

TypeScript Language Features

TypeScript is a superset of JavaScript. That means that the TypeScript language includes the entire JavaScript language plus a collection of useful additional features. This contrasts with the various subsets of JavaScript and the various linting tools that seek to reduce the available features to create a smaller language with fewer surprises. This part will introduce you to the extra language features, starting with simple type annotations and progressing to more advanced features and structural elements of TypeScript.

One important thing to remember is that all the standard control structures found in JavaScript are immediately available within a TypeScript program. This includes the following:

- Control flows
- Data types
- Operators
- Subroutines

The basic building blocks of your program will come from JavaScript, including if statements, switch statements, loops, arithmetic, logical tests, and functions. This is one of the key strengths of TypeScript — it is based on a language (and a family of languages) that is already familiar to a vast and varied collection of programmers. JavaScript is thoroughly documented not only in the ECMA-262 specification, but also in guides, on developer network portals, forums, and question-and-answer websites. When features are added to JavaScript, they will also appear in TypeScript.

The TypeScript compiler is typically updated with new JavaScript features early in their specification. Most of the features are available before browsers support them. In many cases, you can use the features in your TypeScript program as the compiler will convert them into code that targets the older versions of the ECMAScript standard.

Each of the language features discussed in this part has short, self-contained code examples that put the feature in context. For the purposes of introducing and explaining features, the examples are short and to the point; this allows the part to be read end to end. However, this also means you can refer back to the part as a reference later on. Once you have read this part, you should know everything you will need to understand the more complex examples described throughout the rest of the guide.

JavaScript Is Valid TypeScript

Before we find out more about the TypeScript syntax, it is worth stressing one important fact: All JavaScript is valid TypeScript. You don't need to discard any of your JavaScript know-how as it can all be transferred directly to your TypeScript code. You can take existing JavaScript code, add it to a TypeScript file, and all the statements will be valid. There is a subtle difference between valid code and error-free code in TypeScript; because, although your code may work, the TypeScript compiler will warn you about any potential problems it has detected. Finding subtle and previously undetected bugs is a common story shared by programmers making the transition to TypeScript.

If you transfer a JavaScript listing into a TypeScript file, you may receive errors or warnings even though the code is considered valid. A common example comes from the dynamic type system in JavaScript wherein it is perfectly acceptable to assign values of different types to the same variable during its lifetime. TypeScript detects these assignments and generates errors to warn you that the type of the variable has been changed by the assignment. Because this is a common cause of errors in a program, you can correct the error by creating separate variables, by performing a type assertion, or by making the variable dynamic. There is further information on type annotations later in this part, and the type system is discussed in detail in [Part 3](#).

Unlike some compilers that will only create output where no compilation errors are detected, the TypeScript compiler will still attempt to generate sensible JavaScript code. The TypeScript code shown in [Listing 1-1](#) generates an error, but the JavaScript output is still produced. This is an admirable feature, but as always with compiler warnings and errors, you should correct the problem in your source code and get a clean compilation. If you routinely ignore these messages, your program will eventually exhibit unexpected behavior. In some cases, your listing may contain errors that are so severe that the TypeScript compiler won't be able to generate the JavaScript output.

the only exceptions to the “all JavaScript is valid TypeScript” rule are the `with` statement and vendor-specific extensions, until they are formally added to the ECMaScript specification. you could technically still use the `with` statement, but all statements within the block would be unchecked.

The JavaScript `with` statement in [Listing 1-1](#) shows two examples of the same routine. Although the first calls `Math.PI` explicitly, the second uses a `with` statement, which adds the properties and functions of `Math` to the current scope. Statements nested inside the `with` statement can omit the `Math` prefix and call properties and functions directly, for example, the `PI` property or the `floor` function.

At the end of the `with` statement, the original lexical scope is restored, so subsequent calls outside of the `with` block must use the `Math` prefix.

Listing 1-1. Using JavaScript's “with” statement

```
// Not using with
const radius1 = 4;
const area1 = Math.PI * radius1 * radius1;

// Using with
const radius2 = 4;
with (Math) {
  const area2 = PI * radius2 * radius2;
}
```

The `with` statement was not allowed in strict mode in ECMAScript 5 and later versions of ECMAScript use strict mode by default for classes and modules. TypeScript treats `with` statements as an error and will treat all types within the `with` statement as dynamic types. This is due to the following:

- The fact it is disallowed in strict mode.
- The general opinion that the `with` statement is dangerous.
- The practical issues of determining the identifiers that are in scope at compile time.

So, with these minor exceptions to the rule in mind, you can place any valid JavaScript into a TypeScript file and it will be valid TypeScript. As an example, here is the area calculation script transferred to a TypeScript file.

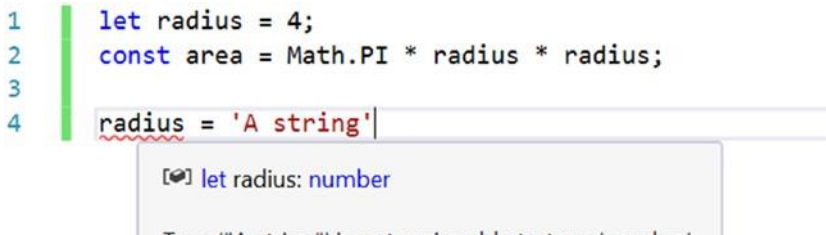
the eCmaScript 6 specification, also known as “eS6 harmony,” represented a substantial change to the JavaScript language. the specification has been divided into annual chunks, released as eCmaScript 2015, eCmaScript 2016, and so on.

Listing 1-2. Transferring JavaScript in to a TypeScript file

```
const radius = 4;
const area = Math.PI * radius * radius;
```

In Listing 1-2, the statements are just plain JavaScript, but in TypeScript the variables `radius` and `area` will both benefit from type inference. Because `radius` is initialized with the value 4, it can be inferred that the type of `radius` is `number`. With just a slight increase in effort, the result of multiplying `Math.PI`, which is known to be a number, with the `radius` variable that has been inferred to be a number, it is possible to infer the type of `area` is also a number.

With type inference at work, assignments can be checked for type safety. Figure 1-1 shows how an unsafe assignment is detected when a string is assigned to the `radius` variable. There is a more detailed explanation of type inference in Part 3. For now, rest assured that type inference is a good thing, and it will save you a lot of effort.



```
1 let radius = 4;
2 const area = Math.PI * radius * radius;
3
4 radius = 'A string'
```

let radius: number

Figure 1-1. Static type checking

Variables

TypeScript variables must follow the JavaScript naming rules. The identifier used to name a variable must satisfy the following conditions.

The first character must be one of the following:

- an uppercase letter
- a lowercase letter
- an underscore
- a dollar sign
- a Unicode character from categories—*Uppercase letter* (Lu), *Lowercase letter* (Ll), *Title case letter* (Lt), *Modifier letter* (Lm), *Other letter* (Lo), or *Letter number* (Nl)

Subsequent characters follow the same rule and additionally allow the following:

- numeric digits
- a Unicode character from categories—*Non-spacing mark* (Mn), *Spacing combining mark* (Mc), *Decimal digit number* (Nd), or *Connector punctuation* (Pc)
- the Unicode characters U+200C (Zero Width Non-Joiner) and U+200D (Zero Width Joiner)

You can test a variable identifier for conformance to the naming rules using the JavaScript variable name validator by Mathias Bynens.

<http://mothereff.in/js-variables>

the availability of some of the more exotic characters can allow some interesting identifiers. you should consider whether this kind of variable name causes more problems than it solves. For example, this is valid JavaScript: `const = 'Dignified';`

Variables declared with `const` or `let` are block scoped, whereas variables declared with the older `var` keyword are function scoped. If you omit these keywords, you are implicitly (and perhaps accidentally) declaring the variable in the global scope. It is advisable to reduce the number of variables you add to the global scope, as they are at risk of name collisions. You can avoid the global scope by declaring variables local to their use, such as within functions, modules, namespaces, classes, or a simple set of curly braces if you are using the block-scoped keywords.

When you limit the scope of a variable, it means it cannot be manipulated from outside of the scope of which it was created. The scope follows a nesting rule that allows a variable to be used in the current scope, and in inner nested scopes –but not outside. In other words, you can use variables declared in the current scope and variables from wider scopes. See Listing 1-3.

Listing 1-3. Block scope

```
let globalScope = 1;

{
  let blockScope = 2;

  // OK. This is from a wider scope
  globalScope = 100;

  // Error! This is outside of the scope the variable is declared in
  nestedBlockScope = 300;

  {
    let nestedBlockScope = 3;

    // OK. This is from a wider scope
    globalScope = 1000;

    // OK. This is from a wider scope
    blockScope = 2000;
  }
}
```

TypeScript catches scope violations and will warn you when you attempt to access a variable that is declared in a narrower scope. You can help the compiler to help you by avoiding a valid, but often accidental coding style of reusing a name in a different scope. In Listing 1-4, the logging statements work correctly, with both `firstName` variables being preserved separately. This means the original variable is not overwritten by the nested variable with the same name.

Listing 1-4. Name reuse with `let`

```
let firstName = 'Chris';

{
  let firstName = 'Tudor';

  console.log('Name 1: ' + firstName);
}

console.log('Name 2: ' + firstName);

// Output:
// Name 1: Tudor
// Name 2: Chris
```

If in place of the `let` keyword the `var` keyword had been used, both logging statements would show the name “Tudor,” as shown in Listing 1-5. Despite both variables appearing to be a separate declaration, only one variable named `firstName` exists, and it is overwritten by the nested scope.

Listing 1-5. Name reuse with var

```
var firstName = 'Chris';

{
  var firstName = 'Tudor';

  console.log('Name 1: ' + firstName); }

console.log('Name 2: ' + firstName);

// Output:
// Name 1: Tudor
// Name 2: Tudor
```

Based on this example, you could decide whether you want let-flavored scope, or var-flavored scope for the variable whose name you reuse; or you can use better variable names to avoid relying on either behavior.

Constants

Constants are variables that follow the scope rules of the **let** keyword, but that cannot be reassigned. When you declare a variable with the **const** keyword, you can't later assign a new value to the variable. It is important to note that this doesn't make the variable immutable, as you can see in Listing 1-6. Direct assignments after a constant is declared are not allowed, but the value within the constant can be mutated, for example, by calling methods on the value already assigned, or by adding items in the case of an array

Listing 1-6. Constants

```
const name = 'Lily';

// Error! Cannot assign to name because it is a constant
name = 'Princess Sparkles';

const digits = [1, 2, 3];

// Mutable - this changes the value of digits without using an assignment
digits.push(4, 5, 6);
```

The recommended coding style is to start by using the **const** keyword for all variables, and open a variable to reassignment with the **let** keyword if you decide to allow it. Constants reduce code complexity by following the principle of least privilege as you don't need to continue scanning the program to see whether a later assignment will change the value; but be aware that the variable is not immutable and can be changed in other ways.

Types

TypeScript is optionally statically typed; this means that types are checked automatically to prevent accidental assignments of invalid values. It is possible to opt out of this by declaring dynamic variables. Static type checking reduces errors caused by accidental misuse of types. You can also create types to replace

primitive types to prevent parameter ordering errors, as described in Part 3. Most important, static typing allows development tools to provide intelligent autocompletion.

Figure 1-2 shows autocompletion that is aware of the variable type, and supplies a relevant list of options. It also shows the extended information known about the properties and methods in the autocompletion list. Contextual autocompletion is useful enough for primitive types — but most reasonable integrated development environments can replicate simple inference even in a JavaScript file. However, in a program with many custom types, modules, and classes, the deep type knowledge of the TypeScript Language Service means you will have sensible autocompletion throughout your entire program.

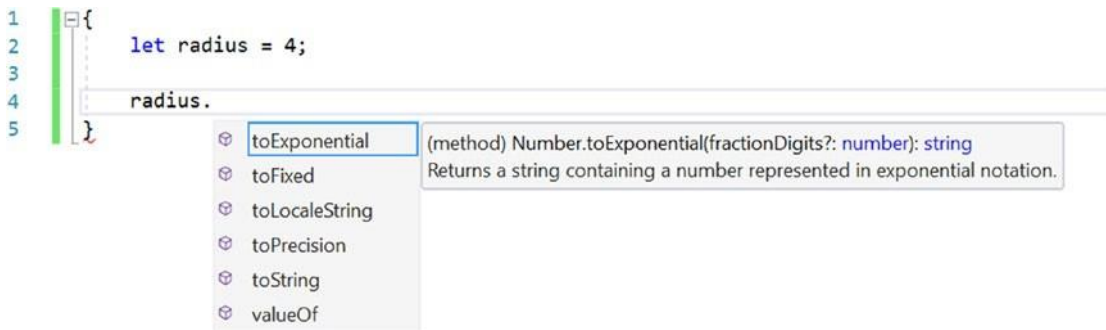


Figure 1-2. TypeScript autocompletion

Type Annotations

Although the TypeScript language service is expert at inferring types automatically, there are times when it can't fathom your intentions. There will also be times where you will wish to make a type explicit for safety or to narrow the type. In these cases, you can use a type annotation to specify the type.

For a variable, the type annotation comes after the identifier and is preceded by a colon. Figure 1-3 shows the combinations that result in a typed variable. The combinations are shown in order of preference with the first being most desirable. The least preferable is the most verbose style of both adding a type annotation and assigning the value. Although this is the style shown in many examples in this part, in practice this is the one you will use the least.

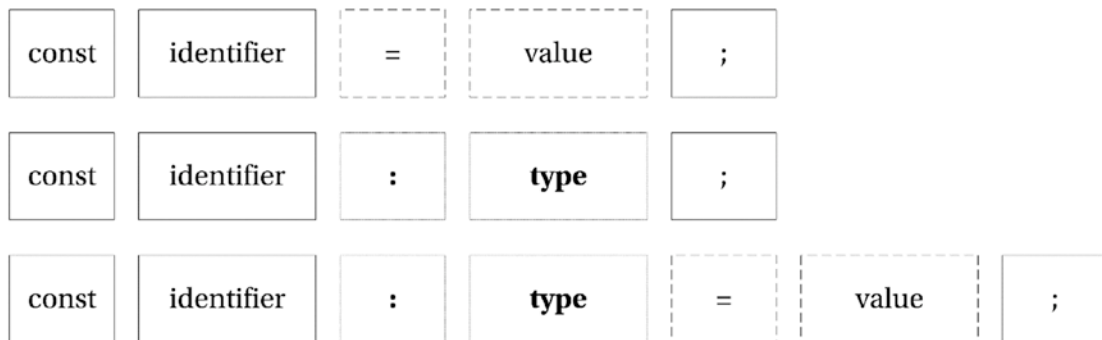


Figure 1-3. Typed variable combinations

To demonstrate type annotations in code, Listing 1-7 shows an example of a variable that has an explicit type annotation that marks the variable as a string. Primitive types are the simplest form of type annotation, but you are not restricted to such simple types.

Listing 1-7. Explicit type annotation

```
const name: string = 'Steve';
```

The type used to specify an annotation can be a primitive type, an array type, a function signature, a type alias, or any complex structure you want to represent including the names of classes and interfaces you create. You can also be permissive by allowing one of multiple types (union types), or more restrictive by limiting the range of allowable values (literal types). If you want to opt out of static type checking, you can use the special `any` type, which marks a variable's type as dynamic. No checks are made on dynamic types. Listing 1-8 shows a range of type annotations that cover some of these different scenarios.

Listing 1-8. Type annotations

```
// primitive type annotation
const name: string = 'Steve';
const heightInCentimeters: number = 182.88;
const isActive: boolean = true;

// array type annotation
const names: string[] = ['James', 'Nick', 'Rebecca', 'Lily'];

// function annotation with parameter type annotation and return type annotation
let sayHello: (name: string) => string;

// implementation of sayHello function
sayHello = function (name) {
  return 'Hello ' + name;
};

// object type annotation
let person: { name: string; heightInCentimeters: number; };

// Implementation of a person object
person = {
  name: 'Mark',
  heightInCentimeters: 183
};
```

although many languages specify the type before the identifier, the placement of type annotations in TypeScript after the identifier helps to reinforce that the type annotation is optional. it also allows you to use the full range of variable statements, including `const` and `let`. this style of type annotation is also inspired by *type theory*.

If a type annotation becomes too complex, you can create an interface, or a type alias to represent the type to simplify annotations. Listing 1-9 demonstrates how to simplify the type annotation for the person object, which was shown at the end of the previous example in Listing 1-8. This technique is especially useful if you intend to reuse the type as it provides a reusable definition. Interfaces and type aliases are not limited to describing object types; they are flexible enough to describe any structure you are likely to encounter. Interfaces are discussed in more detail later in this part.

When choosing whether to use an interface or a type alias, it is worth understanding the things an interface can do that a type alias cannot do. An interface can be used in an `extends` or `implements` clause, which means you can explicitly use them when defining other interfaces and classes. An interface can also accept type arguments, making the interface generic. A type alias can do neither of these.

Listing 1-9. Interface and type alias

```
// Interface
interface PersonInterface { name:
    string; heightInCentimeters:
    number;
}

const sherlock: PersonInterface = {
    name: 'Bendict',
    heightInCentimeters: 183
}

// Type Alias
type PersonType = { name: string;
    heightInCentimeters: number;
};

const john: PersonType = {
    name: 'Martin',
    heightInCentimeters: 169
}
```

Primitive Types

The primitive types in TypeScript are incredibly basic, but through the tools of the type system you can combine them, widen them, and narrow them to represent the concepts in your program. These types directly represent the underlying JavaScript types and follow the standards set for those types:

- `string` – a sequence of UTF-16 code units
- `boolean` – true or false
- `number` – a double-precision 64-bit floating point value
- `symbol` – a unique, immutable symbol, substitutable for a string as an object key

There are no special types to represent integers or other specific variations on the numeric type as it wouldn't be practical to perform static analysis to ensure all possible values assigned are valid.

The type system also contains several types that represent special values:

- The `undefined` type is the value of a variable that has not been assigned a value.
- The `null` type can be used to represent an intentional absence of an object value. For example, if you had a method that searched an array of objects to find a match, it could return `null` to indicate that no match was found.
- The `void` type is used to represent cases where there is no value, for example, to show that a function doesn't return anything.
- The `never` type represents an unreachable section of code, for example a function that throws an exception has the return type of `never`.

Object and Dynamic Types

Everything that isn't a primitive type in TypeScript is a subclass of the `object` type. Most of the types in your program are likely to fall within this definition.

The final item in our types is the `dynamic` any type, which can be used to represent literally any type. When using dynamic types, there is no compiler type checking for the type.

The any type is also used by the compiler in situations where it cannot infer the type automatically, although you can disallow these implicit any types using a compiler flag. You can also use it in cases where you don't want the type to be checked by the compiler, which gives you access to all of the dynamic features of the JavaScript language.

Now that the fundamental types are all described, the next section covers some special mechanisms for combining the types in ways that either narrow or widen the range of allowable values.

Enumerations

Enumerations are one of the simplest narrowing types. Enumerations represent a collection of named elements that you can use to avoid littering your program with hard-coded values. By default, enumerations are zero based although you can change this by specifying the first value, in which case numbers will increment from value you set. You can opt to specify values for all identifiers if you wish to.

In Listing 1-10 the `VehicleType` enumeration can be used to describe vehicle types using well-named identifiers throughout your program. The value passed when an identifier name is specified is the number that represents the identifier, for example, in Listing 1-10 the use of the `VehicleType.Lorry` identifier results in the number 5 being stored in the type variable. It is also possible to get the identifier name from the enumeration by treating the enumeration like an array.

Listing 1-10. Enumerations

```
enum VehicleType {  
    PedalCycle,  
    MotorCycle,  
    Car,  
    Van,  
    Bus,  
    Lorry  
}  
  
const type = VehicleType.Lorry;  
  
const typeName = VehicleType[type]; // 'Lorry'
```

In TypeScript enumerations are open ended. This means all declarations with the same name inside a common root will contribute toward a single type. When defining an enumeration across multiple blocks, subsequent blocks after the first declaration must specify the numeric value to be used to continue the sequence, as shown in Listing 1-11. This is a useful technique for extending code from third parties, in ambient declarations, and from the standard library.

Listing 1-11. Enumeration split across multiple blocks

```
enum BoxSize {  
  Small,  
  Medium }  
  
//...
```

```
enum BoxSize {  
  Large = 2,  
  XLarge,  
  XXLarge }
```

Consumers of enumerations declared in multiple blocks can tell no difference with an enumeration declared in one block, as shown in Figure 1-4.

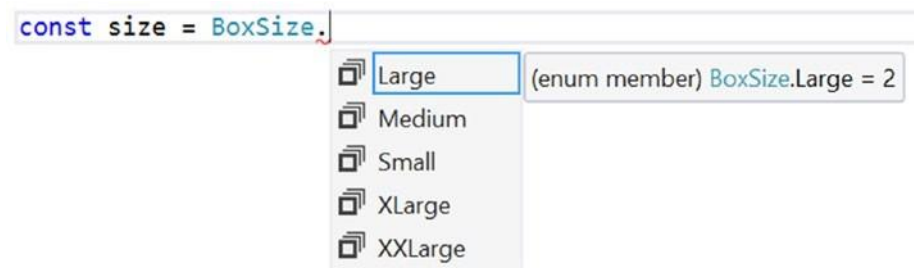


Figure 1-4. Using a multi-block enumeration

the term *common root* comes from graph theory. in TypeScript, this term relates to a particular location in the tree of modules within your program. Whenever declarations are considered for merging, they must have the same fully qualified name, which means the same name at the same level in the tree.

Bit Flags

You can use an enumeration to define bit flags. Bit flags allow a series of items to be selected or deselected by switching individual bits in a sequence on and off. To ensure that each value in an enumeration relates to a single bit, the numbering must follow the binary sequence whereby each value is a power of two, for example:

1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1,024, 2,048, 4,096, and so on

Listing 1-12 shows an example of using an enumeration for bit flags. By default, when you create a variable to store the state, all items are switched off. To switch on an option, it can simply be assigned to the variable. To switch on multiple items, items can be combined with the bitwise OR operator (`|`). Items remain switched on if you happen to include them multiple times using the bitwise OR operator.

Listing 1-12. Flags

```
enum DiscFlags {
    None = 0,
    Drive = 1,
    Influence = 2,
    Steadiness = 4,
    Conscientiousness = 8
}

// Using flags
var personality = DiscFlags.Drive | DiscFlags.Conscientiousness;

// Testing flags

// true
var hasD = (personality & DiscFlags.Drive) == DiscFlags.Drive;

// false
var hasI = (personality & DiscFlags.Influence) == DiscFlags.Influence;

// false
var hasS = (personality & DiscFlags.Steadiness) == DiscFlags.Steadiness;

// true
var hasC = (personality & DiscFlags.Conscientiousness) == DiscFlags.Conscientiousness;
```

The value assigned to each item in an enum can be constant, or computed. Constant values are any expression that can be interpreted by the type system, such as literal values, calculations, and binary operators. Computed values are expressions that could not be efficiently interpreted by the compiler, such as assigning a string length, or calling out to a method. The terminology here is dangerously overloaded; the term “constant” when considering “constant vs computed” should not be confused with the `const` keyword, which can be used with variables and with a special kind of enumeration, called a constant enumeration.

Constant Enumerations

A constant enumeration can be created using the `const` keyword, as shown in Listing 1-13. Unlike a normal enumeration, a constant enumeration is erased during compilation and all code referring to it is replaced with hard-coded values.

Listing 1-13. Constant enumeration

```
const enum VehicleType {
    PedalCycle,
    MotorCycle,
    Car,
```

```

    Van,
    Bus,
    Lorry
}

```

```
const type = VehicleType.Lorry;
```

The compiled JavaScript output for this example is shown in Listing 1-14. The entire enumeration is gone, and the code referencing the lorry vehicle type has been replaced with the value 5. To aid comprehension, comments are inserted to describe the literal values.

Listing 1-14. JavaScript output of a constant enumeration

```
var type = 5 /* Lorry */;
```

To make the inlining of values possible, constant enumerations are not allowed to have computed members.

Union Types

A union type widens the allowable values by specifying that the value can be of more than a single type. Libraries such as jQuery commonly expose functions that allow you to pass either a jQuery object, or a string selector, for example; and union types allow you to limit the possible values to just these two types (rather than resorting to a completely dynamic type).

As Listing 1-15 demonstrates, it is possible to create a type that is either a Boolean or a number. Union types use the pipe-delimiter to separate each of the possible types, which you can read as an “OR.” Attempting to supply a value that doesn’t match one of the types in the union results in an error.

Listing 1-15. Union Types

```

// Type annotation for a union type
let union: boolean | number;

// OK: number
union = 5;

// OK: boolean
union = true;

// Error: Type "string" is not assignable to type 'number | boolean'
union = 'string';

// Type alias for a union type
type StringOrError = string | Error;

// Type alias for union of many types
type SeriesOfTypes = string | number | boolean | Error;

```

When creating a union type, consider using a type alias to reduce the repetition of the definition in your program and to give the concept a name. A union can be created using any types available in your program, not just primitive types.

Literal Types

Literal types can be used to narrow the range of allowable values to a subset of the type, such as reducing a string, to a set of specific values. Listing 1-16 shows a type of `Kingdom`, which has the six taxonomic divisions of living organisms as possible values. Variables with the `Kingdom` type may only use the specific values included in the literal type.

Listing 1-16. String literal type

```
type Kingdom = 'Bacteria' | 'Protozoa' | 'Chromista' | 'Plantae' | 'Fungi' | 'Animalia';
```

```
let kingdom: Kingdom;
```

```
// OK
```

```
kingdom = 'Bacteria';
```

```
// Error: Type 'Protista' is not assignable to type 'Kingdom'
```

```
kingdom = 'Protista';
```

Literal types are really just union types made up of specific values, so you can also create a number literal type, or a union/literal hybrid type using the same syntax.

Listing 1-17. Number literal types and hybrid union/literal types

```
// Number literal type
```

```
type Fibonacci = 1 | 2 | 3 | 5 | 8 | 13;
```

```
let num: Fibonacci;
```

```
// OK
```

```
num = 8;
```

```
// Error: Type '9' is not assignable to type 'Fibonacci'
```

```
num = 9;
```

```
// Hybrid union/literal type
```

```
type Randoms = 'Text' | 10 | false;
```

```
let random: Randoms;
```

```
// OK
```

```
random = 'Text';
```

```
random = 10;
```

```
random = false;
```

```
// Error: Not assignable.
```

```
random = 'Other String';
```

```
random = 12;
```

```
random = true;
```

The behavior of literal types is similar to the behavior of enumerations, so if you are using only numbers in your literal type, consider whether an enumeration would be more expressive in your program.

Intersection Types

An intersection type combines several different types into a single supertype that includes the members from all participating types. Where a union type is “either type A or B,” an intersection type is “both type A and B.” Listing 1-18 shows two interfaces for Skier and Shooter, which are combined into the biathlete intersection type. Intersection types use the ampersand character, which you can read as “AND.”

Listing 1-18. Intersection Types

```
interface Skier {  
    slide(): void;  
}  
  
interface Shooter {  
    shoot(): void;  
}  
  
type Biathlete = Skier & Shooter;
```

To see the result of an intersection type, Figure 1-5 shows the autocompletion displayed for the biathlete type, with both the shoot and slide methods present. The autocompletion also shows the originating type of the members.

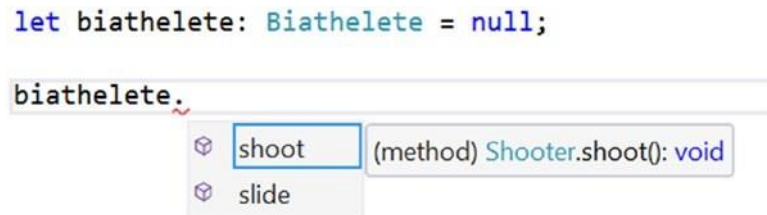


Figure 1-5. Intersection type members

Intersection types are useful for working with mixins, which you’ll read about in Part 4.

Arrays

TypeScript arrays have precise typing for their contents. To specify an array type, you simply add square brackets after the type name. This works for all types whether they are primitive or custom types. When you add an item to the array, its type will be checked to ensure it is compatible. When you access elements in the array, you will get quality autocompletion because the type of each item is known. Listing 1-19 demonstrates each of these type checks.

There are some interesting observations to be made in Listing 1-19. When the monuments variable is declared, the type annotation for an array of Monument objects can either be the shorthand: `Monument[]` or the longhand: `Array<Monument>`—there is no difference in meaning between these two styles. Therefore, you should opt for whichever you feel is more readable. Note that the array is instantiated after the equals sign using the empty array literal (`[]`). You can also instantiate it with values, by adding them within the brackets, separated by commas.

The objects being added to the array using `monuments.push(...)` are not explicitly `Monument` objects. This is allowed because they are compatible with the `Monument` interface. If you miss a property you will be warned that the type is not compatible; and if you add an additional member you will receive a warning too, which helps to catch misspelled member names.

The array is sorted using `monuments.sort(...)`, which takes in a function to compare values. When the comparison is numeric, the comparer function can simply return `a - b`, in other cases you can write custom code to perform the comparison and return a positive or negative number to be used for sorting (or a zero if the values are the same).

Listing 1-19. Typed arrays

```
interface Monument { name:
    string; heightInMeters:
    number;
}

// The array is typed using the Monument interface
const monuments: Monument[] = [];

// Each item added to the array is checked for type compatibility
monuments.push({
    name: 'Statue of Liberty',
    heightInMeters: 46
});

monuments.push({
    name: 'Peter the Great',
    heightInMeters: 96
});

monuments.push({
    name: 'Angel of the North',
    heightInMeters: 20
});

function compareMonumentHeights(a: Monument, b: Monument) {
    if (a.heightInMeters > b.heightInMeters) {
        return -1;
    }
    if (a.heightInMeters < b.heightInMeters) {
        return 1;
    }
    return 0;
}

// The array.sort method expects a comparer that accepts two Monuments
const monumentsOrderedByHeight = monuments.sort(compareMonumentHeights);

// Get the first element from the array, which is the tallest
const tallestMonument = monumentsOrderedByHeight[0];

// Peter the Great
console.log(tallestMonument.name);
```


The elements in an array are accessed using an index. The index is zero based, so the first element in the `monumentsOrderedByHeight` array is `monumentsOrderedByHeight[0]`. When an element is accessed from the array, autocompletion is supplied for the `name` and `heightInMeters` properties.

Tuple Types

A tuple type uses an array, and specifies the type of elements based on their position. Listing 1-20 shows a tuple type where three items in the array are typed.

Listing 1-20. Tuple types

```
let poem: [number, boolean, string];

// OK
poem = [1, true, 'love'];

// Error: 'string' is not assignable to 'number'
poem = ['my', true, 'love'];
```

When accessing the types by index, the type is known and will be checked by the compiler. The autocompletion list will also be type specific as shown in Figure 1-6, where the third item (at index 2) is known to be a string.

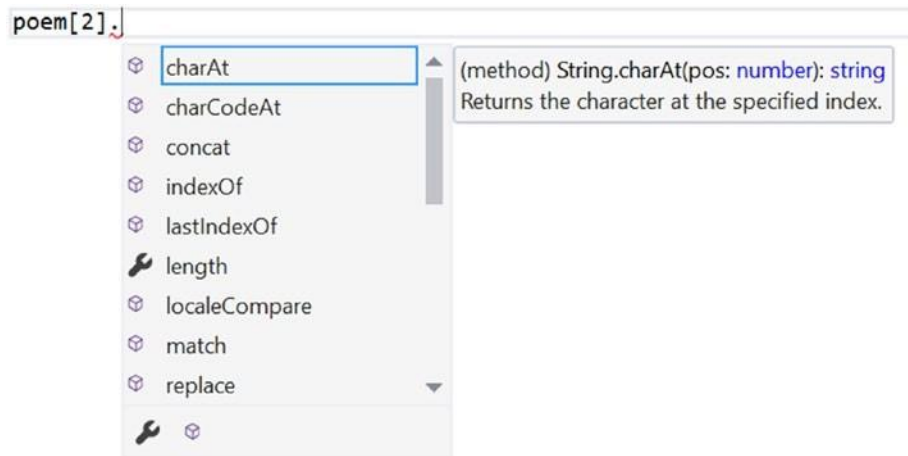


Figure 1-6. Autocompletion members for tuple types

Tuples are named after the number of items they define (and tuples larger than seven items are named *n-tuples*).

- Pair: 2 items
- Triple: 3 items
- Quadruple: 4 items

- Quintuple: 5 items
- Sextuple: 6 items
- Septuple: 7 items

The common use case for tuples is the ability to return multiple values from a method without having to define a more complex structure. Conceptually, tuples are useful as long as the data is related and its lifetime is short.