TypeScript Template Literals

Template literals understandable

Motivation

Template literal types `build on` **`string literal types`**, and have the ability to **`expand into many strings`** via unions.

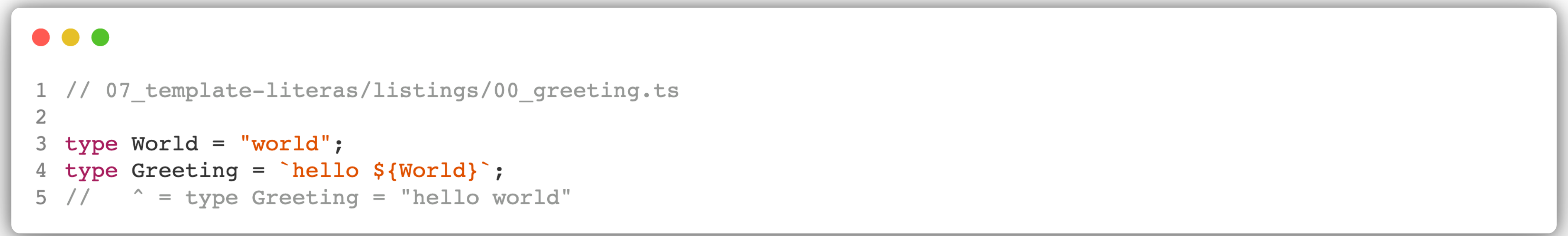
Template Literal Types

We have already seen template literal types during the basics chapter.

Since they have some quite interesting aspects, let us talk about them a little more.

Template Literal Types

Following a simple example of a template literal type:



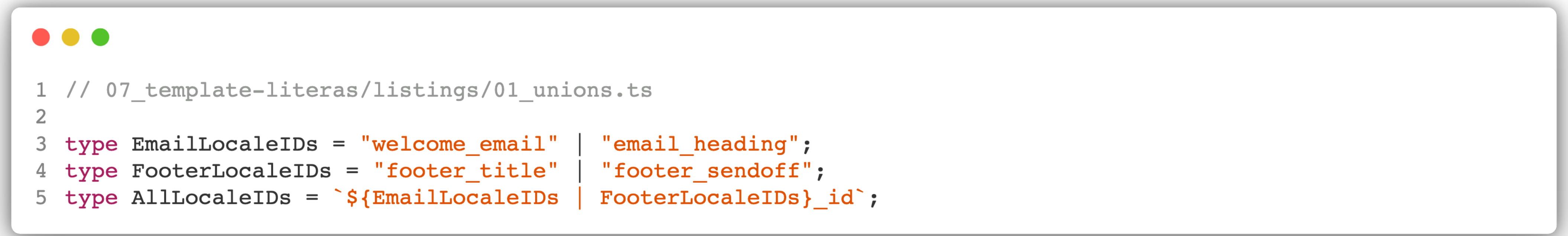
```typescript
// 07_template-literas/listings/00_greeting.ts

type World = "world";
type Greeting = `hello ${World}`;
//   ^ = type Greeting = "hello world"
```

Please keep in mind, this is only a type.

Unions

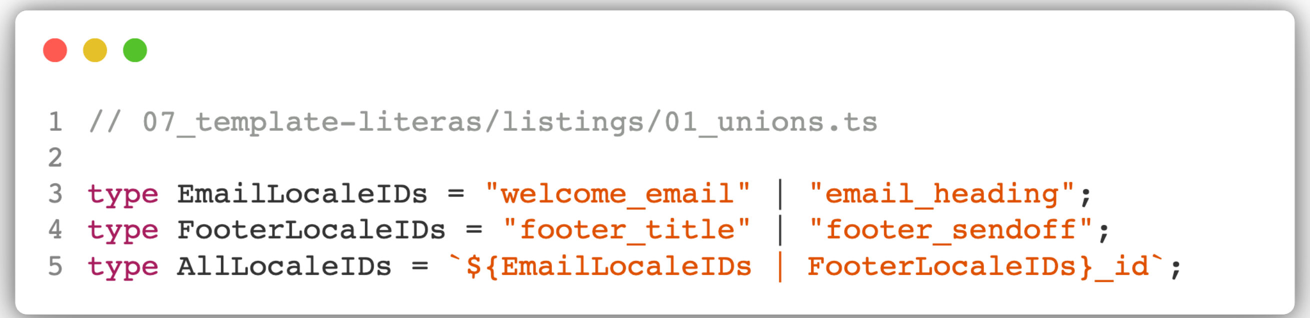
The former example does not add much use and is better done with **`enums`**.
But sometimes you have a whole dictionary of words,
`too complicated for enums`.

Here you can use **`Template Literal Types together with unions`**:



```typescript
// 07_template-literas/listings/01_unions.ts

type EmailLocaleIDs = "welcome_email" | "email_heading";
type FooterLocaleIDs = "footer_title" | "footer_sendoff";
type AllLocaleIDs = `${EmailLocaleIDs | FooterLocaleIDs}_id`;
```

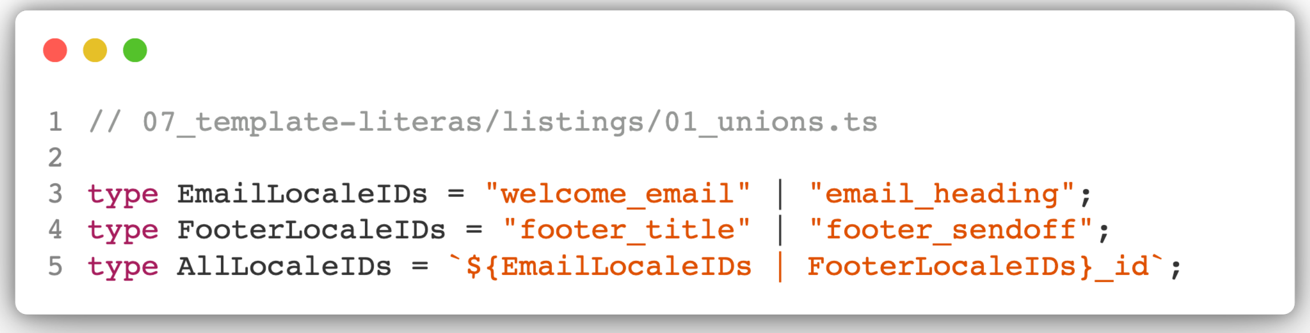
Unions



```ts
// 07_template-literas/listings/01_unions.ts

type EmailLocaleIDs = "welcome_email" | "email_heading";
type FooterLocaleIDs = "footer_title" | "footer_sendoff";
type AllLocaleIDs = `${EmailLocaleIDs | FooterLocaleIDs}_id`;
```

Question: What would be acceptable for **`AllLocaleIDs`**?

Unions



```typescript
// 07_template-literas/listings/01_unions.ts

type EmailLocaleIDs = "welcome_email" | "email_heading";
type FooterLocaleIDs = "footer_title" | "footer_sendoff";
type AllLocaleIDs = `${EmailLocaleIDs | FooterLocaleIDs}_id`;
```

Question: What would be acceptable for **`AllLocaleIDs`**?

Solution: **`welcome_email_id`** or **`email_heading_id`** or **`footer_title_id`** or **`footer_sendoff_id`**.

Unions

For each interpolated position in the template literal, the unions are cross multiplied:

```typescript
// 07_template-literas/listings/02_unions_crossproduct.ts

type AllLocaleIDs = `${EmailLocaleIDs | FooterLocaleIDs}_id`;
type Lang = "en" | "de" | "fr";
type LocaleMessageIDs = `${Lang}_${AllLocaleIDs}`;
```

We will get en_welcome_email_id, en_email_heading, en_footer_title and en_footer_sendoff for each lang.

String Union in Types

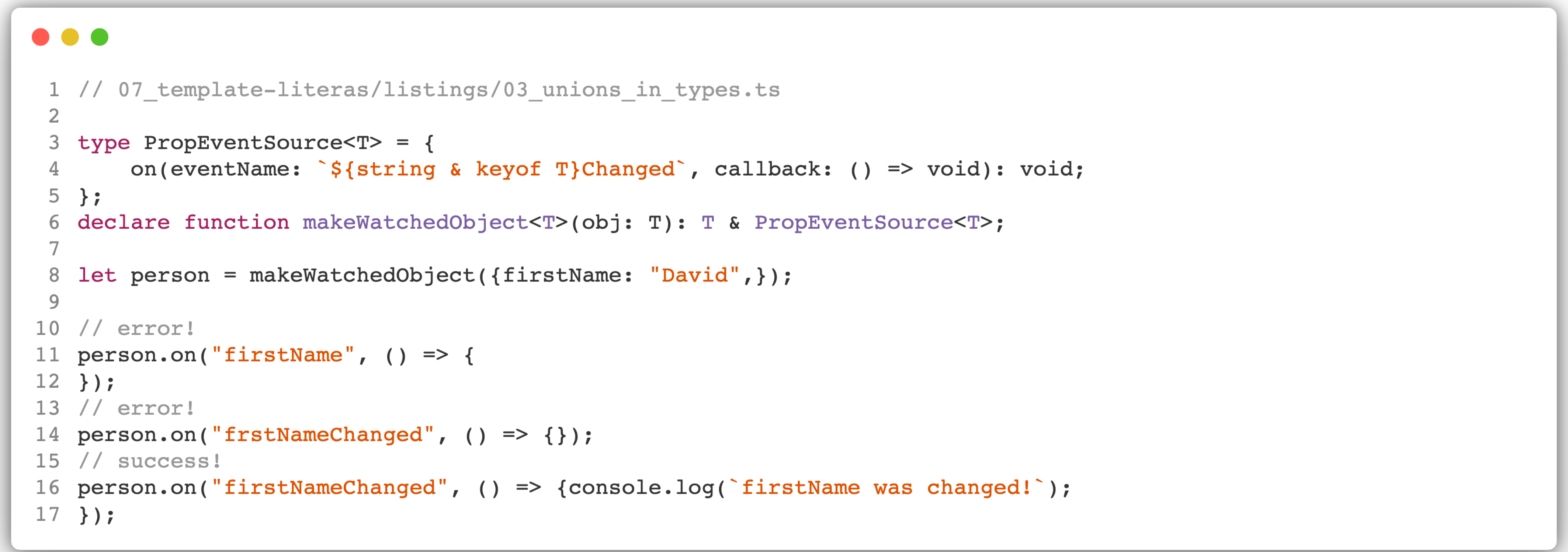
The power in template literals comes when defining a new string based off an existing string inside a type.

Very common:

- extend an object based on the fields that it currently has.

String Union in Types

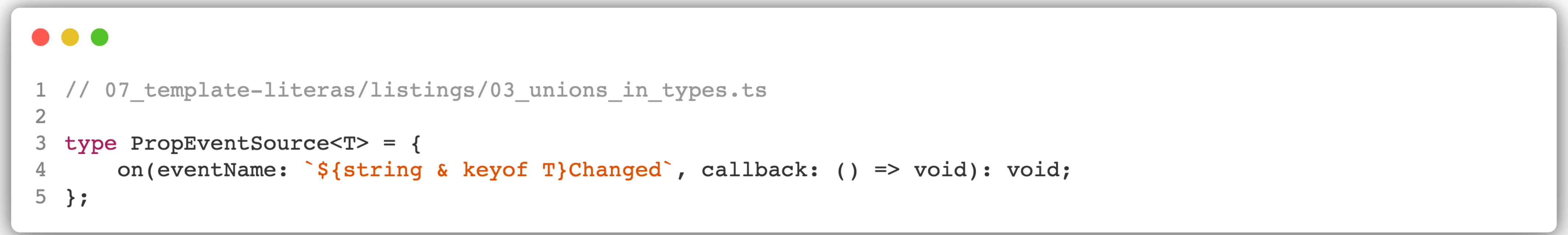
Please have a look at the following example:



```typescript
// 07_template-literas/listings/03_unions_in_types.ts

type PropEventSource<T> = {
    on(eventName: `${string & keyof T}Changed`, callback: () => void): void;
};
declare function makeWatchedObject<T>(obj: T): T & PropEventSource<T>;

let person = makeWatchedObject({firstName: "David",});

// error!
person.on("firstName", () => {
});
// error!
person.on("frstNameChanged", () => {});
// success!
person.on("firstNameChanged", () => {console.log(`firstName was changed!`);
});
```

String Union in Types

Let us have a deeper insight into what happens:



```typescript
// 07_template-literas/listings/03_unions_in_types.ts

type PropEventSource<T> = {
    on(eventName: `${string & keyof T}Changed`, callback: () => void): void;
};
```

We first create a type that enforces the later change to be named ...Changed and being a void callback.

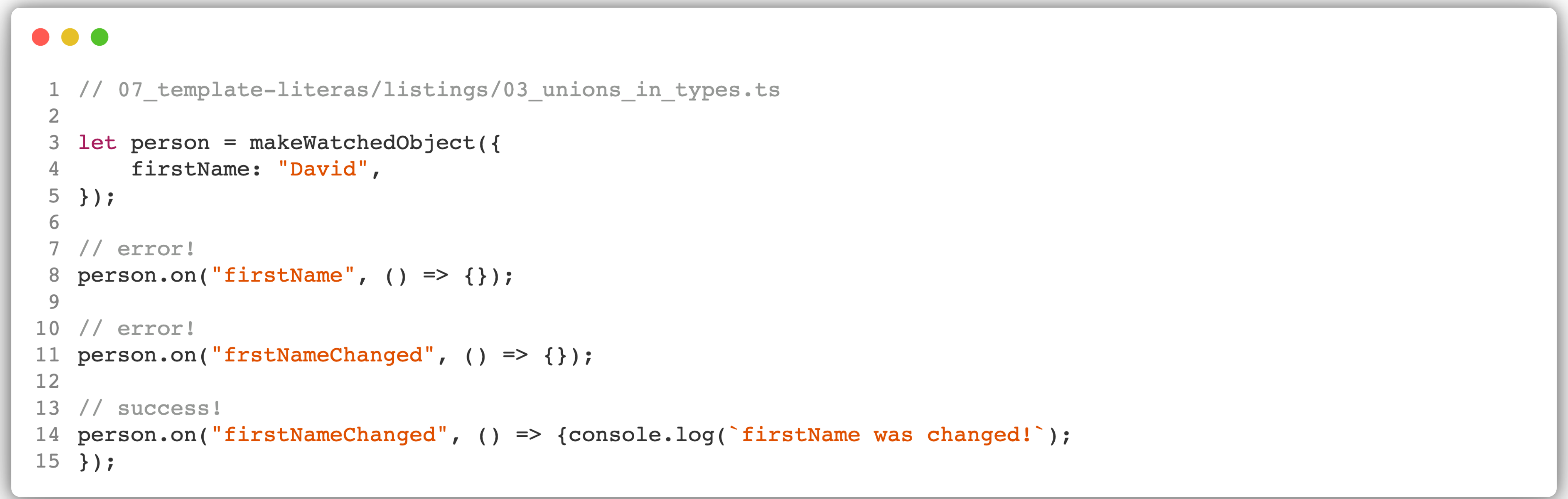
String Union in Types

Then we create a "watched object" with an 'on' method, so that we can watch for changes to properties.

```typescript
// 07_template-literas/listings/03_unions_in_types.ts

declare function makeWatchedObject<T>(obj: T): T & PropEventSource<T>;
```

String Union in Types

After that we need an object (person), and some .on methods



```ts
// 07_template-literas/listings/03_unions_in_types.ts

let person = makeWatchedObject({
    firstName: "David",
});

// error!
person.on("firstName", () => {});

// error!
person.on("frstNameChanged", () => {});

// success!
person.on("firstNameChanged", () => {console.log(`firstName was changed!`);
});
```

The only one which fits with our expectation is the last one.

Great Job!

That was all for template literals!

End

That was all for this chapter