4.

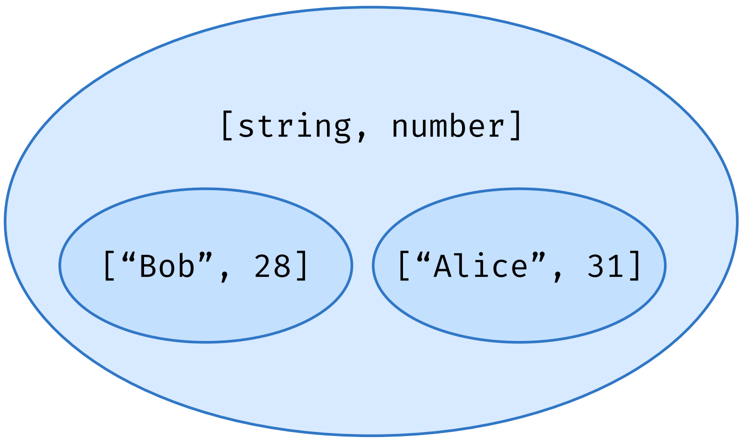
Arrays & Tuples

After learning about object types in the previous chapter, let's take a look at the second most important data structure of Type-level TypeScript – **Tuples**.

It might come as a surprise, but Tuples are much more interesting than Arrays at the type level. In fact, they **are** the real arrays of type-level programs. In this chapter, we are going to learn why they are so useful and how to use all their awesome features. Let's get started!

Tuples

Tuple types define sets of arrays with a **fixed length**, and each index can contain a value of **a different type**. For example, the tuple `[string, number]` defines the set of arrays containing **exactly two values**, where the first value is a `string` and the second value is a `number`.



Tuples are essentially **lists of types**! They can contain zero, one, or many items, and each one can be a completely different type. Unlike unions, types in a tuple are ordered and can be present more than once. They look just like JavaScript arrays and are their type-level equivalent:

```
type Empty = [];
type One = [1];
type Two = [1, "2"]; // types can be different!
type Three = [1, "2", 1]; // tuples can contain duplicates
```

Reading indices of a tuple

Just like with JS arrays, you can access a value inside a tuple by using a **numerical index**:

```
type SomeTuple = ["Bob", 28];

type Name = SomeTuple[0]; // "Bob"
type Age = SomeTuple[1]; // 28
```

The only difference is that tuples are indexed by **number literal types** and not just numbers.

Reading several indices

We have seen in Objects & Records that it was possible to simultaneously read several keys from an object using a **union** of string literals:

```
type User = { name: string; age: number; isAdmin: true };

type NameOrAge = User["name" | "age"]; // => string | number
```

We can do the same with tuples by using a **union** of number literal types!

```
type SomeTuple = ["Bob", 28, true];

type NameOrAge = SomeTuple[0 | 1]; // => "Bob" | 28
```

We can simultaneously read all indices in a tuple `T` with `T[number]` :

```
type SomeTuple = ["Bob", 28, true];

type Values = SomeTuple[number]; // "Bob" | 28 | true
```

`T[number]` is essentially a way of turning **a list into a set** at the type level.

Remember the trick of using `O[keyof O]` to get the union of all values in an object type `O` ? Well, `T[number]` is how you do the same thing with tuples.

But couldn't we use `keyof` here as well?

Can we use `keyof` ?

Technically we can, but the `keyof` keyword will not only return all indices but also the names of all methods on the prototype of Array, like `map` , `filter` , `reduce` , etc:

```
type Keys = keyof ["Bob", 28]; // "0" | "1" | "map" | "filter" | ...

const key: Keys = "map"; // ✅ 😬
```

`keyof` is a bit impractical with tuples, so we rarely use it.

Concatenating tuples

Just like with JS arrays, we can **spread** the content of a tuple into another one using the `...` *rest element* syntax:

```
type Tuple1 = [4, 5];

type Tuple2 = [1, 2, 3, ...Tuple1];
// => [1, 2, 3, 4, 5]
```

And here is how to merge 2 tuples together:

```
type Tuple1 = [1, 2, 3];
type Tuple2 = [4, 5];

type Tuple3 = [...Tuple1, ...Tuple2];
// => [1, 2, 3, 4, 5]
```

Tuples created with `...` are called Variadic Tuples. They are super handy! It will become an extremely powerful tool in our toolbox once we start combining them with other features like conditional types.

Named Indices

The Tuple syntax allows giving names to indices. Just like with objects, names precede values, and names and values are separated by a colon `:` character:

```
type User = [firstName: string, lastName: string];
```

Names help **disambiguate** the purpose of values of the same type, which makes them pretty useful. They help us understand the kind of data we are dealing with but they **don't affect** the behavior of **the type-checker** in any way.

Optional indices

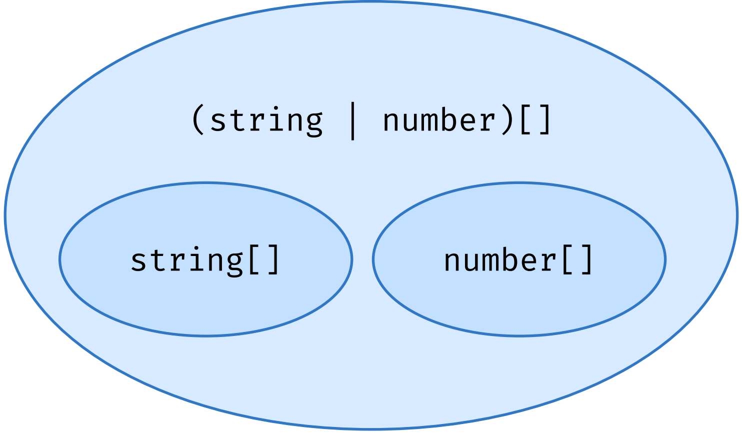
A lesser-known feature of tuples is their ability to have optional indices! To mark an index as optional, you only need to add a question mark `?` after it:

```
type OptTuple = [string, number?];
//                         ^ optional index!

const tuple1: OptTuple = ["Bob", 28]; // ✅
const tuple2: OptTuple = ["Bob"]; // ✅
const tuple3: OptTuple = ["Bob", undefined]; // ✅
//    ^ we can also explicitly set it to `undefined`
```

Arrays

In TypeScript, array types are extremely common. They represent sets of arrays with an **unknown length**. All of their values must share **the same type**, but since this type can be a **union**, they can also represent arrays of **mixed values**:



In TypeScript, Array types can be created in **two equivalent ways**: either by adding square brackets after a type, like `number[]` or by using the more explicit `Array<number>` generic:

```
type Tags = string[];

type Users = Array<User>; // same as `User[]`

type Bits = (0 | 1)[];
```

Since all values in an `Array` have the same type, arrays don't contain a ton of type-level information — they're just wrappers around a single type. In that regard, they are very similar to Records:

```
// `Arrays` are similar to `Records`:

type BooleanRecord = { [k: string]: boolean };
type BooleanArray = boolean[];
```

`BooleanRecord` and `BooleanArray`:

- both have an unknown number of keys or indices
- both contain a single type that is shared between all of their values

In the upcoming chapters of this course, we will mainly focus on **object types** and **tuples**. Since they are the type-level equivalents of our good old objects and arrays that we already know so well, we will be able to use them in algorithms we are already familiar with, like recursive loops!

Extracting the type in an `Array`

Just like with Tuples, we can read the type of values in an array by using the `number` type:

```
type SomeArray = boolean[];

type Content = SomeArray[number]; // boolean
```

Mixing Arrays and Tuples

Since the introduction of Variadic Tuples, we can use the `...` rest element syntax to mix Arrays and Tuples. This allows us to create types representing arrays **with any number of values**, but with a few **fixed types** at specific indices.

```
// number[] that starts with 0
type PhoneNumber = [0, ...number[]];

// string[] that ends with a `!`
type Exclamation = [...string[], "!"];

// non-empty list of strings
type NonEmpty = [string, ...string[]];

// starts and ends with a zero
type Padded = [0, ...number[], 0];
```

This is great to **capture** some of the **invariants** of our code that we often don't bother typing properly. For example, a French social security number always starts with a `1` or a `2`. We can encode this with this type:

```
type FrenchSocialSecurityNumber = [1 | 2, ...number[]];
```

Neat!

Tuples & Function Arguments

Now, if we combine **variadic** tuples, **named** indices, and **optional** indices, here is the kind of tuples we can create:

```
type UserTuple = [name: string, age?: number, ...addresses: string[]];
```

Feels familiar, doesn't it? It looks just like functions arguments:

```
function createUser(name: string, age?: number, ...addresses: string[]) {
```

We could also have used our `UserTuple` to type the same function:

```
function createUser(...args: UserTuple) {
  const [name, age, ...addresses] = args;
  //     ~~~~  ~~~     ~~~~~~~~~
  //      ^     ^          ^
  //   string   number   string[]
}

createUser("Gabriel", 29, "28 Central Ave", "7500 Greenback Ln"); // ✅
createUser("Bob"); // ✅ `age` is optional and addresses can be empty.
createUser("Alice", 0, false);
//                     ~~~~~ ❌ not a `string`!
```

Using tuples to type function parameters can be convenient if you want to **share the types of parameters** between several different functions:

```
function createUser(...args: UserTuple) {}
function updateUser(user: User, ...args: UserTuple) {}
```

or if your function has **several signatures**:

```
type Name =
  | [first: string, last: string]
  | [first: string, middle: string, last: string];

function createUser(...name: Name) {}

createUser("Gabriel", "Vergnaud"); // ✅
createUser("Gabriel", "Léo", "Vergnaud"); // ✅
createUser("Gabriel"); // ❌
createUser("Oups", "Too", "Many", "Names"); // ❌
```

Leading Rest Elements

Now that you have seen how similar tuples and function arguments look, you may be thinking that we can always swap one for the other.

This is true in most cases, but here is something you **wouldn't be able to do** with regular **function arguments** — *Leading Rest Elements*. It's the ability to use the `...` syntax **before** other elements in a tuple type.

As an example, let's type the zipWith function from lodash. `zipWith(...arrays, zipper)` takes several arrays, a "zipper" function, and zips them into a single array by calling the zipper function with all values for each index.

Here is a possible way of typing `zipWith` :

```
type ZipWithArgs<I, O> = [
  ...arrays: I[][], // <- Leading rest element!
  zipper: (...values: I[]) => O
];


declare function zipWith<I, O>(...args: ZipWithArgs<I, O>): O[];
// ^ The `declare` keyword lets us define a type
// without specifying an implementation

const res = zipWith(
  [0, 1, 2, 3, 4],
  [1930, 1987, 1964, 2013, 1993],
  [149, 170, 186, 155, 180],
  (index, year, height) => {
    // index, year, and height are inferred as
    // numbers!
    return [index, year, height];
  }
)
```

You **couldn't** type this function with regular parameter types because the **JavaScript** syntax doesn't support *leading rest elements*:

```
declare function zipWith<I, O>(
  ...arrays: I[][], /* ~~~
   ^ ❌ A rest parameter must be last in a parameter list */
  fn: (...values: I[]) => O
): O[];
```

The good news is that they are supported at the type level!

▶ 🤔 **What if my arrays contain values of different types?**

▶ 🤔 **What does the implementation of `zipWith` look like?**

Summary

In this chapter, we learned about the true arrays of the type level — **Tuples**. We have seen how to create them, how to read their content, and how to merge them to form bigger tuples!

We have also talked about `Array` types, which represent arrays with an **unknown number of values all sharing the same type**. Arrays and tuples are **complementary** — we can mix them together in variadic tuples.

Time to practice! 💪

As always, let's finish with a few challenges to put our new skills to work!

Challenge Solution Format ✨ Reset

```
/**
 * Implement a generic that returns the first type
 * in a tuple.
 *
 * Hint: How would you do it if `Tuple` was a value?
 */
namespace first {
  type First<Tuple extends any[]> = TODO

  type res1 = First<[]>;
  type test1 = Expect<Equal<res1, undefined>>;

  type res2 = First<[string]>;
  type test2 = Expect<Equal<res2, string>>;

  type res3 = First<[2, 3, 4]>;
  type test3 = Expect<Equal<res3, 2>>;

  type res4 = First<["a", "b", "c"]>;
  type test4 = Expect<Equal<res4, "a">>;
```

Challenge Solution Format ✨ Reset

```
/**
```

```
 * Implement a generic that adds a type to the end
 * of a tuple.
 */
namespace append {
  type Append<Tuple extends any[], Element> = TODO

  type res1 = Append<[1, 2, 3], 4>;
  type test1 = Expect<Equal<res1, [1, 2, 3, 4]>>;

  type res2 = Append<[], 1>;
  type test2 = Expect<Equal<res2, [1]>>;
}
```

Challenge Solution Format ✨ Reset

```
/**
 * Implement a generic that concatenates two tuples.
 */
namespace concat {
  type Concat<Tuple1 extends any[], Tuple2 extends any[]> =
    TODO

  type res1 = Concat<[1, 2, 3], [4, 5]>;
  type test1 = Expect<Equal<res1, [1, 2, 3, 4, 5]>>;

  type res2 = Concat<[1, 2, 3], []>;
  type test2 = Expect<Equal<res2, [1, 2, 3]>>;
}
```

Challenge Solution Format ✨ Reset

```
/**
 * Implement a generic taking a tuple and returning
 * an array containing the union of all values in this tuple.
 */
namespace tupleToArray {
  type TupleToArray<Tuple extends any[]> = TODO

  type res1 = TupleToArray<[1, 2, 3]>;
  type test1 = Expect<Equal<res1, (1 | 2 | 3)[]>>;

  type res2 = TupleToArray<[number, string]>;
  type test2 = Expect<Equal<res2, (number | string)[]>>;

  type res3 = TupleToArray<[]>;
  type test3 = Expect<Equal<res3, never[]>>;

  type res4 = TupleToArray<[1] | [2] | [3]>;
  type test4 = Expect<Equal<res4, (1 | 2 | 3)[]>>;
}
```

Challenge Solution Format ✨ Reset

```
/**
 * Create a generic `NonEmptyArray` type that represents
 * Arrays that contain at least one element.
 */
namespace nonEmptyArray {
  type NonEmptyArray<T> = TODO

  function sendMail(addresses: NonEmptyArray<string>) {
    /* ... */
  }
  sendMail(["123 5th Ave"]); // ✅
  sendMail(["75 rue Quincampoix", "75003 Paris"]); // ✅
  // @ts-expect-error
  sendMail([]);
  //       ^ ❌ `[]` isn't assignable to `NonEmptyList<string>`
}
```

Challenge Solution Format ✨ Reset

```
/**
 * Implement a generic that gets the length
 * of a tuple type.
 *
 * Hint:
 * How would you get the length of an array in JavaScript?
 * The type-level version is very similar :)
 */
namespace length {
  type Length<Tuple extends any[]> = TODO

  type res1 = Length<[]>;
  type test1 = Expect<Equal<res1, 0>>;

  type res2 = Length<[any]>;
  type test2 = Expect<Equal<res2, 1>>;

  type res3 = Length<[any, any]>;
  type test3 = Expect<Equal<res3, 2>>;
```

Challenge Solution Format ✨ Reset

```
/**
 * Implement a generic that gets the length
 * of a tuple type, and adds one to it.
 *
 * This challenge may seem a bit random, but
 * this is actually the basis of representing
 * numbers and doing arithmetics at the type level!
 */
namespace lengthPlusOne {
  type LengthPlusOne<Tuple extends any[]> = TODO

  type res1 = LengthPlusOne<[]>;
  type test1 = Expect<Equal<res1, 1>>;

  type res2 = LengthPlusOne<[any]>;
  type test2 = Expect<Equal<res2, 2>>;

  type res3 = LengthPlusOne<[any, any]>;
  type test3 = Expect<Equal<res3, 3>>;
```

If you enjoyed this free chapter of Type-Level TypeScript, consider supporting it by becoming a member!

Support Type-Level TypeScript!

Enrollment is **temporarily closed**. Subscribe to the newsletter to get notified when the course is available again!

```
firstName?: string
```

```
email: `${string}@${string}.${string}`
```

SUBSCRIBE

Only receive Type-Level TypeScript content. Unsubscribe anytime.

⟸ Previous
3. Objects & Records FREE

Next ⟹
5. Code branching with Conditional Types