Union and Intersection Types

This lesson talks about the two most basic ways of composing types, union and intersection type operators.

WE'LL COVER THE FOLLOWING

Overview

The primary way to compose types in TypeScript is via *union* and *intersection* type operations.

We've already seen union types in action in this course. I assume that you're more or less familiar with them, given that they're ubiquitous in TypeScript. In this lesson, I'd like you to gain an in-depth understanding of union types.

Have you ever wondered where these names come from? While you might have some intuition about what a union of two types is, the intersection is usually not understood well.

To properly understand this lesson, it's important that you're familiar with the idea of treating types as mathematical sets. Please read this section if you haven't yet.

Simple union types

Union type is very often used with either `null` or `undefined`.

```
const sayHello = (name: string | undefined) => { /* ... */ };
```

For example, the type of `name` here is `string | undefined` which means that either a `string` OR an `undefined` value can be passed to `sayHello`.

```
sayHello("milosz");
sayHello(undefined);
```

Looking at the example, you can intuit that a union of types `A` and `B` is a type that accepts both `A` and `B` values.

Union and intersection of object types

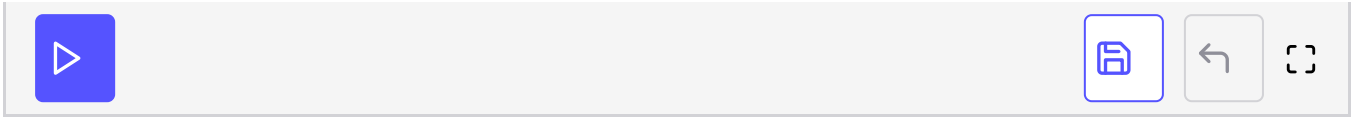
This intuition also works for complex types.

```
interface Foo {
    foo: string;
    xyz: string;
}

interface Bar {
    bar: string;
    xyz: string;
}

const sayHello = (obj: Foo | Bar) => { /* ... */ };

sayHello({ foo: "foo", xyz: "xyz" });
sayHello({ bar: "bar", xyz: "xyz" });
```



Run the code to see that there are not compile errors.

`Foo | Bar` is a type that has either all required properties of `Foo` OR all required properties of `Bar`. Inside the `sayHello` function, it's only possible to access `obj.xyz` because it's the only property that is included in both types.

What about the intersection of `Foo` and `Bar`, though?

```
const sayHello = (obj: Foo & Bar) => { /* ... */ };

sayHello({ foo: "foo", bar: "bar", xyz: "xyz" });
```

Now `sayHello` requires the argument to have both `foo` AND `bar` properties.

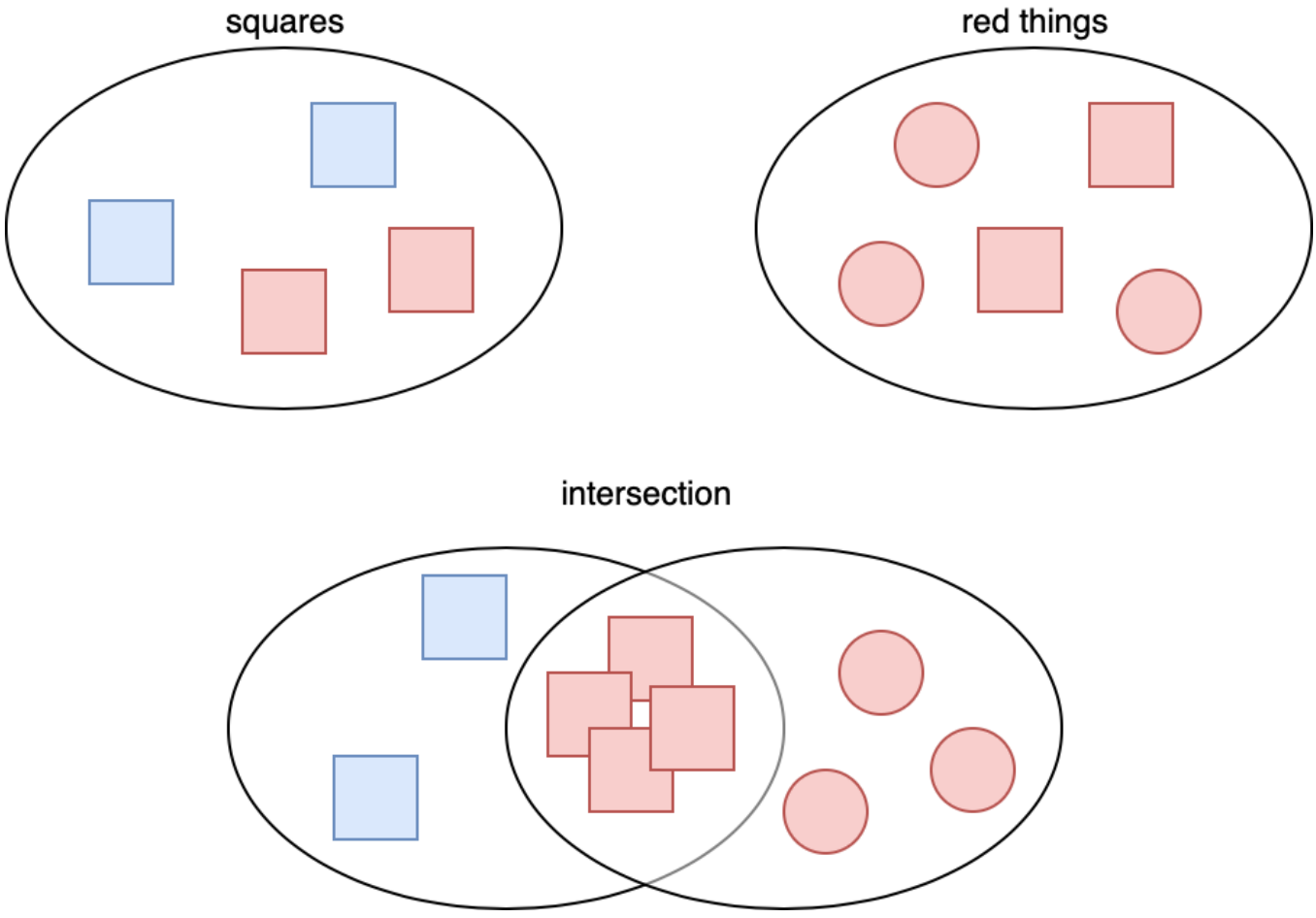
Inside `sayHello` it's possible to access `obj.foo`, `obj.bar`, and `obj.xyz`.

Hmm, but what does it have to do with *intersection*? One could argue that since `obj` has properties of both `Foo` and `Bar`, it sounds more like a union of properties and not an intersection. Similarly, a union of two object types gives you a type that only has the intersection of properties of constituent types.

It sounds confusing. I even stumbled upon a GitHub issue in the TypeScript repository ranting about the naming of these types. To understand the naming better we need to look at types from a different perspective.

Understanding union and intersection types

The confusion disappears if you look at union and intersection types from the perspective of set theory. In set theory, union and intersection are names of operations on sets. The union of two sets is a set that contains all elements from both sets. The intersection of two sets is a set that only contains elements that are present in both sets.



Armed with this knowledge, you're ready to understand the meaning of union and intersection types.

Union type `A | B` represents a set that is a union of the set of values associated with type `A` and the set of values associated with type `B`.

Intersection type `A & B` represents a set that is an intersection of the set of values associated with type `A` and the set of values associated with type `B`.

Therefore, `Foo | Bar` represents **a union** of the set of objects having `foo` and `xyz` properties and the set of objects having `bar` and `xyz`. Objects belonging to such sets all have the `xyz` property. Some of them have `foo` property and others have `bar` property.

`Foo & Bar` represents **an intersection** of the set of objects having `foo` and `xyz` properties and the set of objects having `bar` and `xyz`. In other words, the set contains objects that belong to the sets represented by both `Foo` and `Bar`. Only objects that have all three properties (`foo`, `bar` and `xyz`) belong to the intersection.

Real-world example of intersection type

Union types are quite widespread so let's focus on an example of an intersection type.

In React, when you declare a class component, you can parameterize it with the type of its properties:

```
class Counter extends Component<CounterProps> { /* ... */ }
```

Inside the class, you can access the properties via `this.props`. However, the type of `this.props` is not simply `CounterProps`, but:

```
Readonly<CounterProps> & Readonly<{ children?: ReactNode; }>
```

The reason for this is that React components can accept children elements:

```
<Counter><span>Hello</span></Counter>
```

The children element trees are accessible to the component via the `children` prop. The type of `this.props` reflects that. It's an intersection of (read-only) `CounterProps` and a (read-only) object type with an optional `children` property.

In terms of sets, it's an intersection of the set of objects that have properties as defined in `CounterProps` and the set of objects that have the optional `children` property. The result is a set of objects that have all the properties of `CounterProps` and the optional `children` property.

The next lesson introduces discriminated union types, a very powerful special case of union types.