Cheatsheets / **Learn TypeScript**

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

```
function modulo(dividend: number,
```

types match those of the tuple elements.

```
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

```
// This is a numeric enum type
enum ClassGrade {
  Freshman = 9,
```

assigned to them.

```
  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

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
```