TYPESCRIPT CHEAT SHEET

JAYSON LENNON

V1.01

HEEELLLOOOOO!

I'm Andrei Neagoie, Founder and Lead Instructor of the Zero To Mastery Academy.

After working as a Senior Software Developer over the years, I now dedicate 100% of my time to teaching others in-demand skills, help them break into the tech industry, and advance their careers.

In only a few years, **over 1,000,000 students** around the world have taken Zero To Mastery courses and many of them are now working at top tier companies like Apple, Google, Amazon, Tesla, IBM, Facebook, and Shopify, just to name a few.

This cheat sheet, created by our TypeScript instructor (Jayson Lennon) provides you with the key TypeScript information and concepts you need to know.

If you want to not only learn TypeScript but also get the exact steps to build your own projects and get hired as a Developer, then check out our Career Paths.

Happy Learning!
Andrei

[illegible]

Founder & Lead Instructor, Zero To Mastery
Andrei Neagoie

TypeScript Cheatsheet

Variables

A variable is a named memory location that can hold a value. Variables can be used to store a wide range of data types, such as numbers, strings, and arrays. A variable is declared by specifying its name, data type, and optionally an initial value. Once a variable is declared, it can be read and updated in other parts of the program.

```
// uninitialized variable
let name: string;
name = "Alice";

// initialized variable
let name2: string = "Bob";

// variable cannot be reassigned
const name3: string = "Carol";
name3 = "Dan"; // ERROR: cannot reassign name3
```

Type annotations for variables aren't required, and omitting them can help improve code readability:

```
// We know this is a number by looking at the value:
let count = 5;
```

There are also opposite situations where including the annotations help readability:

```
// We don't know the type of `value` here:
let result = someFunc();

// This makes things a bit clearer:
let result: QueryResult =  someFunc();
```

Regardless of whether you choose to use an annotation on a variable, TypeScript will analyze the code and figure out the actual type. So you'll get IDE completion and compiler support in both cases.

Functions

TypeScript return type annotations on functions are optional, however you should always include them. You can configure ESLint `.eslintrc.json` to emit an error if a function is missing a return type by setting this rule:

```
rules: {
    '@typescript-eslint/explicit-function-return-type': 'error',
}
```

Functions use the `function` keyword and can have any number of function parameters. Remember to *always* include a return type because this makes your code easier to work with:

```
//      name(param: type, param: type): returnType {}
function sum(lhs: number, rhs: number): number {
  return lhs + rhs;
}
```

Function expressions can be assigned to variables and they come in two forms: one uses the `function` keyword, and the other uses arrow syntax:

```
// using `function`
const sum = function sum(lhs: number, rhs: number): number {
  return lhs + rhs;
}
// using arrow
const sum = (lhs: number, rhs: number): number => {
  return lhs + rhs;
}
```

Arithmetic

TypeScript provides basic arithmetic to perform calculations.

```
const add = 1 + 1; // 2
const subtract = 2 - 1; // 1
const multiply = 3 * 3; // 9
const divide = 8 / 2; // 4
```

```
// You won't get a divide by zero error.
const infinity = 1 / 0; // Infinity
const notANumber = 0 / 0; // NaN

const remainder = 10 % 3; // 1
const negate = -remainder; // -1
const exponent = 2 ** 3; // 8
```

Arrays

Arrays offer a way to store and manipulate collections of values of the same type. They are defined using square brackets and can be populated with values at initialization, or later using various methods such as `push()`, `splice()`, and `concat()`. Arrays can be of a fixed length or dynamically resized as needed, and they can be used with various array methods to perform common operations like sorting, filtering, and mapping.

```
// create an array
const numbers: number[] = [1, 2, 3];
let letters: string[] = ["a", "b", "c"];

// iterate over an array
for (let n of numbers) {
  // ...
}

// array method examples
numbers.pop();  // remove last item
const doubled = numbers.map(n => n * 2);  // double each number
```

Tuples

Tuples provide a way to express an array with a fixed number of elements of different types, creating a data structure with multiple different types. They can be especially handy when dealing with scenarios such as representing coordinates, storing key-value pairs, or returning multiple values from a function. Since they are type-checked, TypeScript can ensure that the values in the tuple are correct at compile time.

```
// type aliases for clarity
type Title = string;
type PublishYear = number;
```

```
// declare a tuple type
let book: [Title, PublishYear];

// initialize a tuple
book = ["sample", 1980];

// return a tuple from a function
function newBook(): [Title, PublishYear] {
  return ["test", 1999];
}

// destructure a tuple into two variables
const [title, published] = newBook();
//    "test", 1999
```

Control Flow

Control flow allows programmers to control the flow of their code based on certain conditions or events. Control flow structures include conditional statements, loops, and function calls, which allow for branching and repetition of code.

`if..else`

Using an `if..else` statement allows you to make a choice based on the result of a boolean expression:

```
const age = 20;
if (age === 20) {
  // execute when true
}

if (age === 20) {
  // execute when true
} else {
  // execute when false
}

if (age === 20) {
  // execute when true
} else if (age < 20) {
  // execute when age is less than 20
} else {
  // execute when age is greater than 20
}
```

`switch`

The `switch` statement executes different code based on a list of cases. This is useful when there are many possibilities to choose from. A case will "fall through" if you omit `break`, which executes statements from the matching case onwards. This is sometimes desired, but of the time you'll want to include `break` so only the matching case gets executed:

```
const n = 3;
switch (n) {
  case 1:
    // execute when n is 1
    break;
  case 2:
    // execute when n is 2
    break;
  case 3:
    // execute when n is 3
    // FALL THROUGH
  case 4:
    // execute when n is 3 or 4
    break;
  default:
    // execute for all other numbers
}
```

Repetition

Repetition structures allow you to execute code repeatedly.

C-style `for` loop

The `for` loop is implemented as a C-style `for` loop which requires three expressions:

1. loop counter initialization
2. loop condition
3. what to do at the end of each iteration

```
//  (init; execute until; at end)
for (let i = 1; i <= 5; i++) {
  // template string literal
  console.log(`${i}`);
}
```

`while` loop

A `while` loop executes the body 'while' some boolean expression is `true`. It is your responsibility to manage when and how the loop exits.

This `while` loop has the same output as the previous C-style loop shown above:

```
let i = 1; // create loop counter
while (i <= 5) {
  console.log(`${i}`);
  i += 1; // increment counter
}
```

Iteration: `for..of` loop

Iterators offer a way to traverse the elements of a collection one by one. The purpose of iterators is to provide a standard way for accessing and iterating over collections, such as arrays or maps in a language-agnostic way. Using iterators, you can iterate over collections in a loop without having to worry about the underlying implementation of the collection.

```
const abc = ["a", "b", "c"];

// Iterate through an array using a standard `for` loop:
for (let i = 0; i < abc.length; i++) {
  console.log(abc[i]); // 'a' 'b' 'c'
}

// Iterate through an array using a `for..of` loop:
for (const letter of abc) {
  console.log(letter); // 'a' 'b' 'c'
}
```

Classes

Classes are a way to define blueprints for objects. They encapsulate data and behavior and can be used to create instances of objects with predefined properties and methods.

Classes can be extended and inherited, allowing for the creation of complex object hierarchies.

```
class Dimension {
  // Private properties can only be accessed
  // by this class:
  private length: number;
  // Protected properties can be accessed by
  // this class and any subclasses:
  protected width: number;
  // Public properties can be accessed by anything:
  public height: number;

  // We can set the initial values in the constructor:
  constructor(l: number, w: number, h: number) {
    this.length = l;
    this.width = w;
    this.height = h;
  }

  // class method
  getLength(): number {
    return this.length;
  }
}

const box = new Dimension(3, 2, 1);
// call method:
box.getLength(); // 3
```

You can also use shorthand constructor to avoid some boilerplate. Here is the same class as above, but using a shorthand constructor instead:

```
class Dimension {
  constructor(
    private length: number,
    protected width: number,
    public height: number,
  ) {}
  getLength(): number {
    return this.length;
  }
}
```

Interfaces

Interfaces provide a way to define the shape of objects or classes. They define the contracts that objects must follow, specifying the properties and methods that an object must have. Interfaces make it easier to write type-safe code by providing a way to ensure that objects are of the correct shape before they are used in a program. They also allow for code to be more modular and reusable, since objects can be easily swapped out as long as they adhere to the interface's contract.

```
interface Area {
  area(): number;
}

interface Perimeter {
  perimeter(): number;
}

// OK to implement more than one interface
class Rectangle implements Area, Perimeter {
  constructor(
    public length: number,
    public width: number,
  ) {}

  // We must provide an `area` function:
  area(): number {
    return this.length * this.width;
  }

  // We must provide a `perimeter` function:
  perimeter(): number {
    return 2 * (this.length + this.width);
  }
```

Interfaces can be combined to help with ergonomics:

```
type AreaAndPerimeter = Area & Perimeter;

class Circle implements AreaAndPerimeter {
  constructor(
    public radius: number,
  ) {}

  area(): number {
    return Math.PI * this.radius ** 2;
  }

  perimeter(): number {
    return 2 * Math.PI * this.radius;
  }
}
```

Maps

A `Map` is a data structure that allows you to store data in a key-value pair format. Keys in a map must be unique, and each key can map to only one value. You can use any type of value as the key, including objects and functions, but strings and numbers are recommended to keep the map easy to work with. Maps are useful when you want to quickly access data and you are able to maintain the key in memory. In situations where you don't have a key for the data you need, a different data structure is more appropriate.

```
type Name = string;
type Score = number;

// make a new map
const testScores: Map<Name, Score> = new Map();

// insert new pair
testScores.set("Alice", 96);
testScores.set("Bob", 88);
// remove pair based on key
testScores.delete("Bob");

// iterate over pairs
for (const [name, score] of testScores) {
  console.log(`${name} score is ${score}`);
}

// empty the map
testScores.clear();
```

Exceptions

Exceptions are a way to handle errors and unexpected behavior in your code. When an exception occurs, it interrupts the normal flow of the program and jumps to a predefined error handling routine. Exceptions can be used to catch and handle errors in a way that doesn't crash the program or cause unexpected behavior. Exceptions are thrown using the `throw` keyword and caught using the `try...catch` statement.

```
function divide(lhs: number, rhs: number): number {
  if (rhs === 0) {
    // use `throw` to generate an error
    throw new Error("unable to divide by zero");
  }
  return lhs / rhs;
}

try {
  // try to run this code
  const num = divide(10, 0);
} catch (e) {
  // run this code if the above fails
  console.log(`an error occurred: ${e}`);
  console.log("try a different number next time");
} finally {
  // run this code no matter what
  console.log("this block will execute no matter what");
}
```

Unions

Union types allows you to declare a variable or parameter that can hold multiple types of value and are declared using the pipe symbol (`|`) between the types. Union types can be useful when you want something to accept multiple types of input.

```
// make a new union type
type Color = "red" | "green" | "blue";

// only "red" "green" or "blue" is allowed here:
const r: Color = "red";

const r: Color = "yellow";  // ERROR: "yellow" not in type Color

function setBgColor(c: Color) {
  // type guard
  switch (c) {
    case "red":
      break;
    case "green":
      break;
    case "blue":
      break;
  }
}
```

Type Predicates

Type predicates offer a way to determine the type of data based on a condition. This is achieved by defining a function that takes a some data as an argument, applies type guards, and returns a boolean indicating whether the data is a specific type. The function is then used to narrow down the type in subsequent code. Type predicates are useful when dealing with union types or other situations where the type of a variable may not be known at compile-time. Type predicates allow the type to be determined correctly which avoids runtime errors.

```
interface Square {
  kind: "square";
  size: number;
}

interface Circle {
  kind: "circle";
  radius: number;
}

type Shape = Square | Circle;

// type predicate function
function isSquare(shape: Shape): shape is Square {
  return shape.kind === "square";
}

// type predicate function
function isCircle(shape: Shape): shape is Circle {
  return "radius" in shape;
}

function calculateArea(shape: Shape): number {
  // use type predicate
  if (isSquare(shape)) {
    return shape.size ** 2;
  }
  // use type predicate
  if (isCircle(shape)) {
    return Math.PI * shape.radius ** 2;
  }
  throw "unknown shape";
}
```

Optional Chaining

Optional properties are convenient because they allow situations where it may not be appropriate to have data present. However, they make it cumbersome to access any additional data that is behind the optional properties. For example, trying to access multiple optional objects one after the other requires multiple checks for `undefined` and multiple `if` blocks.

```
type Pii = {
  age?: number;
  address?: string;
};

type SearchResult = {
  name: string;
  pii?: Pii;
};

// use question marks for optional chaining
if (result?.pii?.age) {
  console.log(`${result.name} age is ${result.pii.age}`);
}
```

Utility Types

TypeScript provides handy utility types which are used to create new types from existing types, which can greatly reduce the amount of code to write when working with complex types.

```
interface UserForm {
  email: string;
  password: string;
  passwordConfirmation: string;
  phoneNumber?: string;
  address?: string;
  agreedToMarketingEmails: boolean;
}

// Use _only_ the properties listed:
```

```
type LoginForm = Pick<UserForm, "email" | "password" | "passwordConfirmation">;

// Use all properties _except_ the ones listed:
type LoginForm2 = Omit<UserForm, "phoneNumber" | "address" | "agreedToMarketingEmails">;

// Make all properties mandatory by removing
// the question marks from the properties.
type SignupForm = Required<UserForm>;

// Cannot reassign any properties:
type SubmittedForm = Readonly<UserForm>;
```

Async/Await

`async/await` allows you to write asynchronous code in a synchronous way. The `async` keyword used with a function or closure creates an asynchronous context. The `await` keyword can be used inside an asynchronous context to wait for a `Promise` to resolve before moving on to the next line of code. While waiting, other code outside the function will execute. When the promise resolves, the value is returned and assigned to the variable on the left side of the `=` sign. This makes it easier to work with asynchronous code as you can write code in a more sequential way.

```
// `async` keyword marks an asynchronous function
async function fetchUserDataAsync(userId: number): Promise<{ name: string }> {
  // use `await` to wait for the response
  const response = await fetch(
    `https://jsonplaceholder.typicode.com/users/${userId}`
  );
  const data = await response.json();
  return { name: data.name };
}

// create a new asynchronous context
(async () => {
  try {
    const userData = await fetchUserDataAsync(1);
    console.log(userData.name);
  } catch (error) {
    console.error(error);
  }
})();

// this is the same call as above, but using
// the Promise API instead of async/await
fetchUserDataAsync(1)
```

```
  .then((userData) => console.log(userData.name))
  .catch((error) => console.error(error));
```

Generic Functions

Generic functions are functions that are designed to work with different types of data. They allow you to create a function that can be used with various types of data without having to write a separate function for each type. This makes your code more efficient, reusable, and easier to maintain. Generic functions are especially useful when working with collections of data, such as arrays, because they allow you to create a function that can work with any type of data in the collection.

```
// generic type T will be replaced with whatever
// type is used with the function
function getFirst<T>(arr: T[]): T | undefined {
  if (arr.length > 0) {
    return arr[0];
  }
  return undefined;
}

const nums = [1, 2, 3];
const one = getFirst(nums);  // T is `number`

const letters = ["a", "b", "c"];
const a = getFirst(letters);  // T is `string`
assert.equal(a, "a");

const objects = [{ first: 1 }, { second: 2 }];
const first = getFirst(objects);  // T is `object`
```

Generic Classes

Generic classes offer the ability to define a class that can work with a variety of different data types. By using generic type parameters, you can create a single class that can be customized to work with any type of data that you need.

Similarly to generic functions, generic classes allow you to write more flexible and reusable code, since you don't have to create a separate class for every possible data type.

```
class Stack<T> {
  private elements: T[] = [];

  public push(element: T): void {
    this.elements.push(element);
  }

  public pop(): T | undefined {
    return this.elements.pop();
  }

  public peek(): T | undefined {
    return this.elements[this.elements.length - 1];
  }

  public isEmpty(): boolean {
    return this.elements.length === 0;
  }
}

// always specify the type when instantiating a class
const strings = new Stack<string>();
strings.push("hello");
strings.push(1);    // ERROR: stack only supports `string`

const nums = new Stack<number>();
const bools = new Stack<boolean>();
```