The problem is that `Person` and `string` are being interpreted in a value context. You're trying to create a variable named `Person` and two variables named `string`. Instead, you should separate the types and values:

```
function email(
  {person, subject, body}: {person: Person, subject: string, body: string}
) {
  // ...
}
```

This is significantly more verbose, but in practice you may have a named type for the parameters or be able to infer them from context (Item 26).

While the similar constructs in type and value can be confusing at first, they're eventually useful as a mnemonic once you get the hang of it.

Things to Remember

- Know how to tell whether you're in type space or value space while reading a TypeScript expression. Use the TypeScript playground to build an intuition for this.
- Every value has a type, but types do not have values. Constructs such as `type` and `interface` exist only in the type space.
- `"foo"` might be a string literal, or it might be a string literal type. Be aware of this distinction and understand how to tell which it is.
- `typeof`, `this`, and many other operators and keywords have different meanings in type space and value space.
- Some constructs such as `class` or `enum` introduce both a type and a value.

Item 9: Prefer Type Declarations to Type Assertions

TypeScript seems to have two ways of assigning a value to a variable and giving it a type:

```
interface Person { name: string };

const alice: Person = { name: 'Alice' };  // Type is Person
const bob = { name: 'Bob' } as Person;  // Type is Person
```

While these achieve similar ends, they are actually quite different! The first (`alice: Person`) adds a *type declaration* to the variable and ensures that the value conforms to the type. The latter (`as Person`) performs a *type assertion*. This tells TypeScript that, despite the type it inferred, you know better and would like the type to be `Person`.

In general, you should prefer type declarations to type assertions. Here's why:

```
const alice: Person = {};
   // ~~~~~ Property 'name' is missing in type '{}'
   //       but required in type 'Person'
const bob = {} as Person;  // No error
```

The type declaration verifies that the value conforms to the interface. Since it does not, TypeScript flags an error. The type assertion silences this error by telling the type checker that, for whatever reason, you know better than it does.

The same thing happens if you specify an additional property:

```
const alice: Person = {
  name: 'Alice',
  occupation: 'TypeScript developer'
// ~~~~~~~~~ Object literal may only specify known properties
//           and 'occupation' does not exist in type 'Person'
};
const bob = {
  name: 'Bob',
  occupation: 'JavaScript developer'
} as Person;  // No error
```

This is excess property checking at work (Item 11), but it doesn't apply if you use an assertion.

Because they provide additional safety checks, you should use type declarations unless you have a specific reason to use a type assertion.

You may also see code that looks like `const bob = <Person>{}`. This was the original syntax for assertions and is equivalent to `{} as Person`. It is less common now because `<Person>` is interpreted as a start tag in *.tsx* files (TypeScript + React).

It's not always clear how to use a declaration with arrow functions. For example, what if you wanted to use the named `Person` interface in this code?

```
const people = ['alice', 'bob', 'jan'].map(name => ({name}));
// { name: string; }[]... but we want Person[]
```

It's tempting to use a type assertion here, and it seems to solve the problem:

```
const people = ['alice', 'bob', 'jan'].map(
  name => ({name} as Person)
); // Type is Person[]
```

But this suffers from all the same issues as a more direct use of type assertions. For example:

```
const people = ['alice', 'bob', 'jan'].map(name => ({} as Person));
// No error
```

So how do you use a type declaration in this context instead? The most straightforward way is to declare a variable in the arrow function:

```
const people = ['alice', 'bob', 'jan'].map(name => {
  const person: Person = {name};
  return person
}); // Type is Person[]
```

But this introduces considerable noise compared to the original code. A more concise way is to declare the return type of the arrow function:

```
const people = ['alice', 'bob', 'jan'].map(
  (name): Person => ({name})
); // Type is Person[]
```

This performs all the same checks on the value as the previous version. The parentheses are significant here! `(name): Person` infers the type of `name` and specifies that the returned type should be `Person`. But `(name: Person)` would specify the type of `name` as `Person` and allow the return type to be inferred, which would produce an error.

In this case you could have also written the final desired type and let TypeScript check the validity of the assignment:

```
const people: Person[] = ['alice', 'bob', 'jan'].map(
  (name): Person => ({name})
);
```

But in the context of a longer chain of function calls it may be necessary or desirable to have the named type in place earlier. And it will help flag errors where they occur.

So when *should* you use a type assertion? Type assertions make the most sense when you truly do know more about a type than TypeScript does, typically from context that isn't available to the type checker. For instance, you may know the type of a DOM element more precisely than TypeScript does:

```
document.querySelector('#myButton').addEventListener('click', e => {
  e.currentTarget // Type is EventTarget
  const button = e.currentTarget as HTMLButtonElement;
  button // Type is HTMLButtonElement
});
```

Because TypeScript doesn't have access to the DOM of your page, it has no way of knowing that `#myButton` is a button element. And it doesn't know that the `currentTarget` of the event should be that same button. Since you have information that TypeScript does not, a type assertion makes sense here. For more on DOM types, see Item 55.

You may also run into the non-null assertion, which is so common that it gets a special syntax:

```
const elNull = document.getElementById('foo');  // Type is HTMLElement | null
const el = document.getElementById('foo')!; // Type is HTMLElement
```

Used as a prefix, `!` is boolean negation. But as a suffix, `!` is interpreted as an assertion that the value is non-null. You should treat `!` just like any other assertion: it is erased during compilation, so you should only use it if you have information that the type checker lacks and can ensure that the value is non-null. If you can't, you should use a conditional to check for the `null` case.

Type assertions have their limits: they don't let you convert between arbitrary types. The general idea is that you can use a type assertion to convert between A and B if either is a subset of the other. `HTMLElement` is a subtype of `HTMLElement | null`, so this type assertion is OK. `HTMLButtonElement` is a subtype of `EventTarget`, so that was OK, too. And `Person` is a subtype of `{}`, so that assertion is also fine.

But you can't convert between a `Person` and an `HTMLElement` since neither is a subtype of the other:

```
interface Person { name: string; }
const body = document.body;
const el = body as Person;
       // ~~~~~~~~~~~~~~ Conversion of type 'HTMLElement' to type 'Person'
       //                may be a mistake because neither type sufficiently
       //                overlaps with the other. If this was intentional,
       //                convert the expression to 'unknown' first
```

The error suggests an escape hatch, namely, using the `unknown` type (Item 42). Every type is a subtype of `unknown`, so assertions involving `unknown` are always OK. This lets you convert between arbitrary types, but at least you're being explicit that you're doing something suspicious!

```
const el = document.body as unknown as Person;  // OK
```

Things to Remember

- Prefer type declarations (`: Type`) to type assertions (`as Type`).
- Know how to annotate the return type of an arrow function.
- Use type assertions and non-null assertions when you know something about types that TypeScript does not.

Item 10: Avoid Object Wrapper Types (String, Number, Boolean, Symbol, BigInt)

In addition to objects, JavaScript has seven types of primitive values: strings, numbers, booleans, `null`, `undefined`, symbol, and bigint. The first five have been around since the beginning. The symbol primitive was added in ES2015, and bigint is in the process of being finalized.