**Types and type aliases**

type is a keyword in TypeScript that we can use to define the shape of data. The basic types in TypeScript include:

* String
* Boolean
* Number
* Array
* [Tuple](https://blog.logrocket.com/use-cases-named-tuples-typescript/)
* [Enum](https://blog.logrocket.com/typescript-string-enums-guide/)
* Advanced types

Each has unique features and purposes, allowing developers to choose the appropriate one for their particular use case.

Type aliases in TypeScript mean “a name for any type.” They provide a way of creating new names for existing types. Type aliases don’t define new types; instead, they provide an alternative name for an existing type.  
Type aliases can be created using the type keyword, referring to any valid TypeScript type, including primitive types.

type MyNumber = number;

type User = {

id: number;

name: string;

email: string;

}

In the above example, we create two type aliases: MyNumber and User. We can use MyNumber as shorthand for a number type and use User type aliases to represent the type definition of a user.

When we say “types versus interfaces,” we refer to “type *aliases* versus interfaces.” For example, you can create the following aliases:

type ErrorCode = string | number;

type Answer = string | number;

The two type aliases above represent alternative names for the same union type: string | number. While the underlying type is the same, the different names express different intents, which makes the code more readable.

**Interfaces in TypeScript**

In TypeScript, an interface defines a contract that an object must adhere to. Below is an example:

interface Client {

name: string;

address: string;

}

We can express the same Client contract definition using type annotations:

type Client = {

name: string;

address: string;

};

**Differences between types and interfaces**

For the above case, we can use either type or interface. But there are some scenarios in which using type instead of interface makes a difference.

**Primitive types**

Primitive types are inbuilt types in TypeScripts. They include number, string, boolean, null, and undefined types.  
We can use define a type alias for a primitive type as below:

type Address = string;

We often combine primitive type with union type to define a type alias, to make the code more readable:

type NullOrUndefined = null | undefined;

But, we can’t use an interface to alias a primitive type. The interface can only be used for an object type.  
Therefore, when we need to define a primitive type alias, we use type.

**Union types**

Union types allow us to describe values that can be one of several types and create unions of various primitive, literal, or complex types:

type Transport = 'Bus' | 'Car' | 'Bike' | 'Walk';

Union type can only be defined using type. There is no equivalent to a union type in an interface. But, it is possible to create a new union type from two interfaces, like so:

interface CarBattery {

power: number;

}

interface Engine {

type: string;

}

type HybridCar = Engine | CarBattery;

**Function types**

In TypeScript, a function type represents a function’s type signature. Using the type alias, we need to specify the parameters and the return type to define a function type:

type AddFn = (num1: number, num2:number) => number;

We can also use an interface to represent the function type:

interface IAdd {

(num1: number, num2:number): number;

}

Both type and interface similarly define function types, except for a subtle syntax difference of interface using : vs. => when using type. Type is preferred in this case because it’s shorter and thus easier to read.

Another reason to use type for defining a function type is its capabilities that the interface lacks. When the function becomes more complex, we can take advantage of the advanced type features such as conditional types, mapped types, etc. Here’s an example:

type Car = 'ICE' | 'EV';

type ChargeEV = (kws: number)=> void;

type FillPetrol = (type: string, liters: number) => void;

type RefillHandler<A extends Car> = A extends 'ICE' ? FillPetrol : A extends 'EV' ? ChargeEV : never;

const chargeTesla: RefillHandler<'EV'> = (power) => {

// Implementation for charging electric cars (EV)

};

const refillToyota: RefillHandler<'ICE'> = (fuelType, amount) => {

// Implementation for refilling internal combustion engine cars (ICE)

};

Here, we define a type RefillHander with conditional type and union type. It provides a unified function signature for EV and ICE handlers in a type-safe manner. We can’t achieve the same with the interface as it doesn’t have the equivalent of conditional and union types.

**Declaration merging**

[Declaration merging](https://www.typescriptlang.org/docs/handbook/declaration-merging.html) is a feature that is exclusive to interfaces. With declaration merging, we can define an interface multiple times, and the TypeScript compiler will automatically merge these definitions into a single interface definition.

In the following example, the two Client interface definitions are merged into one by the TypeScript compiler, and we have two properties when using the Client interface:

interface Client {

name: string;

}

interface Client {

age: number;

}

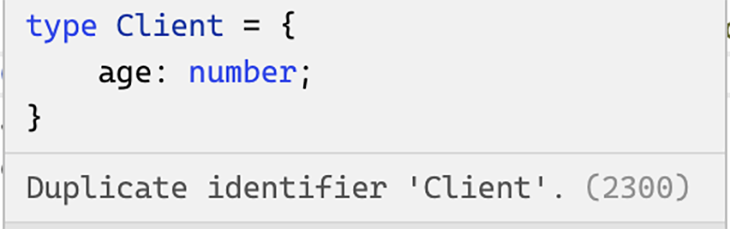
const harry: Client = {

name: 'Harry',

age: 41

}

Type aliases can’t be merged in the same way. If you try to define the Clienttype more than once, as in the above example, an error will be thrown:



When used in the right places, declaration merging can be very useful. One common use case for declaration merging is to extend a third-party library’s type definition to fit the needs of a particular project.

If you need to merge declarations, interfaces are the way to go.

**Handling conflicts when extending**

Another difference between types and interfaces is how conflicts are handled when you try to extend from one with the same property name.

When extending interfaces, the same property key isn’t allowed, as in the example below:

interface Person {

getPermission: () => string;

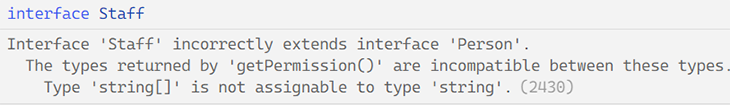
}

interface Staff extends Person {

getPermission: () => string[];

}

An error is thrown because a conflict is detected.



Type aliases handle conflicts differently. In the case of a type alias extending another type with the same property key, it will automatically merge all properties instead of throwing errors.

In the following example, the intersection operator merges the method signature of the two getPermission declarations, and a [typeof operator](https://blog.logrocket.com/how-to-use-keyof-operator-typescript/) is used to narrow down the union type parameter so that we can get the return value in a type-safe way:

type Person = {

getPermission: (id: string) => string;

};

type Staff = Person & {

getPermission: (id: string[]) => string[];

};

const AdminStaff: Staff = {

getPermission: (id: string | string[]) =>{

return (typeof id === 'string'? 'admin' : ['admin']) as string[] & string;

}

}

It is important to note that the type intersection of two properties may produce unexpected results. In the example below, the name property for the extended type Staff becomes never, since it can’t be both string and number at the same time:

type Person = {

name: string

};

type Staff = person & {

name: number

};

// error: Type 'string' is not assignable to type 'never'.(2322)

const Harry: Staff = { name: 'Harry' };

In summary, interfaces will detect property or method name conflicts at compile time and generate an error, whereas type intersections will merge the properties or methods without throwing errors. Therefore, if we need to overload functions, type aliases should be used.

**Implementing classes using interfaces or type aliases**

In TypeScript, we can implement a class using either an interface or a type alias:

interface Person {

name: string;

greet(): void;

}

class Student implements Person {

name: string;

greet() {

console.log('hello');

}

}

type Pet = {

name: string;

run(): void;

};

class Cat implements Pet {

name: string;

run() {

console.log('run');

}

}

As shown above, both interfaces and type aliases can be used to implement a class similarly; the only difference is that we can’t implement a union type.

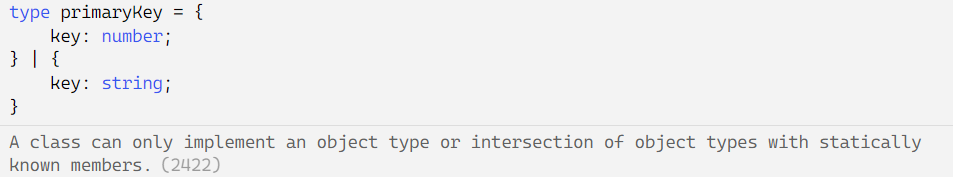
type primaryKey = { key: number; } | { key: string; };

// can not implement a union type

class RealKey implements primaryKey {

key = 1

}



In the above example, the TypeScript compiler throws an error because a class represents a specific data shape, but a union type can be one of several data types.