codecademy

Types

TypeScript

The TypeScript programming language is a superset of JavaScript that adds types to JavaScript using a set of tools called a type system.

Primitive Types

All “primitive”, or built-in data types in JavaScript are recognized by TypeScript.

```
"Hello, world!"  // string
42               // number
true             // boolean
null
undefined
```

TypeScript Type Inference

Type inference assumes the expected type of the variable throughout a program based on the data type of the value initially assigned to it at declaration. Type inference will log a complaint if the variable is later reassigned to a different type.

```
let first = 'Anders';

first = 1337; // Type 'number' is not
assignable to type 'string'
```

The Shape of an Object

TypeScript knows the *shape* of an object—what member properties it does or doesn’t contain. TypeScript will log an error if the code attempts to access members of an object known not to exist. It may even suggest possible corrections.

```
let firstName = 'muriel!';

console.log(firstName.toUppercase());

// error: Property 'toUppercase' does not
exist on type 'string'. Did you mean
'toUpperCase'?
```

Typescript any

When a variable is declared without being assigned an initial value, TypeScript considers it to be of type `any`. A variable of this type can be reassigned without generating any error from TypeScript.

```
let first;
first = 'Anders';
first = 1337;
```

TypeScript Supports Type Annotations

Adding a type annotation ensures that a variable can only ever be assigned to that type. This can be useful when declaring a variable without an initial value. Type annotations get removed when compiled to JavaScript. The type declaration is provided by appending a variable with a colon (`:`) and the type (eg. `number`).

```
let first: string;
first = 'Anders';

// Error: Type 'number' is not assignable
to type 'string'
first = 1337;
```

codecademy

Complex Types

TypeScript Type for One-dimensional Array

The type annotation for a one-dimensional array in TypeScript is similar to a primitive data type, except we add a `[]` after the type.

```
// zipcodes is an array of strings
let zipcodes: string[] = ['03255',
'02134', '08002', '03063'];

// Pushing a number to zipcodes will
generate an error
// Error: Argument of type 'number' is
not assignable to parameter of type
'string'.
zipcodes.push(90210);
```

TypeScript Generic Type for One-Dimensional Array

The type for a one-dimensional array in TypeScript can be annotated with `Array<T>`, where `T` stands for the type.

```
// zipcodes is an array of strings
let zipcodes: Array<string> = ['03255',
'02134', '08002', '03063'];

// Pushing a number to zipcodes will
generate an error
// Error: Argument of type 'number' is
not assignable to parameter of type
'string'.
zipcodes.push(90210);
```

TypeScript Type for Multi-dimensional Array

The type for a multi-dimensional array can be annotated by adding an extra `[]` for each extra dimension of the array.

```
// one-dimensional arrays
let zipcodesNH: string[] = ['03255',
'03050', '03087', '03063'];
let zipcodesMA: string[] = ['02334',
'01801'];

// two-dimensional array
let zipcodes: string[][] = [zipcodesNH];

// Pushing a one-dimensional array to
a two-dimensional array
zipcodes.push(zipcodesMA);
console.log(zipcodes); // prints
[["03255", "03050", "03087", "03063"],
["02334", "01801"]]
```

TypeScript Empty Array Initialization

An array of any dimension can be initialized as an empty array without generating any error.

```
// one-dimensional empty array
let axis: string[] = [];

// two-dimensional empty array
let coordinates: number[][] = [];

axis.push('x');
console.log(axis);        // prints ["x"]

coordinates.push([3, 5]);
coordinates.push([7]);
console.log(coordinates); // prints [[3,
5], [7]]
```

TypeScript Tuple Type

An array that has a fixed size of similar or different element types arranged in a particular sequence is defined as a tuple in TypeScript.

```
// This is an array
let header: string[] = ['Name', 'Age',
'Smoking', 'Salary'];
// This is a tuple
let profile: [string, number, boolean,
number] = ['Kobe', 39, true, 150000];
```

TypeScript Tuple Type Syntax

To annotate a tuple in TypeScript, add a colon (`:`) followed by square brackets (`[...]`) containing a list of comma-separated types.

```
// This is a tuple
let profile: [string, number, boolean,
number] = ['Kobe', 39, true, 150000];

profile[2] = 'false';   // Error: Type
'string' is not assignable to type
'boolean'.
profile[3] = null;      // Error: Type
'null' is not assignable to type
'number'.
```

TypeScript Tuple Type Length and Order

A tuple in Typescript is declared with a fixed number of elements and hence, cannot be assigned to a tuple with a different number of elements. Similarly, a tuple maintains a strict ordering of its elements and therefore, the type for each element is enforced. A transcompiler error will be generated if any of these conditions is violated.

```
let employee: [string, number]
= ['Manager', null];
// Error: Type 'null' is not assignable
to type 'number'.

let grade: [string, number, boolean] = [
'TypeScript', 85, true, 'beginner'];
/*
Error: Type '[string, number, true,
string]'
is not assignable to type '[string,
number, boolean]'.
Source has 4 element(s) but target allows
only 3.
*/
```

TypeScript Tuple Array Assignment

Although a tuple may have all elements of the same type and resembles an array, a tuple is still its own type. A tuple cannot expand, while an array can. Hence, assigning an array to a tuple that matches the same type and length will generate an error.

```
// This is a tuple
let eventDate: [string, string]
= ['January', '2'];

// This is an array
let newDate: string[] = ['January',
'12'];

eventDate = newDate;
/*
Error: Type 'string[]' is not assignable
to type '[string, string]'.
Target requires 2 element(s) but source
may have fewer.
*/
```

TypeScript Array Type Inference

When an array variable is declared without an explicit type annotation, TypeScript automatically infers such a variable instance to be an array instead of a tuple.

```
let mixed = ['one', 2, 3, 'four'];
mixed[4] = 5;                  // no error
because an array is expandable
console.log(mixed);            // prints
["one", 2, 3, "four", 5]
```

TypeScript Array Type Inference on Tuple `.concat()`

The JavaScript method, `.concat()` can be called on a TypeScript tuple, and this produces a new array type instead of a tuple.

```
// This is a tuple
const threeWords: [ string, number,
string] = ['Won', 5, 'games'];

// Calling .concat() on a tuple returns
an array
let moreWords
= threeWords.concat(['last', 'night']);

// An array is expandable
moreWords[5] = ('!');

console.log(moreWords);
// This prints ["Won", 5, "games",
"last", "night", "!"]
```

TypeScript Function Rest Parameter Any Array Type

A rest parameter inside a function is implicitly assigned an array type of `any[]` by TypeScript.

```
const sumAllNumbers = (...numberList):
number => {
  // Error: Rest parameter 'numberList'
implicitly has an 'any[]' type.
  let sum = 0;
  for (let i=0; i < numberList.length;
i++) {
    sum += numberList[i];
  }
  return sum;
}

// Notice third argument is a string
console.log(sumAllNumbers(100, 70,
'30'));
// Prints a string "17030 instead of
a number 200
```

TypeScript Function Rest Parameter Explicit Type

Explicitly type annotating a rest parameter of a function will alert TypeScript to check for type inconsistency between the rest parameter and the function call arguments.

```
const sumAllNumbers = (...numberList:
number[]): number => {
  let sum = 0;
  for (let i=0; i < numberList.length;
i++) {
    sum += numberList[i];
  }
  return sum;
}

console.log(sumAllNumbers(100, 70,
'30')); // Error: Argument of type
'string' is not assignable to parameter
of type 'number'.
```

TypeScript Tuple Type Spread Syntax

Spread syntax can be used with a tuple as an argument to a function call whose parameter types match those of the tuple elements.

```
function modulo(dividend: number,
divisor: number): number {
  return dividend % divisor;
}

const numbers: [number, number] = [6, 4];

// Call modulo() with a tuple
console.log(modulo(numbers));
// Error: Expected 2 arguments, but got
1.
// Prints NaN

// Call modulo() with spread syntax
console.log(modulo(...numbers));
// No error, prints 2
```

TypeScript Enum Type

A programmer can define a set of possible values for a variable using TypeScript's complex type called enum.

```
enum MaritalStatus {
  Single,
  Married,
  Separated,
  Divorced
};

let employee: [string, MaritalStatus,
number] = [
  'Bob Jones',
  MaritalStatus.Single,
  39
];
```

TypeScript Numeric and String Enum Types

TypeScript supports two types of enum: numeric enum and string enum. Members of a numeric enum have a corresponding numeric value assigned to them, while members of a string enum must have a corresponding string value assigned to them.

```
// This is a numeric enum type
enum ClassGrade {
  Freshman = 9,
  Sophomore,
  Junior,
  Senior
};

// This is a string enum type
enum ClassName {
  Freshman = 'FRESHMAN',
  Sophomore = 'SOPHOMORE',
  Junior = 'JUNIOR',
  Senior = 'SENIOR'
}

const studentClass: ClassName
= ClassName.Junior;
const studentGrade: ClassGrade
= ClassGrade.Junior;

console.log(`I am a ${studentClass} in
${studentGrade}th grade.`);
// Prints "I am a JUNIOR in 11th grade."
```

TypeScript Numeric Enum Type Initializers

By default, TypeScript assigns a value of `0` to the first member of a numeric enum type and auto-increments the value of the rest of the members. However, you can override the default value for any member by assigning specific numeric values to some or all of the members.


```
// This numeric enum type begins with
a 1, instead of the default 0
enum Weekdays {
  Monday = 1,
  Tuesday,
  Wednesday,
  Thursday,
  Friday
}

// This is a numeric enum type with all
explicit values
enum Grades {
  A = 90,
  B = 80,
  C = 70,
  D = 60
}

// This numeric enum type has only some
explicit values
enum Prizes {
  Pencil = 5,
  Ruler,     // No error: value is 6
  Eraser = 10,
  Pen        // No error: value is 11
};

const day: Weekdays = Weekdays.Wednesday;
const grade:Grades = Grades.B;
const prize:Prizes = Prizes.Pen;
console.log(`On day ${day} of the week,
I got ${grade} on my test! I won a prize
with ${prize} points!`);
// Prints "On day 3 of the week, I got 80
on my test! I won a prize with 11
points!"
```

TypeScript Numeric Enum Variable Assignment

You can assign a valid numeric value to a variable whose type is a numeric enum.

```
enum Weekend {
  Friday = 5,
  Saturday,
  Sunday
};

// Assign a valid value of Weekend
const today: Weekend = 7;       // No error
console.log(`Today is the ${today}th day of the week!`);
// Prints "Today is the 7th day of the week!"
```

TypeScript String Enum Variable Assignment

Unlike a numeric enum type which allows a number to be assigned to its member, a string enum type does not allow a string to be assigned to its member. Doing so will cause a TypeScript error.

```
enum MaritalStatus {
  Single = 'SINGLE',
  Married = 'MARRIED',
  Separated = 'SEPARATED',
  Divorced = 'DIVORCED',
  Widowed = 'WIDOWED'
};

// Assign a string to a string enum type
let eligibility: MaritalStatus;
eligibility = 'SEPARATED';
// Error: Type '"SEPARATED"' is not assignable to type 'MaritalStatus'.

eligibility
= MaritalStatus.Separated;  // No error
```

TypeScript Object Type

A JavaScript object literal consists of property-value pairs. To type-annotate an object literal, use the TypeScript object type and specify what properties must be provided and their accompanying value types.

```
// Define an object type for car
let car: {make: string, model: string,
year: number};

car = {make: 'Toyota', model: 'Camry',
year: 2020}; // No error
car = {make: 'Nissan', mode: 'Sentra',
year: 2019};
/*
Error: Type '{make: string; mode: string;
year: number;}' is not assignable to
type '{make: string; model: string; year:
number;}'.
Object literal may only specify known
properties, but 'mode' does not exist in
type '{make: string; model: string; year:
number;}'.
Did you mean to write 'model'?
*/
car = {make: 'Chevrolet', model: 'Monte
Carlo', year: '1995'};
// Error: Type 'string' is not assignable
to type 'number'.
```

TypeScript Type Alias

Instead of redeclaring the same complex object type everywhere it is used, TypeScript provides a simple way to reuse this object type. By creating an alias with the `type` keyword, you can assign a data type to it. To create a type alias, follow this syntax:

`type MyString = string;`

```
// This is a type alias
type Student = {
  name: string,
  age: number,
  courses: string[]
};

let boris: Student = {name: 'Boris', age:
35, courses: ['JavaScript',
'TypeScript']};
```

TypeScript Multiple Alias References

codecademy

You can create multiple type aliases that define the same data type, and use the aliases as assignments to variables.

```
// This is also a type alias with the
same type as Student
type Employee = {
  name: string,
  age: number,
  courses: string[]
}

let studentBoris: Student = {name:
'Boris', age: 35, courses: ['JavaScript',
'TypeScript']};
let employeeBoris: Employee
= studentBoris;     // No error
console.log(studentBoris ===
employeeBoris);    // Prints true
```

TypeScript Function Type Alias

In JavaScript, a function can be assigned to a variable. In TypeScript, a function type alias can be used to annotate a variable. Declare a function type alias following this syntax:

`type NumberArrayToNumber = (numberArray: number[]) => number`

```
// This is a function type alias
type NumberArrayToNumber = (numberArray:
number[]) => number;

// This function uses a function type
alias
let sumAll: NumberArrayToNumber
= function(numbers: number[]) {
  let sum = 0;
  for (let i=0; i < numbers.length; i++)
{
    sum += numbers[i];
  }
  return sum;
}

// This function also uses the same
function type alias
let computeAverage: NumberArrayToNumber
= function(numbers: number[]) {
  return sumAll(numbers)/numbers.length;
};

console.log(computeAverage([5, 10,
15]));   // Prints 10
```

TypeScript Generic Type Alias

In addition to the generic Array type, `Array<T>`, custom user-defined generic types are also supported by TypeScript. To define a generic type alias, use the `type` keyword followed by the alias name and angle brackets `<...>` containing a symbol for the generic type and assign it a custom definition. The symbol can be any alphanumeric character or string.

```
// This is a generic type alias
type Collection<G> = {
  name: string,
  quantity: number,
  content: G[]
};

let bookCollection: Collection<string>
= {
  name: 'Nursery Books',
  quantity: 3,
  content: ['Goodnight Moon', 'Humpty
Dumpty', 'Green Eggs & Ham']
};

let primeNumberCollection:
Collection<number> = {
  name: 'First 5 Prime Numbers',
  quantity: 5,
  content: [2, 3, 5, 7, 11]
};
```

TypeScript Generic Function Type Alias

With the TypeScript *generic function* type alias, a function can take parameters of generic types and return a generic type. To turn a function into a generic function type alias, add angle brackets, `<...>` containing a generic type symbol after the function name, and use the symbol to annotate the parameter type and return type where applicable.

```
// This is a generic function type alias
function findMiddleMember<M>(members:
M[]): M {
  return
members[Math.floor(members.length/2)];
}

// Call function for an array of strings
console.log(findMiddleMember<string>
(['I', 'am', 'very', 'happy'])); //
Prints "very"

// Call function for an array of numbers
console.log(findMiddleMember<number>
([210, 369, 102]));     // Prints 369
```

codecademy

Union Types

TypeScript Union Type

TypeScript allows a flexible type called any that can be assigned to a variable whose type is not specific. On the other hand, TypeScript allows you to combine specific types together as a union type.

```
let answer: any;    // any type
let typedAnswer: string | number; //
union type
```

TypeScript Union Type Syntax

TypeScript lets you create a union type that is a composite of selected types separated by a vertical bar, `|`.

```
let myBoolean: string | boolean;

myBoolean = 'TRUE';  // string type
myBoolean = false;   // boolean type
```

TypeScript Union Type Narrowing

Since a variable of a union type can assume one of several different types, you can help TypeScript infer the correct variable type using type narrowing. To narrow a variable to a specific type, implement a type guard. Use the `typeof` operator with the variable name and compare it with the type you expect for the variable.

```
const choices: [string, string] = ['NO',
'YES'];
const processAnswer = (answer: number
| boolean) => {
  if (typeof answer === 'number') {
    console.log(choices[answer]);
  } else if (typeof answer === 'boolean')
{
    if (answer) {
      console.log(choices[1]);
    } else {
      console.log(choices[0]);
    }
  }
}
processAnswer(true);   // Prints "YES"
processAnswer(0);      // Prints "NO"
```

TypeScript Function Return Union Type

TypeScript infers the return type of a function, hence, if a function returns more than one type of data, TypeScript will infer the return type to be a union of all the possible return types. If you wish to assign the function's return value to a variable, type the variable as a union of expected return types.

```
const popStack = (stack: any[]) => {
  if (stack.length) {
    return stack[stack.length-1]; //
return type is any
  } else {
    return null;      // return type is
null
  }
};
let toys: string[] = ['Doll', 'Ball',
'Marbles'];
let emptyBin: any[] = [];
let item: any | null = popStack(toys); //
item has union type
console.log(item);  // Prints "Marbles"
item = popStack(emptyBin);
console.log(item);  // Prints null
```

TypeScript Union of Array Types

TypeScript allows you to declare a union of an array of different types. Remember to enclose the union in parentheses, `(...)`, and append square brackets, `[]` after the closing parenthesis.

```
const removeDashes = (id: string
| number) => {
  if (typeof id === 'string') {
    id = id.split('-').join('');
    return parseInt(id);
  } else {
    return id;
  }
}
// This is a union of array types
let ids: (number | string)[] = ['93-235-
66', '89-528-92'];
let newIds: (number | string)[] = [];
for (let i=0; i < ids.length; i++) {
  newIds[i] = removeDashes(ids[i]); //
Convert string id to number id
}
console.log(newIds);  // Prints [9323566,
8952892]
```

TypeScript Union Type Common Property Access

As a result of supporting a union of multiple types, TypeScript allows you to access properties that are common among the member types without any error.

```
let element: string | number[]
= 'Codecademy';
// The .length property is common for
string and array
console.log(element.length);      //
Prints 10
// The .match method only works for
a string type
console.log(element.match('my')); //
Prints ["my"]

element = [3, 5, 1];
// The length property is common for
string and array
console.log(element.length);      //
Prints 3
// The .match method will not work for an
array type
console.log(element.match(5));  // Error:
Property 'match' does not exist on type
'number[]'.
```

TypeScript Union of Literal Types

You can declare a union type consisting of literal types, such as string literals, number literals or boolean literals. These will create union types that are more specific and have distinct states.

codecademy

```
// This is a union of string literal
types
type RPS = 'rock' | 'paper' | 'scissors'
;
const play = (choice: RPS): void => {
  console.log('You: ', choice);
  let result: string = '';
  switch (choice) {
    case 'rock':
      result = 'paper';
      break;
    case 'paper':
      result = 'scissors';
      break;
    case 'scissors':
      result = 'rock';
      break;
  }
  console.log('Me: ', result);
}
const number
= Math.floor(Math.random()*3);
let choices: [RPS, RPS, RPS] = ['rock',
'paper', 'scissors'];
play(choices[number]);
```

codecademy

Type Narrowing

TypeScript Union Type Narrowing

Since a variable of a union type can assume one of several different types, you can help TypeScript infer the correct variable type using type narrowing. To narrow a variable to a specific type, implement a type guard. Use the `typeof` operator with the variable name and compare it with the type you expect for the variable.

```
const choices: [string, string] = ['NO',
'YES'];
const processAnswer = (answer: number
| boolean) => {
  if (typeof answer === 'number') {
    console.log(choices[answer]);
  } else if (typeof answer === 'boolean')
{
    if (answer) {
      console.log(choices[1]);
    } else {
      console.log(choices[0]);
    }
  }
}
processAnswer(true);    // Prints "YES"
processAnswer(0);       // Prints "NO"
```

TypeScript Type Guard

A TypeScript type guard is a conditional statement that evaluates the type of a variable. It can be implemented with the `typeof` operator followed by the variable name and compare it with the type you expect for the variable.

```
// A type guard implemented with the
typeof operator
if (typeof age === 'number') {
  age.toFixed();
}
```

TypeScript Type Guard Accepted Types with typeof

The `typeof` operator may be used to implement a TypeScript type guard to evaluate the type of a variable including `number`, `string` and `boolean`.

TypeScript Type Guard with `in` operator

If a variable is a union type, TypeScript offers another form of type guard using the `in` operator to check if the variable has a particular property.

```
/*
In this example, 'swim' in pet uses the
'in' operator to check if the property
.swim is present on pet. TypeScript
recognizes this as a type guard and can
successfully type narrow this function
parameter.
*/
function move(pet: Fish | Bird) {
  if ('swim' in pet) {
    return pet.swim();
  }
  return pet.fly();
}
```

TypeScript Type Guard `if-else` Statement

If a variable is of a union type, TypeScript can narrow the type of a variable using a type guard. A type guard can be implemented as a conditional expression in an `if` statement. If an `else` statement accompanies the `if` statement, TypeScript will infer that the `else` block serves as the type guard for the remaining member type(s) of the union.

```
function roughAge(age: number | string) {
  if (typeof age === 'number') {
    // In this block, age is known to be
a number
    console.log(Math.round(age));
  } else {
    // In this block, age is known to be
a string
    console.log(age.split(".")[0]);
  }
}
roughAge('3.5');  // Prints "3"
roughAge(3.5);    // Prints 4
```

TypeScript Type Guard `if` Statement Function Return

If a variable is of a union type, TypeScript can narrow the type of a variable using a type guard. A type guard can be implemented as a conditional expression in an `if` statement. If the `if` block contains a `return` statement and is not followed by an `else` block, TypeScript will infer the rest of the code block outside the `if` statement block as a type guard for the remaining member type(s) of the union.

```
function formatAge(age: number | string)
{
  if (typeof age === 'number') {
    return age.toFixed(); // age must be
a number
  }
  return age; // age must not be a number
}
console.log(formatAge(3.5));    // Prints
"4"
console.log(formatAge('3.5'));  // Prints
"3.5"
```

codecademy

Advanced Object Types

TypeScript Interface Type

TypeScript allows you to specifically type an object using an interface that can be reused by multiple objects. To create an interface, use the `interface` keyword followed by the interface name and the typed object.

```
interface Publication {
  isbn: string;
  author: string;
  publisher: string;
}
```

TypeScript Interface Define Objects Only

In TypeScript, type aliases can define composite types such as objects and unions as well as primitive types such as numbers and strings; interface, however, can only define objects. Interface is useful in typing objects written for object-oriented programs.

```
// Type alias can define a union type
type ISBN = number | string;

// Type alias can define an object type
type PublicationT = {
  isbn: ISBN;
  author: string;
  publisher: string;
}

// Interface can only define an object
type
interface PublicationI {
  isbn: ISBN;
  author: string;
  publisher: string;
}
```

TypeScript Interface for Classes

To apply a TypeScript interface to a class, add the `implements` keyword after the class name followed by the interface name. TypeScript will check and ensure that the object actually implements all the properties and methods defined inside the interface.

```
interface Shape {
  area: number;
  computeArea: () => number;
}

const PI: number = 22/7 ;

// Circle class implements the Shape
interface
class Circle implements Shape {
  radius: number;
  area: number;
  constructor(radius: number) {
    this.radius = radius;
    this.area = this.computeArea();
  }
  computeArea = (): number => {
    return PI * this.radius
* this.radius;
  }
}

let target = new Circle(3);
console.log(target.area.toFixed(2));  //
Prints "28.29"
```

TypeScript Nested Interface

TypeScript allows both type aliases and interface to be nested. An object typed with a nested interface should have all its properties structured the same way as the interface definition.

```
// This is a nested interface
interface Course {
  description: {
    name: string;
    instructor: {
      name: string;
    }
    prerequisites: {
      courses: string[];
    }
  }
}

class myCourse implements Course {
  description = {
    name: '',
    instructor: {
      name: ''
    },
    prerequisites: {
      courses: []
    }
  }
}
```

TypeScript Nesting Interfaces Inside an Interface

Since interfaces are composable, TypeScript allows you to nest interfaces within an interface.

```
// Date is composed of primitive types
interface Date {
  month: number;
  day: number;
  year: number
}

// Passport is composed of primitive
types and nested with another interface
interface Passport {
  id: string;
  name: string;
  citizenship: string;
  expiration: Date;
}

// Ticket is composed of primitive types
and nested with another interface
interface Ticket {
  seat: string;
  expiration: Date;
}

// TravelDocument is nested with two
other interfaces
interface TravelDocument {
  passport: Passport;
  ticket: Ticket;
}
```

TypeScript Interface Inheritance

Like JavaScript classes, an interface can inherit properties and methods from another interface using the `extends` keyword. Members from the inherited interface are accessible in the new interface.

```
interface Brand {
  brand: string;
}

// Model inherits property from Brand
interface Model extends Brand {
  model: string;
}

// Car has a Model interface
class Car implements Model {
  brand;
  model;
  constructor(brand: string, model:
string) {
    this.brand = brand;
    this.model = model;
  }
  log() {
    console.log(`Drive a ${this.brand}
${this.model} today!`);
  }
}

const myCar: Car = new Car('Nissan',
'Sentra');
myCar.log(); // Prints "Drive a Nissan
Sentra today!"
```

TypeScript Interface Index Signature

Property names of an object are assumed to be strings, but they can also be numbers. If you don't know in advance the types of these property names, TypeScript allows you to use an index signature to specify the type for the property name inside an object. To specify an index signature, use square brackets, `[...]`, to surround the type notation for the property name.

```
interface Code {
  [code: number]: string;
}
const codeToStates: Code = {603: 'NH',
617: 'MA'};

interface ReverseCode {
  [code: string]: number;
}
const stateToCodes: ReverseCode = {'NH':
603, 'MA': 617};
```

TypeScript Interface Optional Properties

TypeScript allows you to specify optional properties inside an interface. This is useful in situations where not all object properties have values assigned to them. To indicate if a property is optional, append a `?` symbol after the property name before the colon, `:`.

```
interface Profile {
  name: string;
  age: number;
  hobbies?: string[];
}

// The property, hobbies, is optional, but name and age are required.
const teacher: Profile = {name: 'Tom Sawyer', age: 18};
```