ArkTS Tutorial

version 0.7.0

Huawei Device BG

2023.06.16

Contents

ArkTS Tutorial

Introduction

Welcome to the tutorial for ArkTS, a TypeScript-based programming language designed specifically to build high-performance mobile applications!

ArkTS is optimized to provide better performance and efficiency, while still maintaining the familiar syntax of TypeScript.

As mobile devices continue to become more prevalent in our daily lives, there is a growing need for programming languages optimized for the mobile environment. Many current programming languages were not designed with mobile devices in mind, resulting in slow and inefficient applications that drain battery life. ArkTS has been specifically designed to address such concerns by prioritizing higher execution efficiency.

ArkTS is based on the popular programming language TypeScript that extends JavaScript by adding type definitions. TypeScript is well-loved by many developers as it provides a more structured approach to coding in JavaScript. ArkTS aims to keep the look and feel of TypeScript to enable a seamless transition for the existing TypeScript developers, and to let mobile developers learn ArkTS quickly.

One of the key features of ArkTS is its focus on low runtime overhead. ArkTS imposes stricter limitations on the TypeScript's dynamically typed features, reducing runtime overhead and allowing faster execution. By eliminating the dynamically typed features from the language, ArkTS code can be compiled ahead-of-time more efficiently, resulting in faster application startup and lower power consumption.

Interoperability with JavaScript was a critical consideration in the ArkTS language design. Many mobile app developers already have TypeScript and JavaScript code and libraries they would want to reuse. ArkTS has been designed for seamless JavaScript interoperability, making it easy for the developers to integrate the JavaScript code into their applications and vice versa. This will allow the developers to use their existing codebases and libraries to leverage the power of our new language.

This tutorial will guide the developers through the core features, syntax, and best practices of ArkTS. After reading this tutorial through the end, the developers will be able to build performant and efficient mobile applications in ArkTS.

The Basics

Declarations

Declarations in ArkTS introduce variables, constants, functions and types.

Variable Declaration

A variable declaration introduces a reassignable variable.

```
let hi: string = "hello"
hi = "hello, world"
```

Constant Declaration

A constant declaration introduces a read-only constant that can be assigned a value only once.

```
const hello: string = "hello"
```

Assigning a new value to a constant is an error.

Automatic Type Inference

As ArkTS is a statically typed language, the types of all entities, like variables and constants have to be known at compile time.

However, developers do not need to explicitly specify the type of a declared entity if a variable or a constant declaration contains an initial value. All cases that allow the type to be inferred automatically are specified in the ArkTS Specification.

Both variable declarations are valid, and both variables are of the `string` type:

```
let hi1: string = "hello"
let hi2 = "hello, world"
```

Types

`Class`, `interface`, `function`, `nullable` and `enum` types are described in the corresponding sections.

Number and Other Numeric Types

ArkTS has both `number` type and a set of specific numeric types:

- signed integer types: `Byte`, `Short`, `Int`, `Long`
- unsigned integer types: `UByte`, `UShort`, `UInt`, `ULong`
- floating-point types: `Float`, `Double`
- a `Char` type, that is equal to `UShort`

The `number` type equals to `Double`, any integer and floating-point value can be assigned to a variable of this type.

There is also a set of so-called *primitive types* (predefined value types): `byte`, `short`, `int`, `long`, `ubyte`, `ushort`, `uint`, `ulong`, `float`, `double`, `char`, that can be used for better application performance and for interoperability with low-level languages.

The use of primitive types is restricted: such types cannot be used as generic type arguments.

Numeric literals include integer literals and floating-point literals with the decimal base.

Integer literals include the following:

- decimal integers that consist of a sequence of digits. For example: `0`, `117`, `-345`;
- hexadecimal integers that start with 0x (or 0X) and can contain digits (0-9) and letters a~f or A~F. For example: `0x1123`, `0x00111`, `-0xF1A7`;
- octal integers that start with 0o (or 0O) and can only contain digits (0-7). For example: `0o777`;
- binary integers that start with 0b (or 0B) and can only contain the digits 0 and 1. For example: `0b11`, `0b0011`, `-0b11`.

A floating-point literal includes the following:

- decimal integer, optionally signed (i.e., prefixed with “+” or “-“);
- decimal point (“.”);
- fractional part (represented by a string of decimal digits);
- exponent part that starts with “e” or “E”, followed by an optionally signed (i.e., prefixed with “+” or “-“) integer.

For example:

```
3.14
3.141592
.5
1e10
```

Using the `number` type:

```
function f(n: number) : number {
    if (n <= 1) return 1
    return n * f(n - 1)
}
```

Using the `long` type for better performance:

```
function f(n: long) : long {
    if (n <= 1) return 1
    return n * f(n - 1)
}
```

Boolean

The `boolean` type represents logical values that are either `true` or `false`.

Usually variables of this type are used in conditional statements:

```
let isDone: boolean = false
...

if (isDone) {
    console.log ("Done!")
}
```

String

A `string` is a sequence of characters; some characters can be set using escape-sequences.

A `string` literal consists of zero or more characters enclosed in double quotation marks (“).

```
"Hello, world!\n"
```

Void Type

The `void` type is used to specify that a function does not return a value. This type has the only one value, which is also `void`. As `void` is reference type, it can be used as type argument for generic types.

Object Type

An `Object` class type is a base type for all reference types. Any value, except values of primitive types, can be directly assigned to variables of type `Object`. For details see *Standard Library Reference*.

Array Type

An `array` is an object comprised of elements of data types assignable to the element type specified in the array declaration. A value of an `array` is set using *array composite literal*, that is a list of zero or more expressions enclosed in square brackets ([]). Each expression represents an element of the `array`. The length of the `array` is set by the number of expressions.

The following example creates the `array` with three elements:

```
let names: string[] = ["Alice", "Bob", "Carol"]
```

Enum Type

An `enum` type is a value type with a defined set of named values called enum constants. To be used, an `enum` constant must be prefixed with an enum `type` name.

```
enum Color { Red, Green, Blue }
let c: Color = Color.Red
```

Alias Type

Alias types provide names for anonymous types (array, function or nullable types) or alternative names for existing types.

```
type Matrix = double[][]
type Handler = (s: string, no: int): string
type Predicate <T> = (x: T): Boolean
```

Operators

Assignment Operators

Simple assignment operator '=' is used as in "x = y".

Compound assignment operators combine an assignment with an operator, where `x op = y` equals to `x = x op y`.

Compound assignment operators are as follows: `+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `>>>=`, `&=`, `|=`, `^=`.

Comparison Operators

Operator	Description
==	returns true if both operands are equal
!=	returns true if both operands are not equal
>	returns true if the left operand is greater than the right
>=	returns true if the left operand is greater than or equal to the right
<	returns true if the left operand is less then the right
<=	returns true if the left operand is less than or equal to the right

Arithmetic Operators

Unary operators are `-`, `+`, `--`, `++`.

Binary operators are as follows:

Operator	Description
+	addition
-	subtraction
*	multiplication
/	division
%	remainder after division

Bitwise Operators

Operator	Description
a & b	Bitwise AND: sets each bit to 1 if the corresponding bits of both operands are 1, otherwise to 0.
a | b	Bitwise OR: sets each bit to 1 if at least one of the corresponding bits of both operands is 1, otherwise to 0.
a ^ b	Bitwise XOR: sets each bit to 1 if the corresponding bits of both operands are different, otherwise to 0.

`~ a`	Bitwise NOT: inverts the bits of the operand.
`a << b`	Shift left: shifts the binary representation of *a* to the left by *b* bits.
`a >> b`	Arithmetic right shift: shifts the binary representation of *a* to the right by *b* bits with sign-extension.
`a >>> b`	Arithmetic right shift: shifts the binary representation of *a* to the right by *b* bits with zero-extension.

Logical Operators

Operator	Description
`a && b`	logical AND
`a || b`	logical OR
`! a`	logical NOT

Control Flow

`If` Statements

Use `if` statement to execute a sequence of statements when a logical condition is true, or to choose one of two statement sequences to execute.

The `if` statement looks as follows:

```
if (condition1) {
    statements1
} else if (condition2) {
    statements2
} else {
    else_statements
}
```

All conditional expressions must be of the type `boolean`.

`Switch` Statements

Use `switch` statement to execute a sequence of statements that match the value of the switch expression.

The `switch` statement looks as follows:

```
switch (expression) {
case label1:
    statements1
    [break;]
case label2:
case label3:
    statements23
    [break;]
default:
    default_statements
}
```

The `switch` expression type must be of `integer`, `enum` or `string` type.

Each label must be either a constant expression or the name of an enum constant.

If the value of a `switch` expression equals the value of some label, then the corresponding statements are executed.

If there is no match, and the `switch` has the default clause, then the default statements are executed.

An optional `break` statement allows to break out of the `switch` and continue executing the statement that follows the `switch`.

If there is no break, then the next statements in the `switch` is executed.

Conditional Expressions

The conditional expression `? :` uses the `boolean` value of the first expression to decide which of two other expressions to evaluate.

The `switch` statement looks as follows:

```
condition ? expression1 : expression2
```

The condition must be a logical expression. If that logical expression is `true`, then the first expression is used as the result of the ternary expression; otherwise, the second expression is used.

Example:

```
let message = isValid ? 'Valid' : 'Failed'
```

`For` Statements

A `for` statement is executed repeatedly until the specified loop condition is `false`. A for statement looks as follows:

```
for ([init]; [condition]; [update]) {
    statements
}
```

When a `for` statement is executed, the following process takes place:

#. An `init` expression is executed, if any. This expression usually initializes one or more loop counters. #. The condition is evaluated. If the value of condition is `true`, or if the conditional expression is omitted, then the statements in the for body are to be executed. If the value of condition is `false`, the for loop terminates. #. The statements of the `for` body are executed. #. If there is an `update` expression, the `update` expression is executed. #. Go back to step 2.

Example:

```
for (let i = 0; i < 10; i+=2) {
    sum += i
}
```

`For-of` Statements

Use `for-of` statements to iterate over an array or string. A `for-of` statement looks as follows:

```
for (forVar of expression) {
    statements
}
```

Example:

```
for (let ch of "a string object") { /*process ch*/ }
```

`While` Statements

A `while` statement has its block executed as long as the specified condition evaluates to `true`. It looks as follows:

```
while (condition W) {
    statements
}
```

The condition must be a logical expression.

Example:

```
let n = 0;
let x = 0;
while ( n < 3 ) {
    n++;
    x += n;
}
```

Do-while Statements

The do-while statement is executed repetitively until the specified condition evaluates to false. It looks as follows:

```
do {
    statements
} while (condition)
```

The condition must be a logical expression.

Example:

```
let i = 0;
do {
    i += 1
} while (i < 10)
```

Break Statements

Use a break statement to terminate a loop statement or switch.

Example:

```
let x = 0;
while (true) {
    x++;
    if (x > 5) {
        break;
    }
}
```

A break statement with a label identifier transfers control out of the enclosing statement, which has the same label identifier.

Example:

```
label: while (true) {

    switch (x) {

    case 1:
        statements;
        break label // breaks the while
    }
}
```

Continue Statements

A continue statement stops the execution of the current loop iteration and passes control to the next iteration.

Example:

```
for (x = 0 ; x < 100 ; x ++) {
    if (x % 2 == 0) {
        continue;
    }
    sum += x
}
```

`Throw` and `Try` Statements

A `throw` statement is used to throw an exception or error:

```
throw new Error("this error")
```

A `try` statement is used to catch and handle an exception or error:

```
try {
    // try block
} catch (e) {
    // handle the situation
}
```

The example below shows the `throw` and `try` statements used to handle the zero division case:

```
class ZeroDivisor extends Error {}

function divide (a: number, b: number): number{
    if (b == 0) throw new ZeroDivisor()
    return a / b
}

function process (a: number, b: number) {
    try {
        let res = divide(a, b)
        console.log (res)
    } catch (x) {
        console.log ("some error")
    }
}
```

Functions

Function Declarations

A function declaration introduces a named function, specifying its name, parameters, return type and body.

Below is a simple function with two string parameters and string return type:

```
function add(x: string, y: string): string {
    let z : string = x + " " + y
    return z
}
```

Functions with No Return Type

The return type of a function that does not need to return a value may be explicitly specified as `void` or omitted altogether. No return statement is needed for such functions.

Both notations below are valid:

```
function hi1() { console.log("hi") }
function hi2(): void { console.log("hi") }
```

Function Scope

Variables and other entities defined in a function are local to the function and cannot be accessed outside.

If the name of a variable defined in the function is equal to the name of an entity in the outer scope, then the local definition shadows outer entity.

Function Calls

Calling a function actually performs the actions along the given parameters.

If the function is defined as follows:

```
function join(x :string, y :string) :string {
    let z: string = x + " " + y
    return z
}
```

then it is called with two arguments of the type `string`:

```
let x = join("hello", "world")
console.log(x)
```

Function Types

Function types are commonly used as follows to define callbacks:

```
type trigFunc = (x: double) => double

function do_action(f: trigFunc) {
     f(3.141592653589)  // call the function
}

do_action(sin) // pass the function as the argument
```

Arrow Functions or Lambdas

A function can be defined as an arrow function, for example:

```
let sum = (x: number, y: number): number => {
    return x + y
}
```

Currently ArkTS does not allow omitting parameter types in arrow functions. A return type can be omitted, in such case a return type is implied to be `void`.

Closure

An arrow function is usually defined inside another function. As an inner function, it can access all variables and functions defined in the outer functions.

To capture the context, an inner function forms a closure of its environment. The closure allows to access such inner function outside its own environment.

```
function f(): () => number {
    let count = 0
    return (): number => { count++; return count }
```

```
}

let z = f()
console.log(z()) // output: 1
console.log(z()) // output: 2
```

In the sample above, the arrow function closure captures the `count` variable.

Function Overloading

The function name is said to be overloaded if two functions declared in the same declaration space have the same name but a different parameter list.

When calling an overloaded function name, ArkTS chooses the most appropriate function to call:

```
function log(x: string) {}
function log(x: number) {}

log("abc") // 1st function is called
log(1) // 2nd function is called
```

Parameter lists are different if they contain different number of parameters, or at least one pair of parameters have distinguishable types.

It is an error if two overloaded functions have the same name and identical parameter lists.

Classes

A class declaration introduces a new type and defines its fields, methods and constructors.

In the following example, a Person class is defined, which has fields ‘name’, ‘surname’, a constructor function and a method `fullName`:

```
class Person {
    name: string = ""
    surname: string = ""
    constructor (n: string, sn: string) {
        this.name = n
        this.surname = sn
    }
    fullName(): string {
        return this.name + " " + this.surname
    }
}
```

After the class is defined, its instances can be created using the keyword `new`:

```
let p = new Person("John", "Smith")
console.log(p.fullName())
```

Constructors

A class declaration may contain several constructors that are used to initialize object state.

Constructor is defined as follows:

```
constructor ([parameters]) {...}
```

If there is more than one constructor, all their respective parameter lists must be different, for example:

```
constructor (n: string, sn: string) {
    this.name = n
    this.surname = sn
}
constructor (n: string) {
    this.name = n
    this.surname = ""
}
```

If no constructor is defined, then a default constructor with an empty parameter list is created automatically, for example:

```
class Point {
    x: double
    y: double
}
let p = new Point()
```

In this case the default constructor fills the instance fields with default values for the field types.

Fields

A field is a variable of some type that is declared directly in a class. A classes may have instance fields, static fields or both.

Instance Fields

Instance fields exist on every instance of a class. Each instance has its own set of instance fields.

```
class Person {
    name: string = ""
    age: int = 0
    constructor(n: string, a: int) {
        this.name = n
        this.age = a
    }
}

let p1 = new Person("Alice", 25)
let p2 = new Person("Bob", 28)
```

To access an instance field, use an instance of the class:

```
p1.name
this.name
```

Static Fields

Use the keyword `static` to declare a field as static. Static fields belong to the class itself, and all instances of the class share one static field.

To access a static field, use the class name:

```
class Person {
    static numberOfPersons = 0
    constructor() {
       ...
       Person.numberOfPersons++
       ...
    }
}

console.log(Person.numberOfPersons)
```

Getters and Setters

Setters and getters can be used to provide controlled access to object properties.

In the following example, a setter is used to forbid setting invalid values of the 'age' property:

```
class Person {
    name: string = ""
    private _age: int = 0
    get age(): int { return _age }
    set age(x: int) {
        if (x < 0) { throw Error("Invalid age argument") }
                    this._age = x
    }
}

let p = new Person ()
console.log (p.age) // 0 will be printed out
p.age = -666 // Error will be thrown as an attempt to set incorrect age
```

A class may define a getter, a setter or both.

Methods

A method is a function that belongs to a class. A class can define instance methods, static methods or both. A static method belongs to the class itself, and can have access to static fields only. While instance method has access to both static (class) fields and instance fields including private ones of its class.

Instance Methods

Example below illustrates how instanced methods work. The `calculateArea` method calculates the area of a rectangle by multiplying the height by the width:

```
class Rectangle {
    private height: number;
    private width: number;
    constructor(height: number, width: number) { ... }
    calculateArea(): number {
        return this.height * this.width;
    }
}
```

To use an instance method, call it on an instance of the class:

```
let r = new Rectangle(10, 10);
console.log(square.calculateArea()); // output: 100
```

Static Methods

Use the keyword `static` to declare a method as static. Static methods belong to the class itself and have access to static fields only. A static method defines a common behaviour of the class as a whole. All instances have access to static methods.

To call a static method, use the class name:

```
class Cl {
    static staticMethod(): string {
        return "this is a static method.";
    }
}
Cl.staticMethod();
```

Inheritance

A class can extend another class (called base class) and implement several interfaces using the following syntax:

```
class [extends BaseClassName] [implements listOfInterfaces] { ... }
```

The extended class inherits fields and methods from the base class, but not constructors, and can add its own fields and methods as well as override methods defined by the base class.

The base class is also called ‘parent class’ or ‘superclass’. The extended class also called ‘derived class’ or ‘subclass’.

Example:

```
class Person {
    name: string = ""
    get age(): int
}
class Employee extends Person {
    salary: number
    calculateTaxes(): number {}
}
```

A class containing the `implements` clause must implement all methods defined in all listed interfaces, except the methods defined with default implementation.

```
interface DateInterface {
    now(): string;
}
class Date implements DateInterface {
    now(): string {
        // implementation is here
    }
}
```

Override Methods

A subclass may override implementation of a method defined in its superclass. An overriden method must be marked with the keyword `override`. An overriden method must have the same types of parameters and same or derived return type as the original method.

```
class Rectangle {
    ...
```

```
    area(): number {
        // implementation
    }
}
class Square extends Rectangle {
    private side: number = 0
    override area(): number {
        return this.side*this.side
    }
}
```

Access to Super

The keyword `super` can be used to access instance fields, instance methods and constructors from the super class.

It is often used to extend basic functionality of subclass with the required behaviour taken fronm the super class:

```
class Rectangle {
    protected height: number = 0
    protected width: number = 0

    constructor (h: number, w: number) {
                this.height = h
                this.width = w
    }

    draw() {
                    /* draw bounds */

    }
}
class FilledRectangle extends Rectangle {
            color = ""
            constructor (h: number, w: number, c: string) {
                 super(h, w) // call of super constructor
                 this.color = c
            }

    override draw() {
         super.draw() // call of super methods
         // super.height - can be used here
                    /* fill rectangle */
    }
}
```

Constructors in Derived Class

The first statement of a constructor body can use the keyword `super` to explicitly call a constructor of the direct superclass.

```
class Rectangle {
    constructor(width: number, height: number) { ... }
}
class Square extends Rectangle {
    constructor(side: number) {
        super(side, side)
```

```
    }
}
```

If a constructor body does not begin with such an explicit call of a superclass constructor, then the constructor body implicitly begins with a superclass constructor call `super()`.

Visibility Modifiers

Both methods and properties of a class can have visibility modifiers.

There are several visibility modifiers: `private`, `protected`, `public`, and `internal`. The default visibility is `public`. `internal` allows to limit visibilty within the current package.

Public Visibility

The `public` members (fields, methods, constructors) of a class are visible in any part of the program, where their class is visible.

Private Visibility

A `private` member cannot be accessed outside the class in which it is declared, for example:

```
class C {
    public x: string = ""
    private y: string = ""
    set_y (new_y: string) {
        y = new_y // ok, as y is accessible within the class itself
    }
}
let c = C()
c.x = "a" // ok, the field is public
c.y = "b" // compile-time error, 'y' is not visible
```

Protected Visibility

The `protected` modifier acts much like the `private` modifier, but `protected` members are also accessible in derived classes, for example:

```
class Base {
    protected x: string = ""
    private y: string = ""
}
class Derived extends Base {
    foo() {
        this.x = "a" // ok, access to protected member
        this.y = "b" //  compile-time error, 'y' is not visible, as it is private
    }
}
```

Interfaces

An interface declaration introduces a new type. Interfaces are a common way of defining contracts between various part of codes.

Interfaces are used to write polymorphic code, which can be applied to any class instances that implement a particular interface.

An interface usually contains properties and method headers. Some interface methods may have default implementation.

Examples:

```
interface Style {
    color: string // property
}
interface Area {
    calculateArea(): number // method header
            someMethod() { ... some actions ... }
    // method with default implementation
}
```

Examples of a class implementing interface:

```
class Rectangle implements Area {
    ...
    calculateArea(): number {
        someMethod() // calls another method and returns result
        return this.width * this.height
    }
}
```

Interface Properties

An interface property looks like a field, getter, setter or both getter and setter.

A property in the form of a field is just a shortcut notation of a getter/setter pair, and the following notations are equal:

```
interface Style {
    color: string
}
interface Style {
    get color(): string
    set color(x: string)
}
```

A class that implements an interface may also use a short or a long notation:

```
class StyledRectangle implements Style {
    color: string
}
```

The short notation implicitly defines a private field and getter and setter:

```
class StyledRectangle implements Style {
    _color: string
    get color(): string { return this._color }
    set color(x: string) { this._color = x }
}
```

Interface Methods

An interface method is either a method header (no method body is present) or a method with default implementation.

```
interface I {
    foo(): number // method header
    goo(): number { return 0 } // method with default implementation
}
```

A method with default implementation does not require implementation in a class that implements the interface but it can be provided if necessary.

Interface Inheritance

An interface may extend other interfaces like in the example below:

```
interface Style {
    color: string;
}
interface ExtendedStyle extends Style {
    width: number
}
```

An extended interface contains all properties and methods of the interface it extends and may also add its own properties and methods.

Interface Visibility Modifiers

Properties and methods are `public`.

Only methods with default implementation may be defined as `private`.

Generic Types and Functions

Generic types and functions allow creating the code capable to work over a variety of types rather than a single one.

Generic Classes and Interfaces

A class and an interface can be defined as generics, adding parameters to the type definition, like the type parameter `Element` in the following example:

```
class Stack<Element> {
    public pop(): Element { ... }
    public push(e: Element) { ... }
}
```

To use type Stack, the type argument must be specified for each type parameter:

```
let s = new Stack<string>
s.push("hello")
```

Compiler ensure type safety while working with generic types and functions.

Generic Constraints

Type parameters of generic types can be bounded. For example, the Key type parameter in the `HashMap<Key, Value>` container must have a hash method, i.e., it should be hashable.

```
interface Hashable {
    hash(): long
}
```

```
class HasMap<Key extends Hashable, Value> {
    public set(k: Key, v: Value) {
        let h = k.hash()
        ... other code ...
    }
}
```

In the above example, the `Key` type extends `Hashable`, and all methods of `Hashable` interface can be called for keys.

Generic Functions

Use a generic function to create a more universal code. Consider a function that returns the last element of the array:

```
function last(x: number[]): number {
    return x[x.length -1]
}
console.log(last([1, 2, 3])) // output: 3
```

If the same function needs to be defined for any array, define it as generic with a type parameter:

```
function last<T>(x: T[]): T {
    return x[x.length -1]
}
```

Now, the function can be used with any array.

In a function call, type argument may be set explicitly of implicitly:

```
// Explicit type argument
console.log(last<string>(["aa", "bb"]))
console.log(last<number>([1, 2, 3]))
// Implicit type argument
console.log(last([1, 2, 3]))
```

Null Safety

All types in ArkTS by default are non-nullable, so the value of a type cannot be null. It is similar to TypeScript behaviour in strict null checking mode (`strictNullChecks`), but the rules are stricter, and the `undefined` type is not supported.

In the example below, all lines cause a compile-time error:

```
let x: number = null
let y: string = null
let z: number[] = null
```

A variable that can have a null value is defined with `T | null` type.

Note: Currently ArkTS supports only nullable types and does not support general union type.

```
let x: number | null = null
x = 1 // ok
x = null // ok
if x != null { /*do smth*/ }
```

Non-Null Assertion Operator

A postfix operator `!` can be used to assert that its operand is non-null.

If applied to a null value, the operator throws an error. Otherwise, the type of the value is changed from `T | null` to T:

```
let x: number | null = 1
let y: number
y = x + 1 // compile time error: cannot add to a nullable value
y = x! + 1 // ok
```

Null-Coalescing Operator

The null-coalescing binary operator `??` checks whether the evaluation of the left-hand-side expression is equal to null. If it is, then the result of the expression is the right-hand-side expression; otherwise, it is the left-hand-side expression.

In other words, `a ?? b` equals the ternary operator `a != null ? a : b`.

In the following example, the method `getNick` returns a nickname if it is set; otherwise, the string is empty:

```
class Person {
    ...
    nick: string | null = null
    getNick(): string {
        return this.nick ?? ""
    }
}
```

Optional Chaining

Optional chaining operator `?.` allows writing code where the evaluation stops of an expression that is partially evaluated to `null`.

```
class Person {
    ...
    spouse: Person | null = null
    nick: string | null = null
    getSpouseNick(): string | null {
        return this.spouse?.nick
    }
}
```

Note: that the return type of `getSpouseNick` must be `string | null`, as the method may return null.

An optional chain may be of any length and contain any number of `?.` operators.

In the following sample, the output is a person's spouse nickname if the person has a spouse, and the spouse has a nickname.

Otherwise, the output is `null`:

```
let p: Person = ...
console.log(p?.spouse?.nick)
```

Modules

Programs are organized as sets of compilation units or modules.

Each module creates its own scope, i.e., any declarations (variables, functions, classes, etc.) declared in the module are not visible outside that module unless they are explicitly exported.

Conversely, a variable, function, class, interface, etc. exported from another module must first be imported to a module.

Export

A top-level declaration can be exported by using the keyword `export`.

A declared name that is not exported is considered private and can be used only in the module where it is declared.

```
export class Point {}
export let Origin = new Point(0, 0)
export function Distance(p1: Point, p2: Point): number { ... }
```

Import

Import declarations are used to import entities exported from other modules and provide them bindings in the current module. An import declaration consists of two parts:

- Import path that determines the module to import from;
- Import bindings that define the set of usable entities in the imported module and the form of use. i.e., qualified or unqualified use.

Import bindings may have several forms.

Let's assume a module has the path './utils' and export entities 'X' and 'Y'.

An import binding of the form `* as A` binds the name 'A', and all entities exported from the module defined by the import path can be accessed by using the qualified name `A.name`:

```
import * as Utils from "./utils"
Utils.X // denotes X from Utils
Utils.Y // denotes Y from Utils
```

An import binding of the form `{ ident1, ..., identN }` binds an exported entity with a specified name, which can be used as a simple name:

```
import { X, Y } from "./utils"
X // denotes X from Utils
Y // denotes Y from Utils
```

If a list of identifiers contains aliasing of the form `ident as alias`, then entity `ident` is bound under the name `alias`:

```
import { X as Z, Y } from "./utils"
Z // denotes X from Utils
Y // denotes Y from Utils
X // Compile-time error: 'X' is not visible
```

Top-Level Statements

All statements in the module level must be enclosed in curly braces to form a block as seen below:

```
{ statements }
```

A module may contain any number of such blocks that merge logically into a single block in the order of writing:

```
{ statements_1 }
/* declarations */
{ statements_2 }
```

Is equal to:

```
/* declarations */
{ statements_1; statements_2 }
```

If a module contains a main function (program entry point), statements in this block are executed immediately before the body of this function. Otherwise, they are executed before execution of any other statement of the module.

Program Entry Point

An entry point of a program (application) is the top-level main function. The main function should have either an empty parameter list or a single parameter of string[] type.

```
function main() {
    console.log("this is the entry")
}
```