript to find more errors
- Compiles `.ts` files to `.js` files
- Compiler requires Node.js to run
- Created by Microsoft
- https://www.typescriptlang.org/

Talk Assumptions

- Knowledge of JavaScript
  - only covering what TypeScript adds
- Experience with a language that has classes and interfaces
  - like C# or Java
- Using the most strict settings possible
  - otherwise some errors described will not be reported

Benefits

- Catches errors at compile-time
- Provides documentation
- Allows editors to flag errors while typing and provide completion
- Makes refactoring less error-prone
- Makes some tests unnecessary
- Disallows many JavaScript type coercions
  - fewer WAT examples - https://www.destroyallsoftware.com/talks/wat
  - ex. can add number and string (concatenates),
    but cannot add number and object

Creating a TypeScript Project

- Create **`package.json`** file
  - **`npm init`**
- Install **`typescript`** locally
  - **`npm install -D typescript @types/node`**

types for Node projects

- Create **`tsconfig.json`** file
  - **`npx tsc --init`**
  - will contain lots of commented-out options
  - change to match starting point on next slide

npx searches **`./node_modules/.bin`** for executables installed by **`npm`**

tsc is abbreviation for TypeScript Compiler

to see version of TypeScript enter **`tsc -v`**

- Create **`.ts`** files in and under **`src`** directory
- Compile all **`.ts`** files to **`.js`** files
  - **`npx tsc`**
  - compiles all **`.ts`** files under **`src`** directory to **`.js`** files under **`dist`** directory
- Run main **`.js`** file
  - **`node dist/index.js`**

assumes main file is **`src/index.ts`**

tsconfig.json ...

- Configures options for TypeScript compiler
- Only used when **`tsc`** is run with no input files
- Some top-level properties are **`compilerOptions`** and **`include`**
  - others include **`compileOnSave`**, **`exclude`**, **`extends`**, and **`files`**
- Good starting point
  - see descriptions on next slide

```
{
  "compilerOptions": {
    "esModuleInterop": true,
    "module": "commonjs",
    "noImplicitReturns": true,
    "outDir": "dist",
    "strict": true,
    "target": "es5",
 }
  "include": ["src"]
}
```

... tsconfig.json

- **`compilerOptions`** is object with many properties including
  - **`esModuleInterp`** - allows use of ES5 default exports and imports
  - **`jsx`** - values are **`"preserve"`**, **`"react"`**, and **`"react-native"`**
  - **`lib`** - array of APIs assumed to exist; ex. **`["dom", "es2015"]`**
  - **`module`** - module system to use; ex. **`"es2015"`** for modern browsers or **`"commonjs"`** for Node.js
  - **`noImplicitReturns`** - means functions that return a value must do so from ALL code paths
  - **`outdir`** - directory where **`.js`** files should be written; ex. **`"dist"`**
  - **`sourceMap`** - set to **`true`** to generate source map files

    maps generated **`.js`** lines to **`.ts`** lines for debugging
  - **`strict`** - set to **`true`** to require all code to be typed
  - **`target`** - JavaScript version to generate;
    ex. **`"es5"`** for older browsers or **`"es2015"`** for modern browsers and Node.js
- **`include`** is array of directories where **`.ts`** files are found
  - ex. **`"include": ["src"]`**

to see more options enter **`tsc -h`** and look for **`--lib`**

Strict Mode

- Setting **`compilerOption strict`** to **`true`** implies many other options
  - **`alwaysStrict`** - parses in strict mode and emits "**`use strict`**" for each source file
  - **`noImplicitAny`** - raises error on declarations and expressions with implied **`any`** type
  - **`noImplicitThis`** - raises error on **`this`** expressions with implied **`any`** type
  - **`strictBindCallApply`** - enables stricter checking of the **`bind`**, **`call`**, and **`apply`** methods on functions
  - **`strictFunctionTypes`** - checks function type parameters covariantly instead of bivariantly

    see https://codewithstyle.info/Strict-function-types-in-TypeScript-covariance-contravariance-and-bivariance/
  - **`strictNullChecks`**
    - **`null`** and **`undefined`** values are not allowed for every type by default
    - **`null`** and **`undefined`** and are only assignable to themselves and the **`any`** type (except **`undefined`** is assignable to the **`void`** type)
  - **`strictPropertyInitialization`**
    - ensures class properties with a type other than **`undefined`** are initialized in constructor
    - also requires **`strictNullChecks`**

The Flow

- At compile-time **`tsc`**
  - reads **`.ts`** files
  - creates an abstract syntax tree (AST)
  - performs type checking against AST
  - generates one **`.js`** file for each **`.ts`** file
- At run-time a JavaScript Engine
  - reads **`.js`** files
  - generates a different AST
  - generates bytecode from AST
  - evaluates bytecode

since these steps are not visible,
it feels like JavaScript is
an interpreted language

- Another way to compile and run TypeScript code in a Node environment
  - https://github.com/TypeStrong/ts-node
- **`npm i -g typescript ts-node`**
- **`ts-node`** ***`name`*** **`.ts`**

Editor Support

- VS Code has great TypeScript support
  - must enable “Typescript › Validate” in settings
  - hover over a variable to see its type
  - doesn’t honor **`compileOnSave`** setting in **`tsconfig.json`**
    - see https://github.com/microsoft/vscode/issues/973
    - run **`tsc --watch`** in a terminal window to watch for file changes and automatically compile
- Other editors with TypeScript support include Atom, Eclipse, Emacs, Sublime, Vim, Visual Studio, WebStorm, and more

Linting With TSLint

- TSLint will be deprecated in favor of ESLint
  - but currently TSLint may support rules not yet supported by ESLint
- TSLint npm script
  - **`"lint": "tslint --project ."`**
- TSLint installs
  - **`npm install -D tslint`**
  - to generate default **`tslint.json`** enter **`npx tslint --init`**
  - to run enter **`npx tslint --project .`**
- Running TSLint
  - **`npx tslint --project .`**
    or
    `npm run lint`

Linting With ESLint

- ESLint npm script
  - `"lint": "eslint --cache --fix src/**/*.ts"`
- ESLint Installs
  - `npm install -D *` where `*` is
    `eslint`
    `@typescript-eslint/parser`
    `@typescript-eslint/eslint-plugin`
    `eslint-plugin-react`
    `eslint-config-prettier`
    `eslint-plugin-prettier`
- Create `.eslintrc.json`
- Running ESLint
  - asdfas
  - `npm run lint`

```
{
  "extends": [
    "plugin:@typescript-eslint/recommended",
    "prettier/@typescript-eslint",
    "plugin:prettier/recommended",
    "plugin:react/recommended"   <- only for React
  ],
  "parser": "@typescript-eslint/parser",
  "parserOptions": {
    "ecmaVersion": 2018,
    "sourceType": "module",
    "ecmaFeatures": {
      "jsx":  true   <- only for React
    }
  },
  "rules": {   <- can override recommended rules here
  },
  "settings":  {
    "react": {   <- only for React
      "version": "detect"
    }
  }
}
```

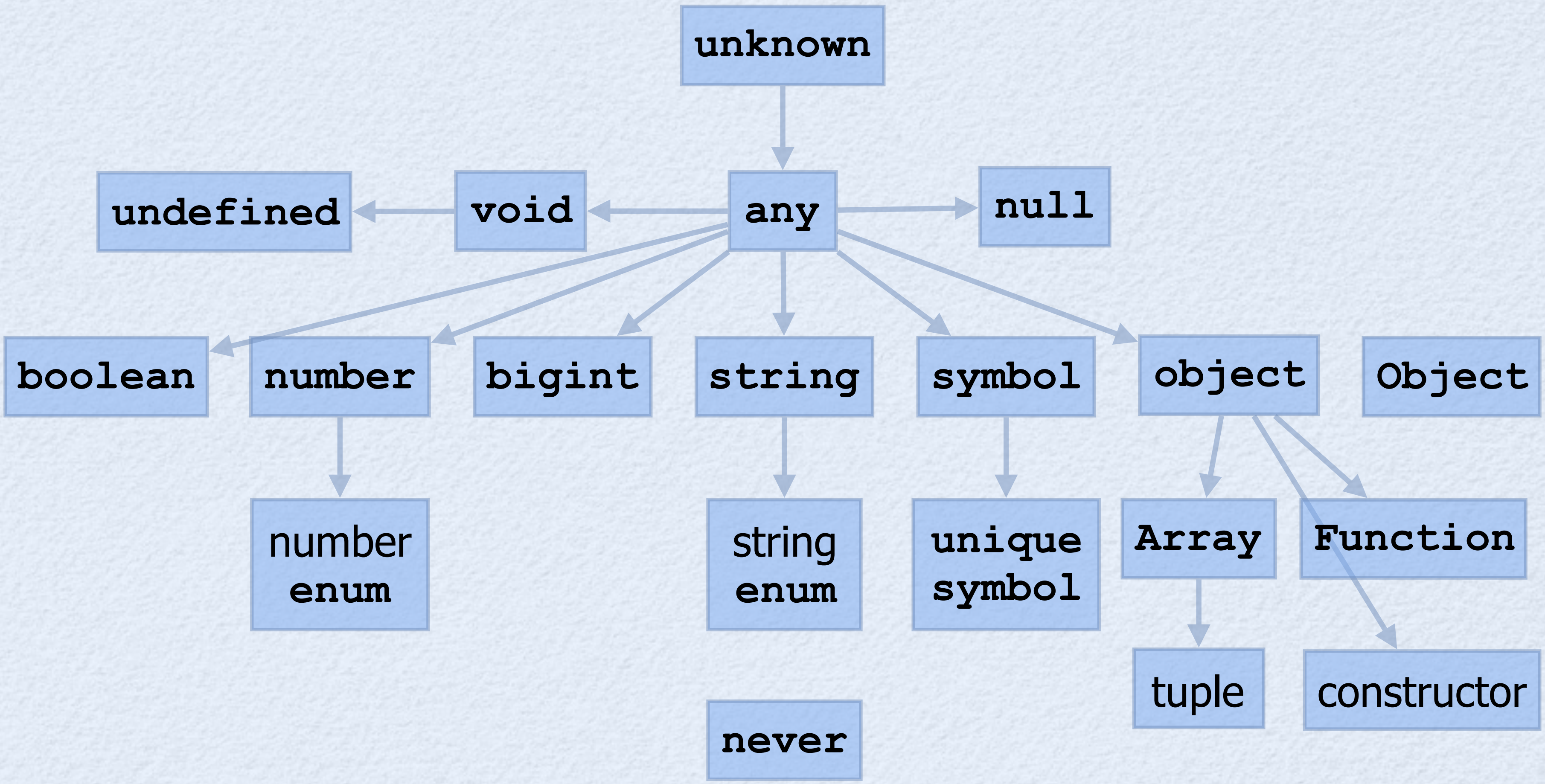
Formatting

- Use Prettier
  - https://prettier.io/
- To install
  - `npm install -D prettier`
- Add script to package.json
  - `"format": "prettier --write src/**/*.ts"`
- To run
  - `npm run format`

Types

- Types define
  - allowed values
  - operations that can be performed on the values
- Example
  - **`number`** type only allows integer and floating point values
  - one operation that can be performed on numbers is multiplication

Type Hierarchy



Primitive Types

- **`boolean`**
  - **`true`** and **`false`**
- **`number`**
  - integer (up to 2^53) and float
- **`string`**
- **`bigint`**
  - holds any size integers (as of 8/30/19 not supported by IE11 or Safari)
- **`symbol`**
  - holds immutable, unique values
  - sometimes used as keys in objects and Maps
- **`null`**
  - represents currently having no value
- **`undefined`**
  - represent having never had a value

Other Builtin Types

- **`any`**
  - default type
  - allows anything
  - avoid using
- **`unknown`**
  - like **`any`**, but requires “refinement” when values are used

Types Not Used For Variables

- **`undefined`** and **`null`**
  - only used in union types, not on their own

```
let name: string;
//console.log(name); // used before being assigned
name = 'Tami';
console.log(name); // Tami
//name = null; // not assignable

let name2: string | null | undefined;
console.log(name2); // undefined
name2 = 'Mark';
console.log(name2); // Mark
name2 = null;
console.log(name2); // null
```

- **`void`**
  - only used as return type of functions that return nothing

```
function printSum(n1: number, n2: number): void {
  console.log(n1 + n2);
}
```

- **`never`**
  - represents something that never happens
  - one use is as return type of functions that never return
  - really tough to come up with a useful example

Non-Primitive Types

- objects
  - described on next slide
- arrays
  - described later
- Any JavaScript class
  - examples include **`Date`**, **`Error`**, **`Function`**, **`Map`**, **`Promise`**, **`RegExp`**, and **`Set`**

Objects w/ Unspecified Properties

- Three ways to declare
- Typically none of these are used
- **`object`**
  - can assign any object or array
  - best option
- **`Object`** or **`{}`**
  - can assign any object, array, or <u>primitive values</u>
  - avoid these
- Cannot assign **`undefined`** or **`null`** to any of these

- Later we will see you to specify object properties with type aliases and interfaces

Union Types

- Define a new type that matched multiple specified types
  - assuming classes **`Dog`** and **`Cat`** are defined, we can define

```
type Pet = Cat | Dog;
```

  - another example

```
type Custom = number | string | undefined;

// If undefined is not an allowed type,
// this is not assignable.
//let c: Custom = undefined;

let c: Custom;
console.log(c); // undefined

c = 1;
console.log(c); // 1
c = 'x';
console.log(c); // x
//c = true; // not assignable
//c = null; // not assignable
```

Enums

- Define list of allowed **`number`** or **`string`** values
- Like objects where keys are strings and values are numbers or strings
- By default values are integers starting from zero
- Can assign specific **`number`** or **`string`** values
  - can even mix **`number`** and **`string`** values, but seems like a bad idea
- When values are numbers keys can be retrieved by value
  - can't do this when values are strings

```
// number enum
enum Color {
  Red, // 0
  Green = 10, // 10
  Blue // 11; next value after Green
}
let c: Color = Color.Blue;
console.log(c); // 11
c = 10; // can set number value
console.log(c); // 10
c = 30; // can set number value not present
console.log(c); // 30, not an error
//c = Color.Yellow; // error, does not exist
console.log(Color[10]); // Green
console.log(Color[12]); // undefined, not error

// string enum
enum HexColor {
  Red = 'F00',
  Green = '0F0',
  Blue = '00F'
}
let hc: HexColor = HexColor.Blue;
console.log(hc); // 00F
//hc = 1; // cannot set numeric value
hc = HexColor.Red;
console.log(hc); // F00
//hc = HexColor['0F0']; // error
```

`const` Enums

- Can access values by key, but cannot access keys by value
- Allows member references to be inlined

```
const enum ConstColor {
  Red, // 0
  Green = 10, // 10
  Blue // 11; next value after Green
}

const cc: ConstColor = ConstColor.Blue;
console.log(cc); // 11
//console.log(ConstColor[10]); // can only access by key
console.log(ConstColor.Green); // 10
console.log(ConstColor['Green']); // 10
```

Use Enums?

- Many developers prefer to use union types instead of enums
  - ex.
```
// Instead of this enum ...
const enum Color { Red, Green, Blue }

// can use this union type.
type Color = 'red' | 'green' | 'blue';

let c: Color = 'red';
c = 'pink'; // "pink" is not assignable
```
- One reason is that enums with number values allow any number to be assigned
  - ex.
```
const enum Color { Red, Green, Blue }
let c: Color = Color.Red; // valid value
c = 19; // not a valid value, but allowed
```

Arrays

- Declare with ***type*[]** or **Array<*type*>**
- Can infer type from initial value
  - ex.
    ```
    const things = [99, 'Gretzky']; // type is (string | number)[]
    ```
- Prefer arrays where all elements have same type
  so TypeScript can better enforce usage
- Type of an untyped array is narrowed when it leaves scope
  - ex.
    ```
    function getThings() {
      const arr = []; // type is any[]
      arr.push(1);
      arr.push('x');
      return arr; // type is (string | number)[]
    }

    const things = getThings(); // type is (string | number)[]
    things.push(new Date()); // error: not a string or number
    ```

Tuples

- Subtype of arrays
- Have a fixed length
- Can hold a different type at each index (heterogeneous)
- Make an element optional by adding **?** after type
  - ex. `type PlayerScore = [string, number?];`

```
type Point = [number, number];

function translate(
  point: Point,
  dx: number,
  dy: number
): Point {
  const [x, y] = point; // destructuring
  return [x + dx, y + dy];
}

const p1: Point = [1, 2];
const p2 = translate(p1, 2, 3);
console.log('p2 =', p2); // [3, 5]
```

```
type PlayerScore = [string, number];
const ps: PlayerScore = ['Mark', 19];
console.log('ps =', ps);

// Usually this approach is better.
interface Player {
  name: string;
  score: number;
};
const p: Player = {name: 'Mark', score: 19};
console.log('p =', p);
```

`readonly` Modifier

- Can apply to
  - arrays and tuples
  - object properties

```
type Numbers = readonly number[]; // primitive array
const n: Numbers = [1, 2, 3];
//n[1] = 7; // only permits reading
//n.push(4); // property "push" does not exist on readonly

type Dates = readonly Date[]; // object array
const d: Dates = [new Date()];
//d[0] = new Date(); // only permits reading

type PlayerScore = readonly [string, number, Date]; // tuple
const ps: PlayerScore = ['Mark', 19, new Date()];
//ps[1] = 20; // cannot assign read-only property

type Lines = string[];
// Can't apply readonly to a type alias,
// only on array and tuple literal types.
//const lines: readonly Lines = ['one', 'two'];

// Can apply to individual properties of object types.
interface ImmutablePoint {
  readonly x: number;
  readonly y: number;
  readonly d: Date;
}

const p1: ImmutablePoint = {x: 1, y: 2, d: new Date()};
//p1.x = 3; // cannot assign to read-only property
//p1.d = new Date(); // cannot assign to read-only property
```

Generic Types

- Can be used in functions, type aliases, classes, interfaces, and methods
- “Type parameters” are names used for parameter types, return types, and variable types
- Specified in angle brackets - ex. `<T, U>`
- Can have any number of them with any names
  - common names are `T`, `U`, and `V`
  - rare to need more than three
- All occurrences of a given type parameter are replaced with the same type at runtime
- Will infer type parameter types at runtime, but they can also be made explicit
- We will see many examples later

`Readonly` Generic Type

- One of the provided utility types
- Use to mark all properties in an object type as **`readonly`**
- Example

```
interface MutablePoint {
  x: number;
  y: number;
  d: Date;
}

const p2: Readonly<MutablePoint> = {x: 1, y: 2, d: new Date()};
//p2.x = 3; // cannot assign to read-only property
//p2.d = new Date(); // cannot assign to read-only property
```

Declaring Variables

- Will infer types when initial values are assigned
  - ex.
    ```
    let score = 0; // infers number
    const teams = 2; // infers 2 because it's immutable
    ```
- Can explicitly specify types
  - ex.
    ```
    let score: number; // defaults to 0
    const teams: number = 2; // silly to specify type here
    ```

Type Aliases

- Can give an alternate name to another type or union type
  - ex.
    ```
    type Age = number;
    ```
    ```
    type Pet = Cat | Dog;
    ```
    ```
    const myAge: Age = 29;
    ```
- Can define a type that allows a fixed set of values
  - ex.
    ```
    type Color = 'red' | 'green' | 'blue';
    ```
    ```
    const color: Color = 'blue';
    ```
- Can define an object "**shape**"
  - ex.
    ```
    type Address = {
      street: string,
      city: string,
      state: string,
      zip: number
    };
    ```
    ```
    const myAddress: Address = {
      street: '123 Some St.',
      city: 'Somewhere',
      state: 'Missouri',
      zip: 12345
    };
    ```
    most developers prefer using `interface` instead of `type` for this
- Type aliases are block-scoped
  - like `const` and `let`

More on Shapes

- Properties are required by default
- Cannot add new properties
- To make optional add **`?`** after names
- To make a property read-only, add **`readonly`** before name
- To allow arbitrary additional properties add an "index signature"

```
[key: keyType]: valueType
```

  - **`key`** can have a different name, but using **`key`** is common
  - ***`keyType`*** must be **`string`** or **`number`**
  - ***`valueType`*** can be any type including **`any`**
- Example

```
type Address = {
  street: string,
  city?: string,
  state?: string,
  readonly zip: number,
  [key: string]: any
}
```

Generic Type Aliases

- Can define types that use specific types at runtime
  - ex.

```
type Range<T> = {
  min: T,
  max: T
};

const numberRange: Range<number> = {min: 1, max: 10};
const letterRange: Range<string> = {min: 'a', max: 'f'};
const letterRange: Range<Team> = {
  min: new Team('Cubs'),
  max: new Team('Cardinals')
};
```

Classes ...

- Must declare all properties

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

JavaScript

```
class Person {
  name: string; // required

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

TypeScript

... Classes ...

- Provide template for creating instances using **`new`** keyword
- Access modifiers
  - **`public`**
    - can access from anywhere; default
  - **`protected`**
    - can access from all instances of this class and subclasses
  - **`private`**
    - can only access from instances of this class
  - can apply to constructor parameters as short-cut for assigning to instance properties

can omit parens if constructor takes no arguments

```
class Demo {}
const d1 = new Demo;
```

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

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

class Person2 {
  // Shortcut for what Person constructor does.
  constructor(
    public name: string,
    private age: number
  ) {}
}

const p1 = new Person('Dylan', 21);
const p2 = new Person2('Paige', 16);
```

without **`public`**, the **`name`** instance property will not be set

... Classes ...

- Can declare **`static`** properties and methods
  - not associated with any instance
  - accessed with class name
- Can declare **`readonly`** instance variables
  - must set in constructor and cannot be changed

```
class Widget {
  private static count: number = 0;
  readonly creationDate: Date;

  constructor() {
    this.creationDate = new Date();
    Widget.count++;
  }

  static getCount() {
    return Widget.count;
  }
}

let w = new Widget();
console.log('first Widget =', w);
w = new Widget;
console.log('second Widget =', w);
w = new Widget;
console.log(Widget.getCount()); // 2
```

... Classes ...

- Classes can
  - **`extend`** one other class to inherit from it
    - can override methods

```
class Car extends Vehicle { ... }
```

  - **`implement`** any number of interfaces
    - must implement all methods

```
class Car extends Vehicle implements Recyclable, Sortable { ... }
```

  - be marked as **`abstract`** when only intended to be used as a superclass
    - can't instantiate with **`new`**
    - extending classes must implement methods marked as **`abstract`** or also be **`abstract`**

```
abstract class Vehicle { ... }
```

... Classes

- Subclasses
  - must call superclass constructor from constructor with **super(*args*)** even if superclass has no constructor or has one that doesn't take arguments
    - must call **super** before accessing **this**
  - can call superclass methods with **super.*methodName*(*args*)**

```
class Car extends Vehicle {
  constructor() {
    super();
    ...
  }

  report() {
    console.log('Car:');
    super.report();
  }
}
```

Generic Classes

- Can define classes that use specific types at runtime

```
class Pair<T> {
  constructor(public first: T, public second: T) {}
}

const p1 = new Pair<number>(3, 19);
const p2 = new Pair<string>('foo', 'bar');
console.log(p1); // Pair { first: 3, second: 19 }
console.log('p2 =', p2); // Pair { first 'foo', second: 'bar' }
```

- Useful to type result of a **`Promise`**
  - ex.

```
const promise = new Promise<Person>(resolve => {
  // Perhaps retrieve data with a REST call here.
  resolve(new Person(...));
});
const person = await promise;
```

Interfaces ...

- Interfaces can
  - describe method signatures that implementing classes will implement
    - instance methods only, not **`static`**
  - describe properties that instances will have; overlaps with shape type aliases
- Interfaces cannot
  - specify access modifiers; all properties are **`public`**
  - define constructor signatures
    - can in an **`abstract class`**
  - define default method implementations

```
interface Vehicle {
  maxSpeed: number;
  accelerate(speed: number): void
}
```

... Interfaces

- Similar to shape type aliases
  - both can define object shapes
  - includes data and method signatures
- Differences between interfaces and shape type aliases
  1. interfaces can extend another interface, a type alias, or a class (weird); type aliases cannot
  2. interfaces can be defined multiple times in the same scope (definitions are merged); type aliases cannot (not a significant benefit)
  3. the right side of a type alias can be another type (truly an alias)
- Bottom line
  - no strong reason to prefer one over the other
  - but it seems the community prefers interfaces when either will do

Record

- One of the provided utility types
- Defines a type that is a mapping from a set of keys to values
  - keys must be strings, numbers, or symbols
  - values can be any type
- Example

```
type Fruit = 'apple' | 'banana' | 'cherry' | 'lime';
type Color = 'red' | 'orange' | 'yellow' | 'green' | 'blue';
const fruitMap: Record<Fruit, Color> = {
  apple: 'red',
  banana: 'yellow',
  cherry: 'red',
  lime: 'green' // error if this line is missing
};

const color: Color = fruitMap.banana;
console.log(color); // yellow
```

- Benefit over using an Object or Map
  - provides "totality checking" (a.k.a exhaustiveness)
    to ensure there is a mapping for all possible keys

Additional Utility Types

- These create a new type based on existing types
  - **`Partial<T>`** - same as **`T`** but all properties are optional
  - **`Required<T>`** - same as **`T`** but all properties are required
  - **`Pick<T, K>`** - same as **`T`** but only includes properties specified by union type **`K`**
  - **`Omit<T, K>`** - same as **`T`** but excludes properties specified by union type **`K`**
  - **`Extract<T, U>`** - same as **`T`** but only includes properties that can be assigned to type **`U`**
  - **`Exclude<T, U>`** - same as **`T`** but excludes properties that can be assigned to type **`U`**
  - **`NonNullable<T>`** - same as **`T`** but does not allow the values **`undefined`** and **`null`**
    - considers the type of T, not its properties like the previous utility types

Function Parameter/Return Types

- Can specify parameter types and return type in any kind of function
  - named, expression, or arrow

```
// Named function
// Same as String repeat method.
function stringRepeat(text: string, repeat: number): string {
  let result = '';
  for (let i = 0; i < repeat; i++) {
    result += text;
  }
  return result;
};

// Example call
const santaSays = stringRepeat('Ho ', 3); // 'Ho Ho Ho '

// Function expression
const stringRepeat = function (text: string, repeat: number): string {
  ...
};

// Arrow function
const stringRepeat = (text: string, repeat: number): string => {
  ...
};
```

Function Signature Types

- Defines parameter types and return types of functions defined elsewhere
  - ex.
    ```
    type StrNumFn = (s: string, n: number) => string;
    ```
  - parameter names are just for documentation; implementations can use other names
  - parameter default values cannot be specified, but can be in implementations
  - return type is required, unlike in function definitions
- Can use as type of functions
  - when defined - see example on next slide
  - when passed as argument to another function

    useful for callback functions

    ```
    function processStr3(text: string, fn: StrNumFn): string {
      return text.length > 0 ? fn(text, 3) : 'empty';
    }
    ```

Parameter Types Inferred

- Parameter types are not inferred unless the function itself is typed
  - using a "function signature type"
- Return type can be inferred by **`return`** statements
  if the type of each **`return`** statement can be inferred

```
// text parameter type is inferred to be string.
// repeat parameter type is inferred to be number.
// return type is inferred to be string.
const stringRepeat: StrNumFn = (text, repeat) => {
  let result = '';
  for (let i = 0; i < repeat; i++) {
    result += text;
  }
  return result;
};
```

not necessary to specify parameter types or return type; could specify parameter default values

Optional/Default/Variadic Parameters

- To make a parameter optional, add `?` after its name
  - only for last parameters
  - ex.
```
function stringRepeat(text: string, repeat?: number): string { ... };
```
- To give parameter a default value, add `= value`
  - can infer type from default value
  - ex.
```
function stringRepeat(text: string, repeat = 1): string { ... };
```
  - more common than optional parameters
- Variadic functions
  - take variable number of arguments using “rest parameter” at end
  - same as in JavaScript
  - ex.
```
function labeledSum(label: string, ...values: number[]): string {
  return label + ': ' + values.reduce((acc, v) => acc + v);
};
console.log(labeledSum('total', 1, 2, 3)); // total: 6
```

Functions That Use `this`

- In functions that use `this`,
  can declare its type as if it is the first parameter
  even though it's not passed that way
  - rarely used feature; better to define as a method in a class
  - ex.

```
type Person = {
  name: string;
  age: number;
};

function greetTeen(this: Person, greeting: string) {
  if (13 <= this.age && this.age <= 19) {
    console.log(greeting, this.name);
  }
}

const p1: Person = {name: 'Dylan', age: 21};
const p2: Person = {name: 'Paige', age: 16};
greetTeen.call(p1, 'Yo'); // no output
greetTeen.call(p2, 'Yo'); // Yo Paige
```

Other Kinds of Functions

- Overloaded functions - not commonly used
- Generator functions - not commonly used

Generic Functions

- Generic functions
  - **`Array map`** method is a good example
  - if written as a function instead of a method, would look like this

```
function map<T, U>(array: T[], fn: (item: T) => U): U[] {
  const result: U[] = [];
  for (const item of array) {
    result.push(fn(item));
  }
  return result;
}

const numbers = [1, 2, 3]; // type inferred as number[]
//const doubled = map(numbers, n => n * 2);
const double = (n: number): number => n * 2;
const doubled = map<number, number>(numbers, double);
console.log(doubled); // [2, 4, 6]

const words = ['apple', 'banana', 'cherry'];
const lengths = map<string, number>(words, s => s.length);
console.log(lengths); // [5, 6, 6]
```

can remove these
type parameter types
since they can be inferred

Bounded Polymorphism

- Says a type parameter cannot be just any type, but must be a type that implements a given interface or extends a given class
- Consider differences between these

```
function foo(bar: Bar): Bar { ... }
function foo<T extends Bar>(bar: T): T { ... }

class Foo {
  constructor(bar: Bar) { ... }
}
class Foo<T extends Bar> {
  constructor(bar: T) { ... }
}
```

- In all cases **`bar`** is an object has a type of **`Bar`**
- In cases that use **`extends`**, the type of **`T`** is the actual subclass of **`Bar`** being used
- If this seems confusing it is because it is!
  - hard to come up with a good example
- Typically the difference is not important and the non-generic form is used

Structural Typing

- When determining object compatibility,
  TypeScript uses structuring typing, not nominal typing (by name)
- Means compatibility is based on their properties, not classes
- Example

```
class Cat {
  constructor(public name: string) {}
}

class Dog {
  constructor(public name: string) {}
}

const cat = new Cat('Whiskers');
const neither = {foo: 'bar'};

let dog: Dog;
dog = cat; // allowed because it has all Dog properties
dog = neither; // not allowed because it doesn't have all Dog properties
```

both classes have the same set of properties with the same types

Flow uses nominal typing

Script vs. Module Mode

- Ways in which modules differ from scripts
  - always executed in strict mode
  - can use **`import`** and **`export`** statements
  - loaded once regardless of how many times they are imported
  - top-level definitions have file scope and are only accessible outside the file if exported
  - top-level value of **`this`** is **`undefined`**, not **`window`**
- Source files are processed in module mode
  if they contain at least one **`import`** or **`export`** statement
- When neither is needed, module mode can be forced
  - by including **`export {}`**, typically at top of file
  - desirable to avoid conflicts between declarations
    of the same names across multiple source files

DefinitelyTyped

- Many open source libraries include TypeScript type definitions when they are installed from npm
- When they don't, often these are available from DefinitelyTyped
  - "The repository for high quality TypeScript type definitions"
  - http://definitelytyped.org/
- Types are published in npm under **`@types`** scope
- To install type definitions for a library
  - **`npm install --save-dev @types/`*`library-name`***
- These type definitions are automatically included by compiler
- Submit pull requests to
  - contribute missing type definitions
  - contribute corrections to existing type definitions

Libraries With No Type Definitions

- To get type checking for a library that doesn't provide its own type definitions and has none in DefinitelyTyped
  - **option #1**: contribute type definitions to the library
  - **option #2**: contribute type definitions to DefinitelyTyped
  - **option #3**: define in own project,
perhaps in **`src/types.ts`**
and import where needed

Type Declaration Files

- TypeScript compiler and editors like VS Code uses these for type checking
- Can generate from TypeScript source files
  - `tsc -d` ***`name`***`.ts`
- Can create manually
  - typically only for JavaScript libraries that do not supply them
  - declare all types that should be visible to using code, omitting privately used types
- Contain "ambient declarations"
  - "ambient" distinguishes from normal declarations
  - use **`declare`** keyword for all but interfaces
- File extension
  - **`.d.ts`** when there is a corresponding **`.js`** file
  - otherwise can be **`.ts`** or **`.d.ts`**

Ambient Declaration Examples

.ts file

```
export const MONTH = 'April';

export let counter: number = 0;

export function double(n: number): number {
  return n * 2;
}

export type Fruit = {
  name: string;
  color: string;
  volume: number;
};

export interface Sortable<T> {
  sort(): T;
}

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

.d.ts file generated by tsc -d

```
export declare const MONTH = "April";
export declare let counter: number;
export declare function double(n: number): number;
export declare type Fruit = {
    name: string;
    color: string;
    volume: number;
};
export interface Sortable<T> {
    sort(): T;
}
export declare class Person {
    name: string;
    age: number;
    constructor(name: string, age: number);
}
```

Shipping TS Libraries ...

- Generate type declarations
- Add **`types`** key in **`package.json`**
  - indicates that type declarations are included
  - value is path to **`.d.ts`** file
- Add npm script in **`package.json`**
  - to generate new type declarations every time changes are published
- Include source maps
- Choose target module format appropriate for using code
  - typically ES5 for browsers

… Shipping TS Libraries

- Omit `.ts` files and only include generated `.js` and `.d.ts` files
  - including `.ts` files increases size of download for little value
  - if `.ts` files are included and generated `.js` files are not,
    users will have to compile them
  - in `.npmignore`, add `src` directory
  - in `.gitignore`, add `dist` directory

TS in React Components ...

function-based

```
import React from 'react';

type Props = {
  name: type,
  ...
};

function ComponentName(props: Props) {
  ...
}

export default ComponentName;
```

class-based

rarely used now that React supports "hooks"

```
import React, {Component} from 'react';

type Props = {
  name: type,
  ...
};

type State = {
  name: type,
  ...
};

class ComponentName
extends Component<Props, State> {
  state = {
    name: type,
    ...
  };

  render() {
    ...
  }
}

export default ComponentName;
```

... TS in React Components

- Use React’s synthetic event types instead of DOM event types
  - see type definitions at https://github.com/DefinitelyTyped/DefinitelyTyped/blob/master/types/react/index.d.ts
  - includes **`ChangeEvent`**, **`FocusEvent`**, **`KeyboardEvent`**, **`MouseEvent`**, **`TouchEvent`**, **`WheelEvent`**, and more
- **`useState`** hook
  - if TypeScript can’t infer type from initial value, specify it like this

```
const [name, setName] = useState<type>(initialValue);
```

Migrating JS Projects to TS ...

After each step,
run **`npm run compile`**
and fix reported errors.

- Five recommended steps
- Step 1 - Add use of **`tsc`**
  - **`npm install -D typescript`**
  - add to **`compilerOptions`** in **`tsconfig.json`**
    - **`"allowJs": true, "outDir": "dist", "target": "ES5"`**
  - add to **`include`** in **`tsconfig.json`**
    - **`"include": ["src"]`**
  - add script to **`package.json`** - **`"compile": "tsc"`**

generated files will be
written to **`dist`** directory

- Step 2 - Enable type checking **`.js`** files
  - add to **`compilerOptions`** in **`tsconfig.json`**
    - **`"checkJs": true`**
  - can add **`// @ts-nocheck`** at top of **`.js`** files with too many errors to fix now
  - may want to temporarily add "**`noImplicitAny": false`** to **`compilerOptions`**
    because **`any`** will be a commonly inferred type

TypeScript type checking
is more lenient on `.js` files.
- function parameters are optional
- class property types are
  inferred based on usage
- can assign extra properties to objects

... Migrating JS Project to TS

```
/**
 * description
 * @param name {type} description
 * @return {type} description
 */
```

- Step 3 - Optionally add JSDoc comments
  - before functions to provided parameter and return types
  - supported JSDoc annotations are listed at https://www.typescriptlang.org/docs/handbook/type-checking-javascript-files.html#supported-jsdoc
- Step 4 - Rename `.js` files to `.ts`
  - perhaps one file at a time, rename, compile, and fix errors
- Step 5 - Stop processing `.js` files and make strict
  - change `compilerOptions` in `tsconfig.json`
    - `"allowJs": false, "checkJs": false, "strict": true`

Processing Details ...

- When a **`.ts`** file imports a **`.js`** file
  it looks for a corresponding **`.d.ts`** file
  in the same directory as **`.js`** file
- If not found
  - if **`allowJs`** and **`checkJs`** are **`true`**, types are inferred
  - if **`allowJs`** and **`checkJs`** are **`false`**, types are treated as **`any`**

... Processing Details

- When a **`.ts`** file imports from **`node_modules`** it looks for type definitions in **`src/types.ts`**
  - can manually create type definitions here
- If not found
  - looks for **`types`** or **`typings`** key in **`package.json`** of the npm package that points to a **`.d.ts`** file
- If not found
  - look for **`.d.ts`** file in **`node_modules/@types/`*`package-name`*** working upward to top **`node_modules`** directory of the app (supports nested dependencies)
- If not found
  - use lookup process for imports not under **`node_modules`**
- Can override where TypeScript looks for type declarations
  - by setting **`typeRoots`** in **`compilerOptions`**, but doing this is not common

Whitelisting

- When an npm package doesn't include type definitions
  and they aren't available in DefinitelyTyped,
  there are three options
- 1) Define yourself, perhaps in **`src/types.ts`**
  - and consider contributing type definitions to the package or DefinitelyTyped
- 2) Whitelist specific imports
  - by preceding them with **`// @ts-ignore`**
- 3) Whitelist all usages of the package
  by adding a line to **`src/types.ts`**
  - **`declare module '`*`package-name`*`';`**

Polyfills

- **`tsc`** transpiles to many target environments, but doesn't provide any polyfills
- Popular polyfill options
  - **`core-js`**, **`@babel/polyfill`**, and polyfill.io
  - can use Babel to get support for language features that TypeScript doesn't yet support
- Set **`lib`** in **`compilerOptions`** to indicate which features have been polyfilled

Using TypeScript with Node.js

- Recommended **`tsconfig.json`**

```
{
  "compilerOptions": {
    "esModuleInterop": true,
    "module": "commonjs",
    "noImplicitReturns": true,
    "outDir": "dist",
    "sourceMap": true,
    "strict": true,
    "target": "es2015",
 }
  "include": ["src"]
}
```

compiles **`import`** statements to **`require`** calls and **`export`** statements to **`module.exports`**

npm package **`source-map-support`** is needed to enable this

Features Coming in TS 3.7

- Optional Chaining

```
// Old way
const zip = person && person.address ? person.address.zip : undefined;

// New way
const zip = person?.address?.zip;
```

- Null Coalescing

```
// Old way - chooses option2 if option1 is any falsy value
// including false, zero, or empty string
const option = option1 || option2;

// New way - chooses option2 only if option1 is undefined or null
const option = option1 ?? option2;
```

Do’s and Don’ts

- https://www.typescriptlang.org/docs/handbook/declaration-files/do-s-and-don-ts.html
- Recommends never using wrapper types
  `Number`, **`String`**, **`Boolean`**, **`Symbol`**, and **`Object`**

Things Never Needed ...

- Tuples with a minimum length using spread
    - ex. `type badIdea = [number, number, ...number[]] // contains 2 or more numbers`
- Writing functions that use **`this`**
    - and declaring the type of **`this`** as the first parameter
- Overloaded functions
- Bounded polymorphism with multiple constraints
- Generic type defaults
- Using **`this`** as a return type of a method in a superclass

... Things Never Needed ...

- Intersection types
  - match only what multiple types have in common
  - can’t think of a good example
- Interfaces extending a type alias or class
- Comparing classes
- Defining a constructor signature in an interface with **`new`**
- Decorators
- **`private`** constructors to simulate final classes
  - can’t be extended or directly instantiated (must use static factory methods)
- **`as const`**
- Type assertions
- Tagged Unions

... Things Never Needed ...

- Keying-in operation
  - extracts a shape type from another
- `keyof` operator
  - gets property names from a shape
- Mapped types, including built-in ones
- Companion object pattern
- Creating tuples with a function
  - instead of `[]` syntax
- User-defined type guards
- Conditional types, including built-in ones
- Distributive conditionals
- `infer` keyword

... Things Never Needed

- Non-null assertions
  - just define non-nullable types
- Definite assignment assertions
- Type branding
- Extending prototypes
- Dynamic imports
- TypeScript namespaces
- Declaration merging
- Project references
- Triple-slash directives
- **`amd-module`** directive
- Type utilities other than **`Record`**