

КОНСПЕКТ С КУРСА ПО TYPESCRIPT

В этом файле находится авторский конспект, который должен помочь студентам в освоении языка TypeScript с примерами и разными особенностями. Так же в нем очень удобно повторять пройденный материал 😊

Если вы получили этот файл не совсем законным путем, то помните, что на официальной площадке он может обновляться и дополняться. То же самое и с уроками, поддержкой студентов, ответами на вопросы и тп. Так что используйте официальные источники - это куда надежнее 😊

Успехов в учебе! С наилучшими пожеланиями, Иван

ОГЛАВЛЕНИЕ

■ Что такое TypeScript и зачем он нужен	3
■ Установка TS и запуск файлов	6
■ Базовые типы: строка, число, логическое значение	7
■ Использование системы типов в функциях	8
■ Специальный тип any	10
■ Тип never	11
■ Типы null и undefined	12
■ Типизация объектов и деструктуризация	14
■ Типизация массивов	15
■ Tuples (Кортежи)	17
■ Union (Объединение)	19
■ Примитивные литеральные типы (Literal types)	22
■ Псевдонимы типов (Type aliases)	24
■ Сужение типов (Narrowing)	20
■ Объектные литералы и аннотации функций	25
■ Более продвинутый Type и пересечение типов (Intersection)	27
■ Интерфейсы (Interfaces)	29
■ Type или Interface?	31
■ Механизм вывода типов (Type Inference)	33
■ Модификаторы свойств: optional (Property Modifiers)	35
■ Оператор Non-Null and Non-Undefined	36
■ Модификаторы свойств: readonly (Property Modifiers)	37
■ Enums	39
■ Тип Unknown	41
■ Запросы типов	43
■ Утверждение типов (Type Assertions)	44
■ Про “внутренние” типы и приведение типов	46
■ Type Guard	48
■ Перегрузка функций	50
■ Работа с DOM	51
■ Generics (Обобщения), что это и зачем нужно	52
■ Generics functions	54
■ Generics types and interfaces, constraints	56
■ Generics classes	59
■ Встроенные обобщения (Readonly, Partial, Required)	60
■ Оператор keyof	61
■ Оператор typeof и снова запросы типов	62
■ Indexed Access Types	63
■ Conditional types and infer	65
■ Mapped types, +/- операторы	67
■ Template literal types	69
■ Utility types: Pick, Omit, Extract, Record	70
■ Дополнительные вспомогательные типы (Utility types)	71
■ Дополнительные вспомогательные типы (Utility types)	71
■ Работа с запросами на сервер, Promise и JSON	72
■ Awaited	74
■ Базовая работа с классом	75
■ Конструкторы, перегрузки и дженерики	76
■ Методы, их перегрузки, getter и setter	77
■ Начальное значение и Index Signatures	78
■ Наследование классов в TS (extends)	79
■ Имплементация в классах (implements)	80
■ Модификаторы видимости свойств	81
■ Приватные поля (#, возможность в JS)	83
■ Статичные свойства и методы	84
■ this и типизация контекста	85
■ Абстрактные классы	86

ПРОБЛЕМЫ В СТАНДАРТНОМ JAVASCRIPT

это язык с динамической типизацией, а это значит, вы можете выполнять любые операции с типами, например:

```

1 console.log(typeof "55" + 5)
2 console.log(typeof (!!5))
  
```

Иногда это удобно, но и вызывает трудности:

- 👉 Чем больше кода в проекте, чем больше разработчиков - тем сложнее уследить за всеми этими манипуляциями
- 👉 Сразу сложно понять структуру кода: что будет строкой, что числом, что обязательно, что нет. Мы тратим время на выяснение этих нюансов
- 👉 Сложности при исправлении ошибок в крупном проекте: получили ошибку, исправляем её, ждем сборку проекта, опять тестируем, опять исправляем... Мы не получаем уведомления об ошибках при разработке, только в runtime *runtime ошибки - это те, которые мы видим только при запуске кода

И еще часть, где JS вам никак не помогает:

- 👉 Нет подсказок про несуществующие свойства и тп
- 👉 Нет автодополнения в коде

*и других вещей, которые бы облегчили нам жизнь. Мы говорим конкретно про язык JS, а не дополнительные плагины и утилиты

```

const student = {
  name: 'Sam',
  age: 25
}

console.log(student.)
  
```

Хотелось бы вот так

```

1 const student = {
2   name: 'Sam',
3   age: 25
4 }
5
6 console.log(student.surname)
7 console.log(student.ega)
  
```

Нет подсказок про опечатку

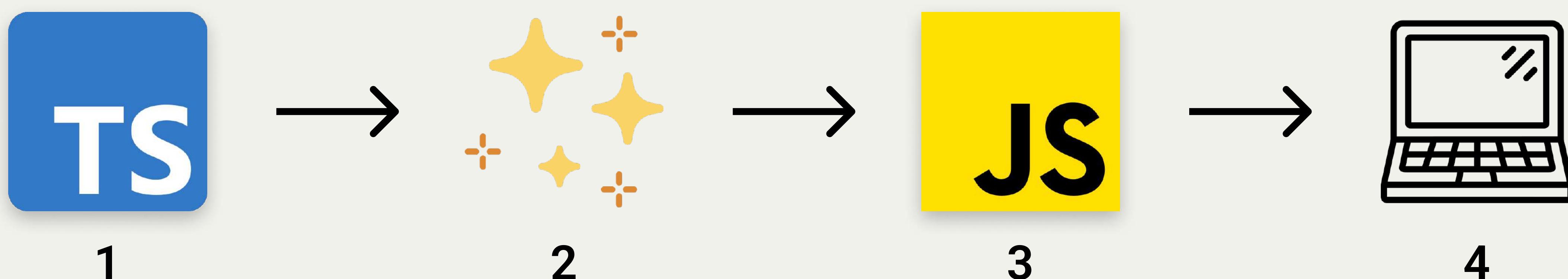
РЕШЕНИЕ ПРОБЛЕМ С ПОМОЩЬЮ TYPESCRIPT

Что дает нам язык TypeScript:

- 👉 Позволяет нам использовать “аннотации” типов. Простыми словами: говорить, что эта сущность всегда будет числом, эта - объектом, эта - массивом... Вы сразу понимаете с чем вы работаете.
- 👉 Накладывает ограничения на часть операций с разными типами. Вам проще отслеживать все действия
- 👉 За счет строгих типов код подвергается активному анализу во время разработки.
Это в свою очередь дает нам:
 - 👉 Autocomplete кода, вам автоматически будут предлагаться только доступные к использованию варианты
 - 👉 Подсказки о прямых или возможных ошибках в коде еще на этапе разработки. Вы не тратите время на поиск их в runtime

*и другое, но пока этого достаточно 😊

КАК ЭТО РАБОТАЕТ ВНУТРИ



- 1 - тут мы пишем код на TypeScript (далее просто TS).
- 2 - код отправляется в компилятор для преобразования
- 3 - на выходе получаем все тот же JS код
- 4 - этот код идет в работу (сайты, приложения, сервера...)

Внутри TS **можно** писать и на стандартном JS, без всех возможностей TS. Это не приведет к ошибкам, так как в итоге мы получим все равно JS код

ГДЕ МОЖНО ИСПОЛЬЗОВАТЬ TS



и другие...

Очень много инструментов поддерживают TypeScript, а в некоторых он обязателен для работы (Angular, Nestjs...)

ПЛЮСЫ И МИНУСЫ TS

Плюсы и зачем его желательно изучить:

- 👉 Много полезных вещей с технической стороны, это облегчает разработку
- 👉 Намного удобнее работать в команде, код документирует сам себя
- 👉 Ускоряет разработку, позволяет избегать ошибок и проблем
- 👉 Сильно облегчает масштабирование проектов
- 👉 Позволяет разработчикам понять, как устроены другие технологии и языки с использованием системы типов

Из минусов:

- 👉 Добавляет лишнюю прослойку, которая может быть не нужна небольшим проектам

УСТАНОВКА TS

Для работы нам понадобится:

- 👉 Установить Node.js, если он еще не установлен. Желательно LTS версию, рекомендованную для всех. С ней почти не бывает проблем
- 👉 Установить сам TypeScript глобально:

```
npm install -g typescript
```

*для систем MacOS и Linux не забываем ключевое слово sudo

- 👉 Дополнительно установить пакет ts-node:

```
npm install -g ts-node
```

ЗАПУСК TS ФАЙЛОВ

Для включения компилятора (TS => JS) запускаем команду:

```
tsc ${путь к файлу .ts}
```

Чтобы применить актуальные настройки tsconfig.json при компиляции index.ts запускаем команду (в будущем настроим больше файлов):

```
tsc
```

Для создания шаблона конфигурации запускаем команду:

```
tsc --init
```

Для быстрого просмотра результатов кода запускаем команду:

```
ts-node ${путь к файлу .ts}
```

СИСТЕМА ТИПОВ В TS

Простыми словами: тип – это указание того, чем является сущность и как её можно использовать.



```
1 let userName: string = `Alex`;
```

Конструкция через **: тип** называется аннотацией типа, указанием.



```
1 let userName: string = `Alex`;
2 userName = 5;
```

Теперь при попытке изменить содержимое на другой тип или присвоить другой тип изначально, TS будет предупреждать об ошибке.

Это важно по двум причинам:

- 👉 Теперь TS отслеживает эту переменную, анализирует код и четко понимает что в ней должно быть. Это не даст вам или кому-то другому переопределить её или использовать недоступный метод/свойство, а значит и не получить runtime ошибку
- 👉 Если ваш код будет использовать другой разработчик, то он сразу поймет какой тип данных ему ожидать или использовать

БАЗОВЫЕ ПРИМИТИВНЫЕ ТИПЫ

Строки: `", "", ``` - аннотация : **string**

Числа: `10, 0.5, 0.00001, -50, 4e10` - аннотация : **number**

Логический тип: `true, false` - аннотация : **boolean**

ТИПИЗАЦИЯ АРГУМЕНТОВ ФУНКЦИЙ

Первый ключевой момент любой функции - это её **аргументы**. При стандартной настройке, если не указать тип каждого аргумента - TS подскажет об ошибке. Это значит, что аргумент может быть чем угодно

А это позволяет выполнить с этим аргументом любую операцию внутри функции, а это большая вероятность получить ошибку.

Два **неправильных** способа её исправить:

- 👉 Аргументам назначить тип any (что угодно)
- 👉 В конфигурации TS поставить noImplicitAny в позицию false (tsconfig.json)

Правильный способ - указать каждому аргументу чем он может быть:



```
function logBrtMsg(isBirthday: boolean, userName: string, age: number) {  
    if (isBirthday) {  
        return `Congrats ${userName.toUpperCase()}, age: ${age + 1}`;  
    } else {  
        return "Error";  
    }  
}
```



```
const logBrtMsg = (  
    isBirthday: boolean,  
    userName: string,  
    age: number  
) => {}
```

Теперь при вызове функции TS будет подсказывать нам, какие аргументы нужно передать и в каком виде:



```
logBrtMsg(true, "John", 40);
```

Такая аннотация типов аргументов у функции позволяет выполнять только доступные операции в теле функции, позволяет получать подсказки при её вызове и предупреждения, если вы сделали что-то не так

Это особенно полезно, когда вы работаете с чужими функциями или с теми, которые вы создали когда-то давно 😊

ТИПИЗАЦИЯ ВОЗВРАЩАЕМОГО ЗНАЧЕНИЯ

Второй ключевой момент любой функции - это **возвращаемое значение**. Если TS будет знать, что именно функция может вернуть, то он сможет вам подсказывать об ошибках в будущем, при её использовании

Указывать тип возвращаемого значения нужно после аргументов:

```
● ● ●  
1 function logBrtMsg(isBirthday: boolean, userName: string, age: number): string {  
2     if (isBirthday) {  
3         return `Congrats ${userName.toUpperCase()}, age: ${age + 1}`;  
4     } else {  
5         return "Error";  
6     }  
7 }
```

Например, эта функция всегда возвращает строку, а значит, к её результату можно применить только методы, которые есть у строк.

Таким образом любой разработчик сразу видит, что он получит в итоге, не копаясь во внутренностях функции. А TS будет предупреждать вас об ошибках. Даже если вы захотите изменить логику внутри

Если функция ничего не возвращает (вывод в консоль, отправка данных на сервер, работа с DOM-деревом...), то возвращаемый тип будет `void`

Void – это не только отсутствие возвращаемого значения, но и его полное игнорирование.

Нужно ли ставить аннотацию типа возвращаемого значения, если TS и сам это может сделать?

! Да, это позволит намного быстрее понимать логику любой функции и отлавливать ошибки, когда у вас несколько возвращаемых значений

СПЕЦИАЛЬНЫЙ ТИП ANY

Тип **any** - это дословно «любое значение». TS не может определить тип, с которым ему работать, не знает какие методы можно использовать, какие подсказки давать и когда сообщать об ошибках

Главное: старайтесь никогда не использовать этот тип.

Это приводит не только к ошибкам, но и заключению о том, что вы не умеете работать с системой типов

В JS/TS существуют операции, которые возвращают any в любом случае. Например, `JSON.parse()` Это происходит из-за того, что результат функции невозможно предсказать:

```

● ● ●
1 const userData =
2   '{"isBirthdayData": 50, "ageData": 40, "userNameData": "John"}';
3
4 const userObj = JSON.parse(userData);
5
6 console.log(userObj.smth());

```

В JSON строке может быть что угодно и в любом виде. TS не знает, есть ли там свойство или метод smth и не подскажет вам об ошибке. Он вообще не может знать, что за тип получится в итоге “парсинга”: строка, число, объект...

На такие операции нужно обращать внимание и применять к ним проверки (условия, try/catch...). Особенно, когда мы работаем со сторонними данными, например полученные после запроса к серверу.

В TS можно получить any, если создать пустую переменную. Нужно избегать такого поведения. Указывайте сразу, что за данные там будут:

```

● ● ●
1 let salary;
2 salary = 5000;

```

Неправильно

```

● ● ●
1 let salary: number;
2 salary = 5000;

```

Правильно

ТИП NEVER

Существуют функции, которые **никогда не заканчиваются возвращаемым значением**: они могут специально вернуть ошибку или “зависнуть”. В таком случае функция возвращает специальный тип **never**

Например:

```

1 const createError = (msg: string) => {
2   throw new Error(msg);
3 };
  
```

Функция вернет ошибку и никогда не дойдет до return
Но если поместить ошибку в условие, то тип уже будет не never

```

1 const createNever = () => {
2   while (true) {
3   }
4 };
5 };
  
```

Функция “зависнет” на бесконечном цикле и ничего не вернет.
Тоже самое и с бесконечной рекурсией.

Использовать нет смысла, так как скрипт просто зависнет, но знать желательно

Этот тип можно использовать для “исчерпывающей проверки”, когда мы хотим сказать коду, что в каком-то случае функция ничего не возвращать не будет. Почти всегда это разветвленные условия:

```

1 function logBrtMsg(isBirthday: boolean, userName: string, age: number): string {
2   if (isBirthday === true) {
3     return `Congrats ${userName.toUpperCase()}, age: ${age + 1}`;
4   } else if (isBirthday === false) {
5     return "Too bad";
6   }
7   return createError("Error");
8 }
  
```

Последняя ветка в условии else не прописана, а значит там может быть undefined
Но если мы вернем там never, то все будет в порядке, ведь в рантайме появится ошибка
Но обычно условия стараются прописать полностью 😊

NULL И UNDEFINED В ЦЕЛОМ

null – отсутствие чего-либо полностью

Например, несуществующая переменная:

```
console.log(smth)
```

Этот тип данных стоит использовать тогда, когда мы четко хотим сказать, что чего-то не существует: нет таких данных на сервере, нет такого продукта, нет такого пользователя и тп.

undefined – это значит что-то есть, но значения у него не определено

Например:

```
let smth;  
console.log(smth)
```

Помните, что такой код в TS даст вам тип any, а не undefined. Избегайте этого!

НЕОПРЕДЕЛЕННЫЕ ТИПЫ В ТС



```
1 const test: null = null;  
2 const test2: any = null;  
3 const test3: string = null;  
4 const test4: number = null;
```



```
1 const test5: undefined = undefined;  
2 const test6: any = undefined;  
3 const test7: string = undefined;  
4 const test8: number = undefined;
```

Первые две операции с каждой стороны будут без ошибок, остальные и подобные всегда будут давать ошибку. И это правильное поведение, не смотря ни на что. Оно позволяет избегать ошибок

Хотя и в нативном JS null и undefined можно помещать куда угодно, в TS это будет мешать анализу кода и выявлению неточностей

Такое поведение можно отключить в конфигурации компилятора
Для этого достаточно установить опцию **strictNullChecks** в false
Но! Делать так никогда не стоит, это ведет к ошибкам в коде:

```
1  function getRndData() {  
2      if (Math.random() < 0.5) {  
3          return null;  
4      } else {  
5          return " Some data ";  
6      }  
7  }  
8  
9  const data = getRndData();  
10 const trimmedData = data.trim();
```

В `data` может попасть как `null`, так и строка. С отключенной проверкой TS вам не подскажет о том, что возможна ошибка на этапе `null.trim()`
И даже если вы укажите аннотацию `const data: string`, то ничего не изменится.
TS будет продолжать думать, что в `data` строка, ведь теперь допустимо:

```
1  const test: string = null;
```

Так что не отключайте эту проверку и используйте эти типы по назначению и надобности

ОБЪЕКТЫ В TYPESCRIPT

Объект в JS – это базовая единица, но в TS как типы чаще используются другие сущности: Type, Interface... Но объекты мы можем типизировать на этапе использования

```
1 const userData = {  
2   isBirthdayData: true,  
3   ageData: 40,  
4   userNameData: "John"  
5 }
```

Сам объект

```
1 const logBrtMsg = (data: {  
2   isBirthdayData: boolean,  
3   ageData: number,  
4   userNameData: string,  
5 }): string => {}
```

Его аннотация при использовании в функции

При этом TS позволяет передавать объект **шире**, чем описан в аннотации. Простыми словами: если в userData будут еще свойства, то это не будет ошибкой. Но описанные в аннотации должны быть обязательно

ДЕСТРУКТУРИЗАЦИЯ ОБЪЕКТОВ

TS так же позволяет выполнять деструктуризацию объектов прямо на этапе аннотации аргумента функции. Доступно любое количество вложенностей, но синтаксис может стать трудным для чтения

```
1 function logBrtMsg({  
2   isBirthdayData,  
3   userNameData,  
4   ageData  
5 }: {  
6   isBirthdayData: boolean;  
7   userNameData: string;  
8   ageData: number;  
9 }): string {}
```

Теперь можно применять isBirthdayData, userNameData, ageData как отдельные переменные внутри функции

МАССИВЫ В TYPESCRIPT

Массивы в TS работают точно так же, как и в классическом JS и цель их использования такая же. Мы лишь можем указывать, какие **типы** будут в нем и какая **структура** массива у нас должна быть



```
1 const departments: string[] = ["dev", "design", "marketing"];
2 const nums: number[] = [4, 5, 6]
```

Запись `string[]`, `number[]`, `{ }[]` и тп. - это указание того, что массив состоит из типов, указанных перед `[]`

А если указаны двойные квадратные скобки `[][]` - это массив массивов с определенными данными:



```
1 const nums: number[][] = [
2   [3, 5, 6],
3   [3, 5, 6],
4 ];
```

Для создания массивов с разными типами данных внутри используются кортежи (tuples), union type или аннотация `any[]` (плохой вариант)

За счет указания типов внутри массива, TS “знает” чем является каждый его член и дает нам подсказки про недопустимые операции как с самим массивом, так и с его членами



```

1 const departments: string[] = ["dev", "design", "marketing"];
2 departments.push(5);

```

Ошибка! Нельзя “пушить” число в массив, состоящий из строк



```

1 const departments: string[] = ["dev", "design", "marketing"];
2 const department = departments[0];
3 department.parseInt();

```

Ошибка! Первый элемент - это строка, у неё не существует такого метода

Во время использования методов массива TS определяет **тип аргумента внутри callback-функций**. Это позволяет внутри тела функции применять только допустимые операции.

Tip! Лучше указывать аннотации аргументов самостоятельно, это поможет поддерживать код при будущих изменениях без ошибок



```

1 const departments: string[] = ["dev", "design", "marketing"];
2 const report = departments
3   .filter((d: string) => d !== "dev")
4   .map((d: string) => `${d} – done`);

```

Аргумент d автоматически определен как строка,
но мы дополнительно указали тип вручную



```

1 const departments: string[] = ["dev", "design", "marketing"];
2 const report = departments
3   .filter((d) => d !== "dev")
4   .map((d) => {
5     return 4;
6   })
7   .map((d) => `${d + 10} – done`);

```

Без указания типов, на 7й строке TS определит d как число, ведь в б/у мы вернули массив чисел. И операция может выполняться не так, как нужно вам.

Подсказки от TS не будут в данном случае!

TUPLES (КОРТЕЖИ) В TS

Для разных задач в коде могут использоваться разные структуры данных! Это зависит от ситуации, определенных особенностей проекта или личных предпочтений разработчика

Кортежи - это структура данных, которая существует **только** в TS. Она необходима для записи набора данных в строго определенном порядке:

```
1 const userDataTuple: [boolean, number, string] = [true, 40, "John"];
```

*На данный момент есть предложение добавить tuples в стандартный JS,
но оно пока на рассмотрении

Синтаксисом они похожи на массивы. Именно в них и преобразуются после компиляции. Но могут содержать разные типы данных, которые записываются в аннотацию **в строгом порядке**

Основной минус - мы не сразу можем понять что это за данные и к чему они относятся. Нужно иметь описание, документацию или просто помнить почему именно такой порядок данных. За счет этого данная структура может встречаться нечасто

РАБОТА С TUPLES В TS

Так как кортеж технически - это массив, то на нем работают все методы и свойства массивов. Но есть ограничения

```
● ● ●
1 const userDataTuple: [boolean, number, string] = [true, 40, "John"];
2 userDataTuple.push(50);
3 userDataTuple[3];
```

Метод сработает, но в TS коде вы не сможете обратиться к 4му элементу кортежа. Его нет в аннотации, а значит и обратиться к нему нельзя

```
● ● ●
1 const userDataTuple: [boolean, number, string] = [true, 40, "John"];
2 const res = userDataTuple.map((t) => `${t} - data`);
```

Любые методы будут работать так же, как с обычным массивом и данными внутри. Но могут быть предупреждения об операциях с типами, которые приводят к ошибкам:

```
● ● ●
1 const userDataTuple: [boolean, number, string] = [true, 40, "John"];
2 const res = userDataTuple.map((t) => `${t.toUpperCase()} - data`);
```

Здесь код даст ошибку за счет того, что не все элементы кортежа содержат метод `toUpperCase()`

Деструктуризация кортежей работает так же, как и в массивах. Причем TS автоматически “знает” какой тип будет у полученных элементов:

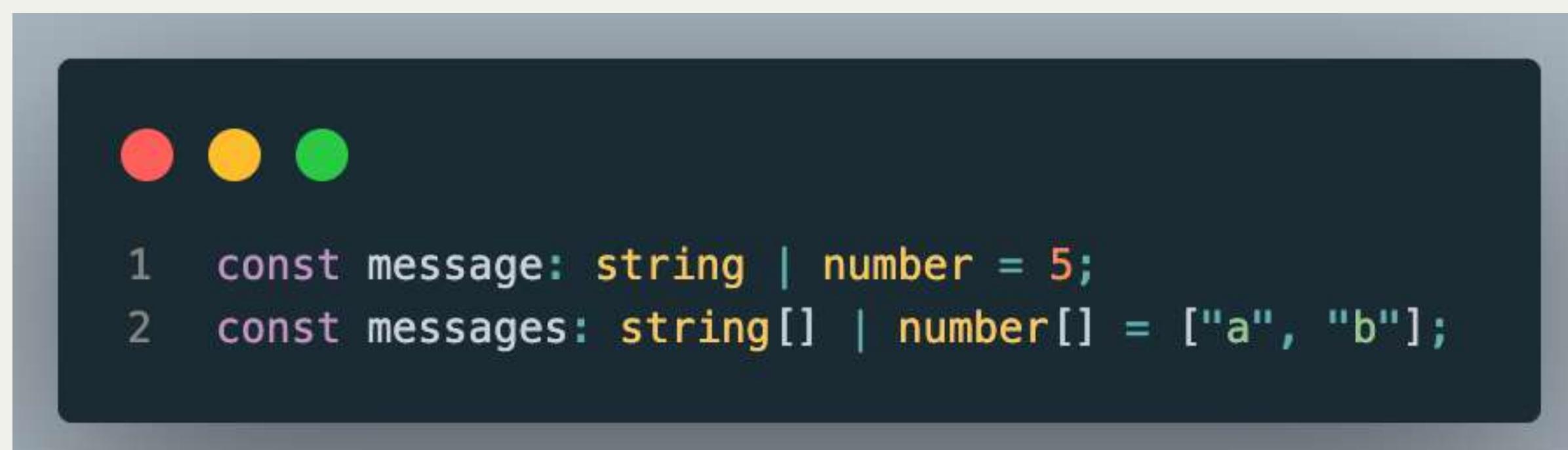
```
● ● ●
1 const userDataTuple: [boolean, number, string] = [true, 40, "John"];
2 const [bthd, age, userName] = userDataTuple;
```

Для расширения кортежей неопределенным количеством элементов используется специальный синтаксис. Можно применять в любом месте аннотации, но не более одного раза:

```
● ● ●
1 const userDataTuple: [boolean, number, ...string[]} = [true, 40, "John", "Alex", "Ann"];
2 const userDataTuple: [...boolean[], number, string] = [true, true, 40, "John"];
3 const userDataTuple: [boolean, ...number[], string] = [true, 40, 50, 60, "John"];
```

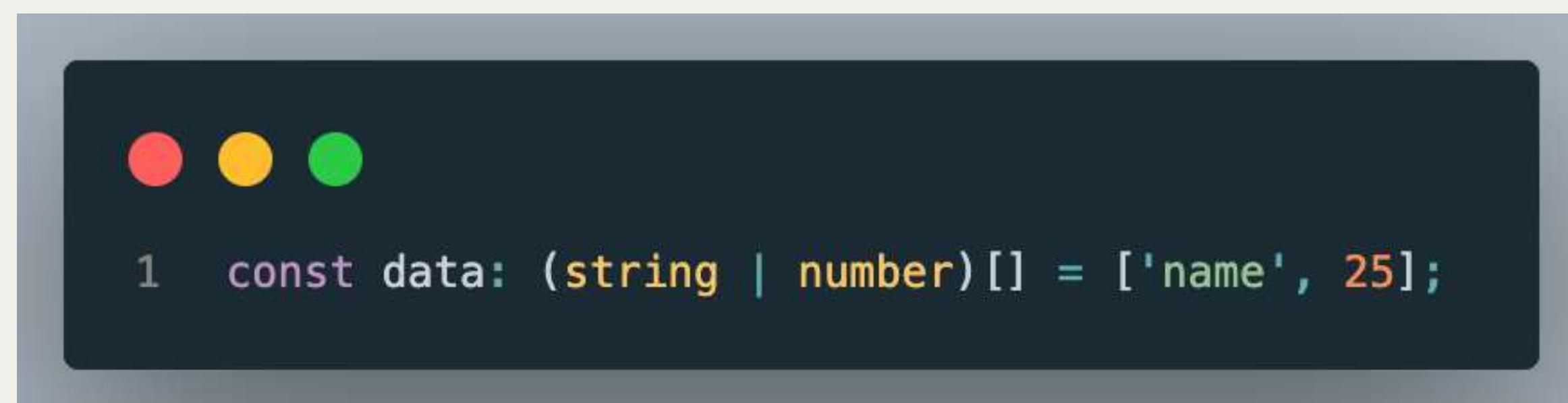
МЕХАНИЗМ ОБЪЕДИНЕНИЯ ТИПОВ

Union тип - это механизм, который позволяет объединить несколько типов, чтобы дословно сказать: эта сущность может быть или этим типом, или этим. Для этого используется специальный оператор:



```
1 const message: string | number = 5;
2 const messages: string[] | number[] = ["a", "b"];
```

Обычно мы стараемся сделать так, чтобы в массивах был только один тип данных. Но при помощи union типа мы можем объявить, что в нем есть разные типы:



```
1 const data: (string | number)[] = ['name', 25];
```

Такой синтаксис и говорит о том, что в массиве разные типы данных. Но старайтесь избегать таких ситуаций. При переборе и выполнении определенных действий с элементами разных типов можно получить ошибки. Лучше, чтобы массив был однородным

Одно из самых частых применений - это аннотация аргументов функций:



```
1 function printMsg(msg: string | number): void {
2   console.log(msg);
3 }
4 printMsg(4);
5 printMsg("string");
```

Но нужно быть аккуратными при манипуляциях с этими аргументами внутри тела функции. Вы можете использовать только операции, доступные со всеми типами в union типе. Иначе будет ошибка:



```
1 function printMsg(msg: string | number): void {
2   console.log(msg.toLowerCase());
3 }
```

Метод `toLowerCase()` существует только на строке. Если в функцию попадет число, то, будет ошибка, о чем вас предупредит TS. Для работы с такими ситуациями необходимо использовать механизм **сужения типов** (narrowing, следующий урок в курсе)

МЕХАНИЗМ СУЖЕНИЯ ТИПОВ (NARROWING)

При использовании union типа в аргументах функции может возникнуть ситуация, когда мы хотим выполнить разные операции в зависимости от типа данных. Для этого нам нужно “сузить типы”, в простейшем варианте с помощью оператора `typeof`:

```
● ● ●  
1 function printMsg(msg: string | number): void {  
2     if (typeof msg === 'string') {  
3         console.log(msg.toLowerCase());  
4     } else {  
5         console.log(msg.toExponential());  
6     }  
7     console.log(msg)  
8 }
```

Внутри первой ветки условия `msg` будет определяться как строка, во второй - как число. А значит им будут доступны разные методы. Вне условия переменная все так же будет union типом

Операций для сужения типов существует немало, например:

```
● ● ●  
1 function printMsg(msg: string | number | boolean): void {  
2     if (typeof msg === 'string' || typeof msg === 'number') {  
3         console.log(msg.toString());  
4     } else {  
5         console.log(msg);  
6     }  
7     console.log(msg)  
8 }
```

Комбинация операторов `typeof`

```
1 function printMsg(msg: string[] | number | boolean): void {
2   if (Array.isArray(msg)) {
3     msg.forEach(m => console.log(m));
4   } else if (typeof msg === "number") {
5     console.log(msg.toFixed());
6   } else {
7     console.log(msg);
8   }
9 }
```

Метод `Array.isArray()` позволяет определить массив

```
1 const printReadings = (a: number | string, b: number | boolean) => {
2   if (a === b) {
3     console.log(a, b);
4   }
5 };
```

Можно применять равенство по значениям и типам

```
1 function checkReadings(readings: { system: number } | { user: number }): void {
2   if ("system" in readings) {
3     console.log(readings.system);
4   } else {
5     console.log(readings.user);
6   }
7 }
```

Оператор `in` позволит определить, существует ли свойство в объекте

```
1 function logValue(x: string | Date) {
2   if (x instanceof Date) {
3     console.log(x.getDate());
4   } else {
5     console.log(x.trim());
6   }
7 }
```

Оператор `instanceof` позволит определить, является ли аргумент экземпляром

```
1 const printReadings2 = (a: number[] | string) => {
2   console.log(a.slice(0, 3));
3 };
```

А иногда нужная операция доступна сразу на разных типах

ПРИМИТИВНЫЕ ЛИТЕРАЛЬНЫЕ ТИПЫ

Примитивы – это простые типы данных, строки, числа, булевые значения, символы и тд.

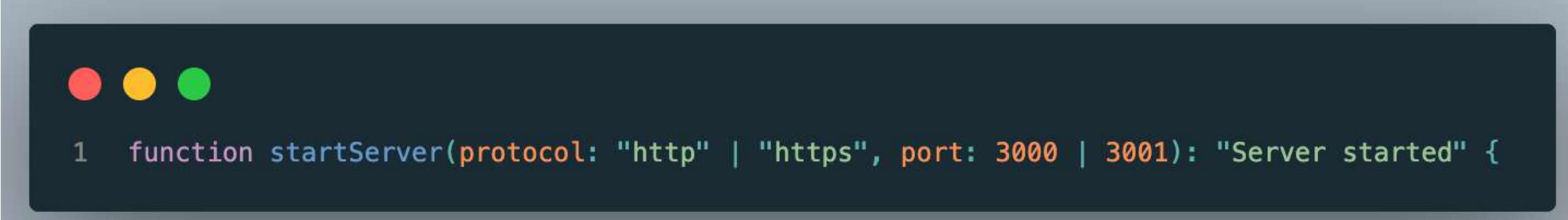
Литералы – это конкретные значения этих типов.

Примитивные литеральные типы - это типы на основании конкретных значений примитивов:



```
1 let msg: "Hello" = "Hello";
2 const salary: 5000 = 5000;
3 const truthy: true = true;
```

Теперь значение в переменной может быть только указанного типа.
Частое применение - это аргументы функций:



```
1 function startServer(protocol: "http" | "https", port: 3000 | 3001): "Server started" {
```

Теперь аргументы и результат работы функции могут иметь только несколько значений, что убережет вас от ошибок + включаются подсказки во время вызова функции

ИСПОЛЬЗОВАНИЕ ЛИТЕРАЛЬНЫХ ТИПОВ

В плане логики кода всегда ищите сущности, которые могут иметь **ограниченное количество значений**: методы запросов на сервер, порты, свойства браузера, анимаций и тп.

Но! В TS существует специальная структура - **Enum (перечисление)**. Она мощнее и служит для тех же целей. Часто предпочтение отдается именно ей. Будет дальше по курсу.

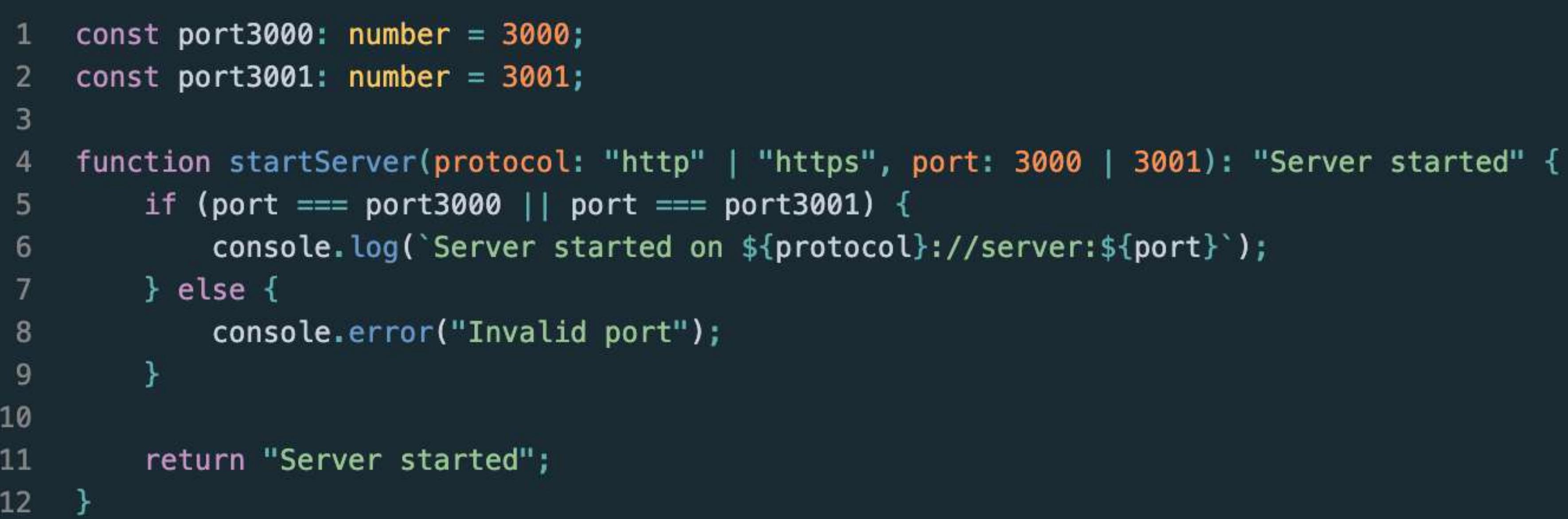
Пример аннотирования функции с ограниченным кол-вом значений:



```
function createAnimation(
  id: string | number,
  animName: string,
  timingFunc: "ease" | "ease-out" | "ease-in" = "ease",
  duration: number,
  iterCount: "infinite" | number
): void {
```

Обратите внимание на аргумент `iterCount`. Тут не перечисление, а выбор из общего или литерального типа. `Enum` тут может и не подойти

Для аргументов, которые приходят динамически в функцию, можно комбинировать аннотации, условия и константы. Так мы будем дополнительно активировать проверку и на уровне runtime кода:



```
const port3000: number = 3000;
const port3001: number = 3001;

function startServer(protocol: "http" | "https", port: 3000 | 3001): "Server started" {
  if (port === port3000 || port === port3001) {
    console.log(`Server started on ${protocol}://server:${port}`);
  } else {
    console.error("Invalid port");
  }
  return "Server started";
}
```

ПСЕВДОНИМЫ ТИПОВ (TYPE ALIASES)

Знакомство с **type** мы начнем с механизма псевдонимов. Продолжение в следующем модуле.

Иногда наши лiteralные типы могут содержать **много значений**, иногда мы хотим их **переиспользовать** в разных частях кода. В таких ситуациях хотелось бы вынести такой код в отдельную сущность по типу переменной. Этим и занимаются псевдонимы:

```
● ● ●  
1 type AnimationTimingFunc = "ease" | "ease-out" | "ease-in";  
2 type AnimationID = string | number;  
3  
4 function createAnimation(  
5   id: AnimationID,  
6   animName: string,  
7   timingFunc: AnimationTimingFunc = "ease",  
8   duration: number,  
9   iterCount: "infinite" | number  
10 ): void {
```

Название псевдонима с большой буквы

Теперь такие значения удобнее читать, можно переиспользовать и поместить любые типы в псевдоним. Их можно создавать внутри функций, внутри методов классов или объектов, внутри отдельных модулей и тп. Когда вам это нужно.

В **type** можно помещать и объекты, но об этом позже

После компиляции они исчезают, так что эта возможность существует только в TS

ОБЪЕКТНЫЕ ЛИТЕРАЛЫ

Объектные литералы почти не используются в реальной практике, но в теории их знать полезно. Так мы будем чуть глубже понимать TS и почему иногда возникают ошибки

Если в коде есть объект, который используется в дальнейшем, то он может не подходить по условиям, которые программисту кажутся очевидными на первый взгляд:

```
● ● ●  
1 const serverConfig = {  
2   protocol: "https",  
3   port: 3001,  
4 };  
5  
6 const startServer = (  
7   protocol: "http" | "https",  
8   port: 3000 | 3001  
9 ): "Server started" => {  
10   console.log(`Server started on ${protocol}:${port}`);  
11  
12   return "Server started";  
13 };  
14  
15 startServer(serverConfig.protocol, serverConfig.port);
```

TS покажет ошибку, ведь в `serverConfig.protocol` или `serverConfig.port` может лежать все, что угодно. А функция принимает только определенные значения на вход

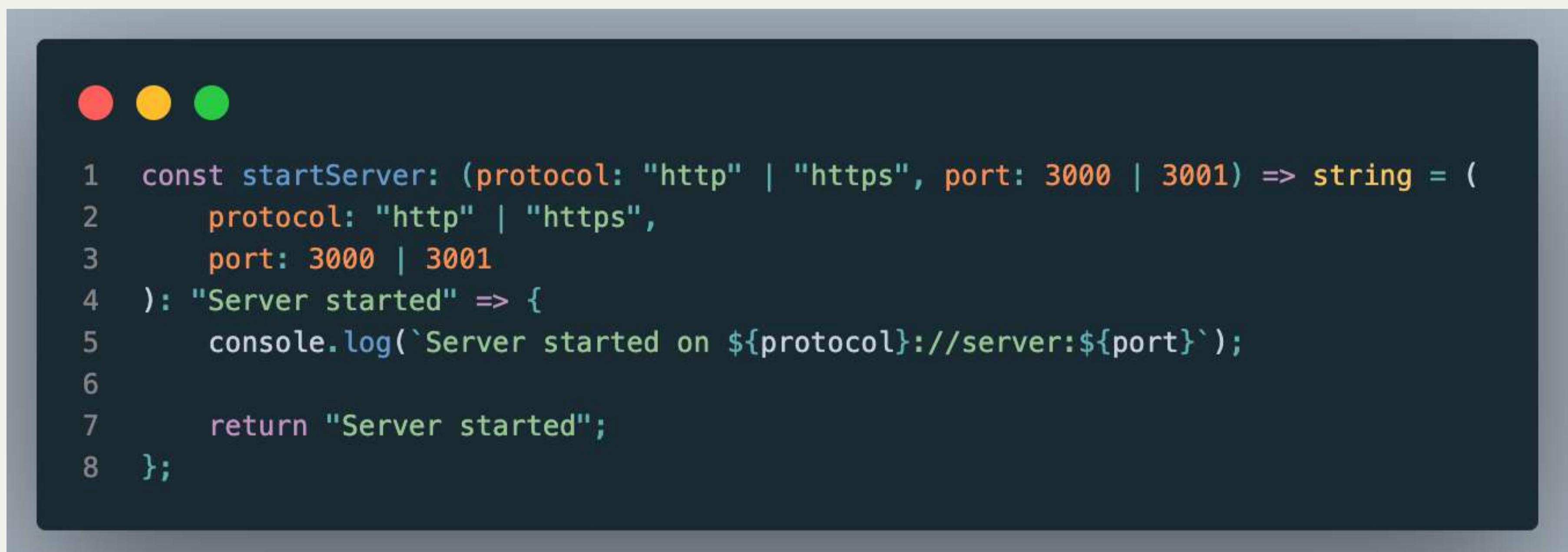
Для указания того, какой вид должен быть у объекта и из чего он может состоять, используются объектные литералы:

```
● ● ●  
1 const serverConfig: { protocol: "http" | "https"; port: 3000 | 3001 } = {  
2   protocol: "https",  
3   port: 3001,  
4 };
```

Ошибка уходит, но использовать такой синтаксис не очень приятно. Для удобства существуют специальные конструкции `type` и `interface` из следующих уроков

ОБЪЕКТНЫЕ ЛИТЕРАЛЫ ФУНКЦИЙ

Подобные литералы можно применять в качестве аннотаций и в функциях. (Функция - это объект в JS). Таким образом мы опишем какие аргументы принимает функция, в каком виде и что она может вернуть:



```
1 const startServer: (protocol: "http" | "https", port: 3000 | 3001) => string = (
2   protocol: "http" | "https",
3   port: 3000 | 3001
4 ): "Server started" => {
5   console.log(`Server started on ${protocol}://${port}`);
6
7   return "Server started";
8 }
```

Самое главное - это отличить аннотацию от объявления самой функции, что при таком синтаксисе не всегда легко. Заметьте, в аннотации данного примера мы возвращаем `string`, а у самой функции указано, что она возвращает определенную строку `"Server started"`

Такой код непросто читать, аннотацию невозможно переиспользовать, так что и применение у него нечастое. Но возможность всегда остается

СОЗДАНИЕ TYPE

Как и было указано, можно создавать type, который содержит описание объекта, а не только отдельные литералы. Это позволит нам указывать объектам, что они должны быть одной формы (одного формата, object shape):

```
● ● ●  
1  type Config = { protocol: "http" | "https"; port: 3000 | 3001 };  
2  
3  const serverConfig: Config = {  
4      protocol: "https",  
5      port: 3001,  
6  };
```

Теперь type с именем Config можно использовать для аннотирования других объектов. Если они не будут соответствовать этой форме - будет ошибка. В свойствах объекта внутри type может быть что угодно (литералы, типы, объекты и тп.)

В отдельный type можно выносить и описание функции:

```
● ● ●  
1  type StartFunction = (protocol: "http" | "https", port: 3000 | 3001) => string;  
2  
3  const startServer: StartFunction = (  
4      protocol: "http" | "https",  
5      port: 3000 | 3001  
6  ): "Server started" => {  
7      console.log(`Server started on ${protocol}://${port}`);  
8  
9      return "Server started";  
10  };
```

TYPE INTERSECTION

Нам часто приходится комбинировать типы для удобного и быстрого создания нужных нам. Иногда мы не хотим дублировать код (принцип DRY), иногда типы приходят нам из сторонней библиотеки или файла. В этих и других случаях нам понадобится оператор пересечения (&)

Добавим в пример новый тип с ролью, с помощью которого будет создан новый тип конфигурации:

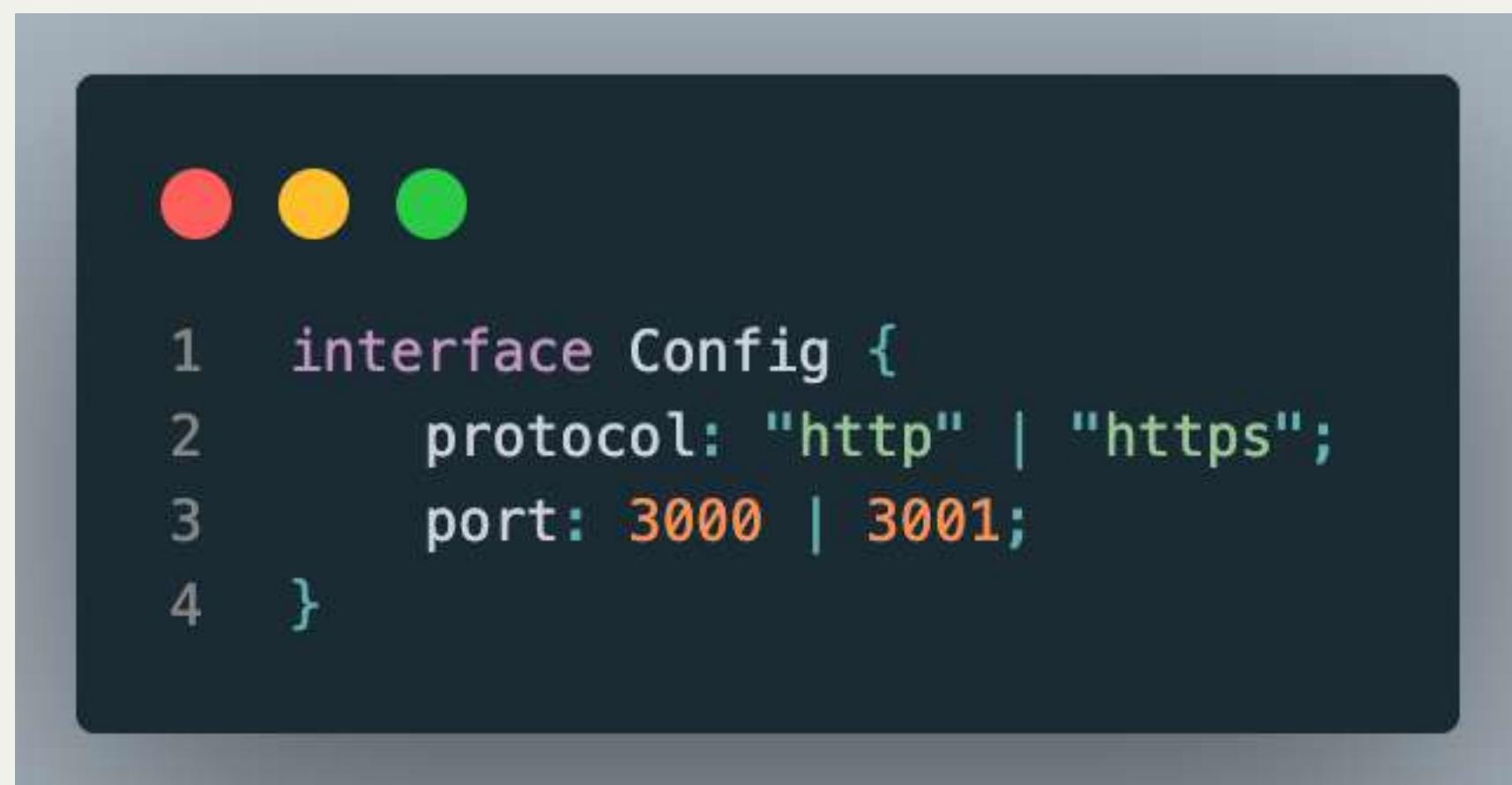
```
1 type Config = { protocol: "http" | "https"; port: 3000 | 3001 };
2
3 type Role = {
4     role: string;
5 };
6
7 type ConfigWithRole = Config & Role;
8
9 const serverConfig: ConfigWithRole = {
10     protocol: "https",
11     port: 3001,
12     role: "admin",
13 };
14
15 const backupConfig: Config = {
16     protocol: "http",
17     port: 3000
18 };
```

Благодаря оператору пересечения (intersection, &) мы скомбинировали два типа и получили тип ConfigWithRole. Он содержит все свойства из объединенных типов. Теперь все три туре можно использовать в коде

INTERFACES

Интерфейс - это еще один тип в TS, который позволяет синтаксически записать шаблон того **объекта**, который будет создан

Интерфейс создается при помощи ключевого слова **interface**, имени с большой буквы и раскрытия фигурных скобок:



```

1 interface Config {
2     protocol: "http" | "https";
3     port: 3000 | 3001;
4 }

```

В свойствах объекта может быть что угодно: литералы, другие типы и тд. При замене type на interface в коде ничего не сломается. Они действительно очень похожи. Отличия в логике использования будут рассмотрены в следующем уроке.

Обычно имена интерфейсов пишут в виде венгерской нотации, начиная с буквы I и дальше camelCase. Это необязательное правило, зависит от проекта и компании (IConfig, IMainSlider, IAdminUser ...)

Префикс	Сокращение от	Смысл	Пример
s	string	строка	sClientName
n, i	int	целочисленная переменная	nSize, iSize
b	boolean	булева переменная	bIsEmpty
a	array	массив	aDimensions
p	pointer	указатель	pBox
r	reference	ссылка	rBoxes
C	class	класс	CString
T	type	тип	TObject
I	interface	интерфейс	IDispatch
v	void	отсутствие типа	vReserved

ОБЪЕДИНЕНИЕ ИНТЕРФЕЙСОВ

Для комбинации нескольких интерфейсов и получения нового используется ключевое слово `extends`:

```
● ● ●  
1 interface Config {  
2     protocol: "http" | "https";  
3     port: 3000 | 3001;  
4 }  
5  
6 interface Role {  
7     role: string;  
8 }  
9  
10 interface ConfigWithRole extends Config, Role {  
11     // Тут могут быть новые свойства или пустые скобки  
12 }  
13  
14 const serverConfig: ConfigWithRole = {  
15     protocol: "https",  
16     port: 3001,  
17     role: "admin"  
18 };
```

Полученный интерфейс будет иметь все свойства указанных + можно добавлять новые внутри фигурных скобок

INDEX SIGNATURES

Если вы не знаете сколько свойств будет в объекте, но знаете, в каком виде они все будут, то можно использовать специальный синтаксис:

```
● ● ●  
1 interface Styles {  
2     [key: string]: string;  
3 }  
4  
5 const styles: Styles = {  
6     position: "absolute",  
7     top: "20px",  
8     left: "50px",  
9 };
```

В документации он называется Index Signatures, в реальной жизни слышал только "индексная сигнатурра"

Теперь все объекты, аннотированные этим интерфейсом, должны соблюдать структуру: свойство только строка, его значение - тоже только строка

РАЗЛИЧИЯ МЕЖДУ TYPE И INTERFACE

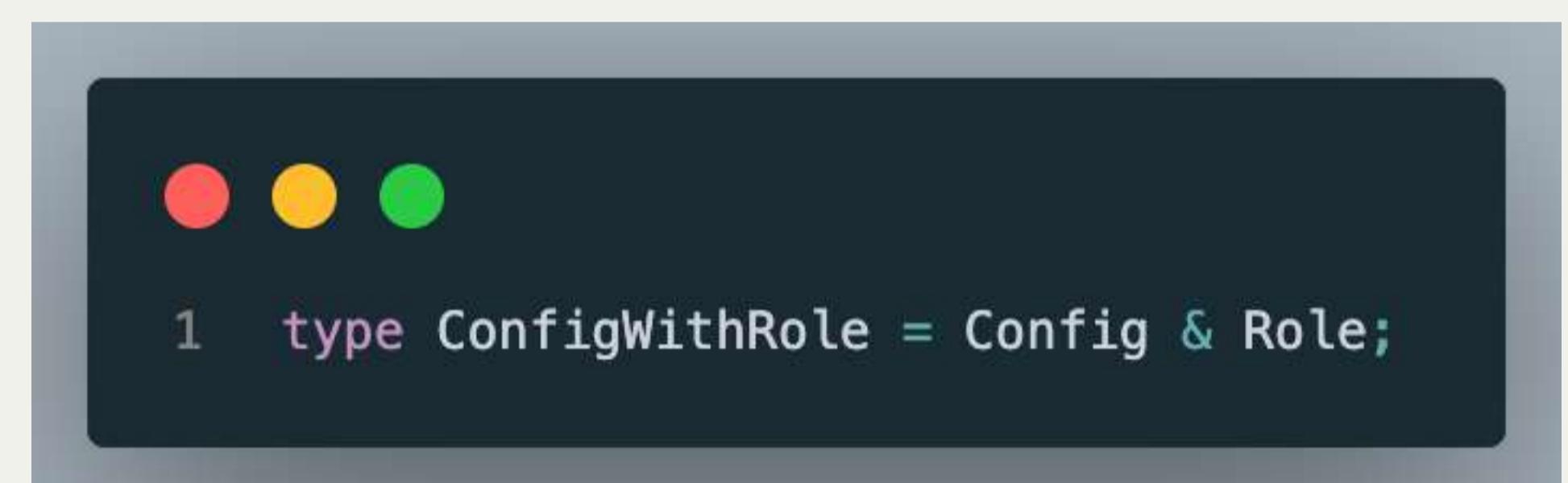
Две этих сущности очень похожи, но есть и отличия, которые определяют что именно использовать в разных ситуациях

- 👉 Разница в синтаксисе создания новых type и interface:



```
1 interface IConfigWithRole extends IConfig, IRole {}
```

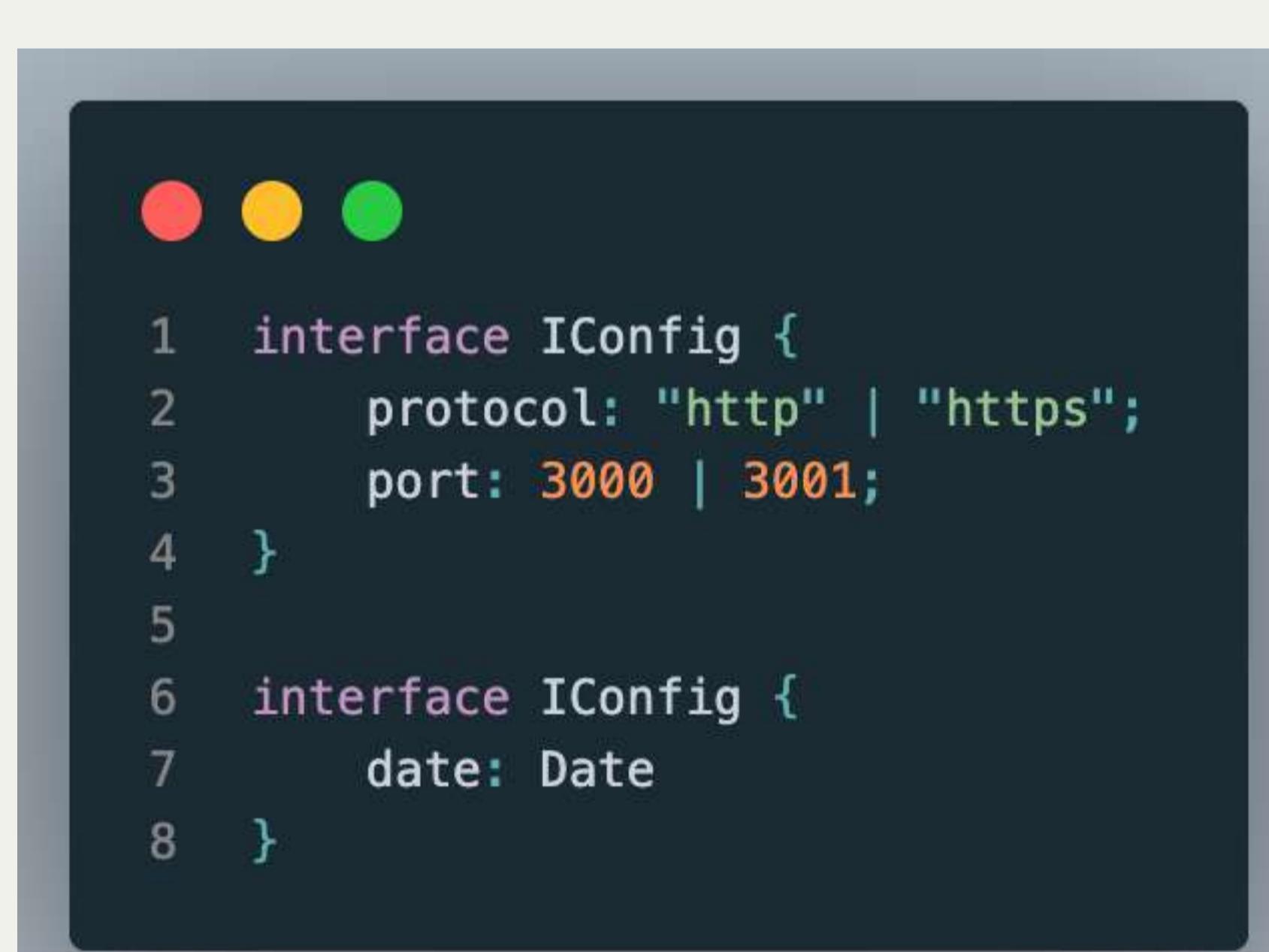
Оператор extends,
внутри скобок можно добавить новые свойства



```
1 type ConfigWithRole = Config & Role;
```

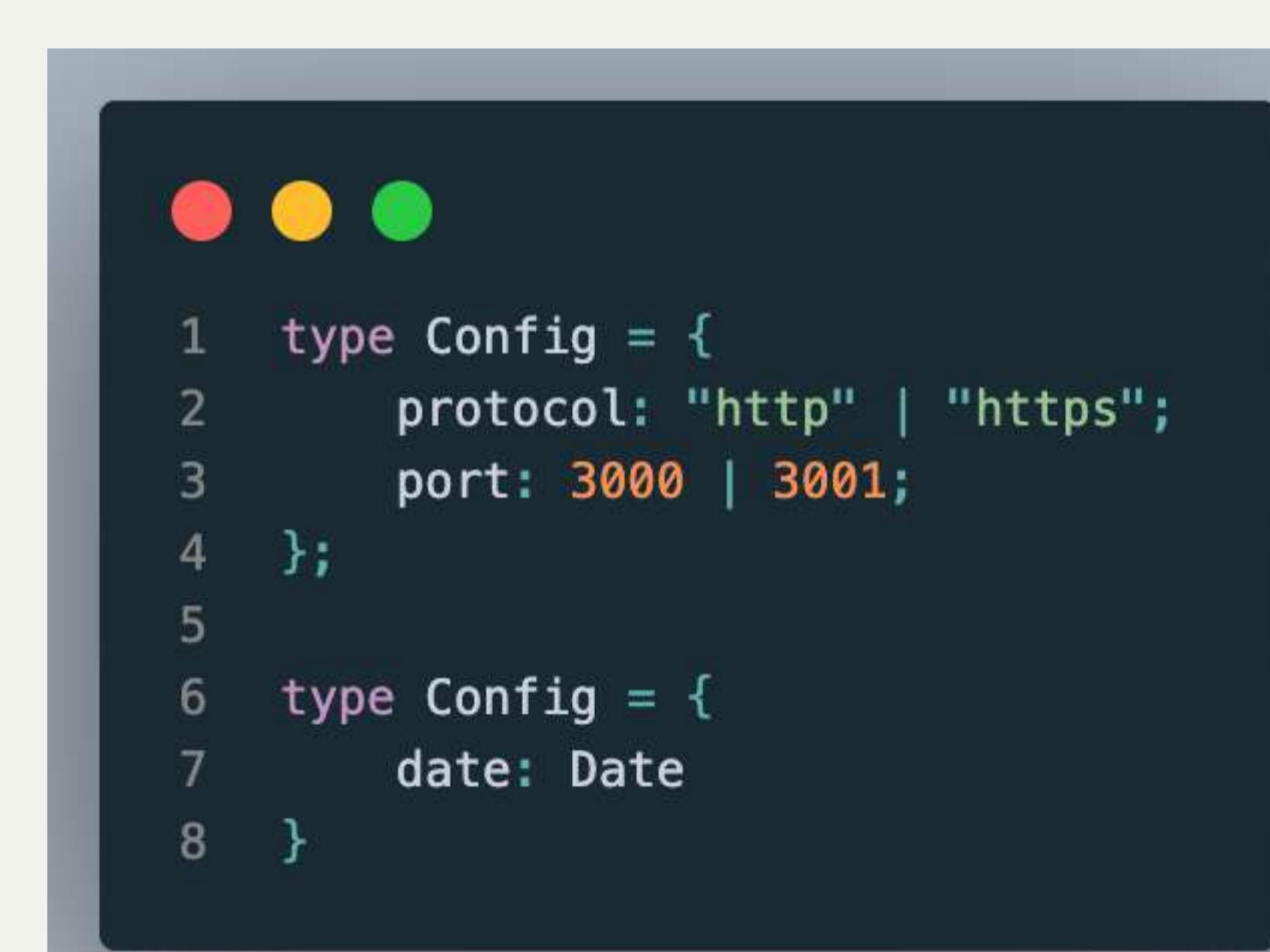
Оператор intersection

- 👉 Разница в синтаксисе расширения существующих type и interface:



```
1 interface IConfig {
2   protocol: "http" | "https";
3   port: 3000 | 3001;
4 }
5
6 interface IConfig {
7   date: Date
8 }
```

Допустимо добавлять новые поля
в существующий интерфейс



```
1 type Config = {
2   protocol: "http" | "https";
3   port: 3000 | 3001;
4 };
5
6 type Config = {
7   date: Date
8 }
```

Ошибка!
Дублирование имени дает ошибку

Учтите, что расширение интерфейсов таким образом - **плохая практика**. Создавайте их сразу полноценными, а если необходимо внести дополнения - делайте их сразу в строках, где он объявлен.

Иключение - это когда приходится расширять сторонний интерфейс, который вы импортировали себе в код и не имеете доступа к оригинальной сущности.

👉 В интерфейсы можно помещать **только объекты**, в type допустимы **литералы**:

```
● ● ●  
1 interface IProtocol "http" | "https";  
2  
3 interface IConfig {  
4     protocol: "http" | "https";  
5     port: 3000 | 3001;  
6 }
```

В интерфейс поместить литерал не выйдет

```
● ● ●  
1 type Protocol = "http" | "https";  
2  
3 type Config = {  
4     protocol: Protocol;  
5     port: 3000 | 3001;  
6 };
```

В type помещен строковый литерал

На основании этих отличий мы и выбираем сущность для работы:

- ! Если вам нужен примитивный тип в качестве псевдонима – то тут **только типы**
- ! Если вы откуда-то берете готовый интерфейс и его нужно расширить – **это интерфейсы**
- ! Если же вы просто работаете с объектами, то конкретной разницы не будет

Но чаще всего для работы с объектами используются интерфейсы! Это связано и с дополнением сторонних интерфейсов, и с работой с классами, и с тем, что так исторически сложилось в программировании

Советую использовать именно их. Документация так же вам об этом скажет:

If you would like a heuristic, use interface until you need to use features from type.

ВЫВОД ТИПОВ

При написании кода TS постоянно анализирует его и следит за тем, чтобы все операции были совместимы, не было синтаксических или логических ошибок.

Это происходит даже тогда, когда вы не указываете аннотации. Именно поэтому при наведении на переменную `a` в коде:

```
let a = 'string';
```

Вы увидите её тип. Этот механизм называется **вывод типов**. И раз он работает все время, то на него можно переложить часть рутинной работы разработчика. Предупреждения об ошибках и autocomplete будут продолжать работать

В различных источниках вы можете встретить такой совет:

Не нужно явно указывать типы, если за вас это может сделать вывод типов.

Но я добавлю два момента, которые нужно учитывать во время реальной работы:

- 👉 Соблюдение семантики кода. Код **должен быть понятен** для вас и других разработчиков, даже спустя время. Аннотации могут помочь в этом, не смотря на то, что TS понимает, с чем работает
- 👉 Разные стилистические правила в проектах и компаниях. Если сказано указывать типы - указываем их

ИСКЛЮЧЕНИЯ ИЗ ПРАВИЛ

Вывод типов срабатывает не всегда. Есть ситуации, когда он не понимает, что будет находиться там. Тут стоит использовать аннотации:

- 👉 Создание пустой переменной. Даже после помещения значения тип у переменной останется any:

```
1 let salary;
2 salary = 500;
```

Переменная так и останется типа any

```
1 let salary: number;
2 salary = 500;
```

Указание аннотации избавит от проблем

- 👉 Функции, которые возвращают any (JSON.parse() и тп.)
- 👉 Конфликт вывода типов с вашей логикой кода:

```
1 const isOkay = true;
2 let movement = false;
3
4 if (isOkay) {
5     movement = "moving";
6 }
```

Ошибка, строка не может быть помещена в boolean

```
1 const isOkay = true;
2 let movement: boolean | string = false;
3
4 if (isOkay) {
5     movement = "moving";
6 }
```

Указание аннотации с union типом избавляет от проблемы

Обратите внимание! Если создавать переменные с примитивными литералами через **const**, то **вывод типов** определит их как литерал. Если через **let** - то как общий тип (string, number, boolean...)

НЕОБЯЗАТЕЛЬНЫЕ СВОЙСТВА И АРГУМЕНТЫ

Для указания того, что свойство в объекте или аргумент в функции может быть необязательным, используется оператор optional - **“?”**
В объекте ставим его после названия свойства:

```
1 interface User {
2   login: string;
3   password: string;
4   age?: number;
5   addr?: string;
6 }
```

```
1 interface User {
2   login: string;
3   password: string;
4   age: number;
5   parents?: {
6     mother?: string;
7     father?: string;
8   };
9 }
```

Бывают ситуации, когда объекты должны соблюдать строгую форму, даже если значения у свойства нет. Свойство должно быть всегда. В таком случае нам поможет **union тип**, а не optional operator:

```
1 interface User {
2   login: string;
3   password: string;
4   age: number;
5   addr: string | undefined;
6 }
```

В функциях optional operator устанавливается после названия аргумента:

```
1 function sendUserData(obj: User, db?: string): void {
2   console.log(obj.parents?.father?.toLowerCase(), db?.toLowerCase());
3 }
```

Теперь при использовании такого аргумента внутри тела функции используется **optional chaining operator (ES2020)**

Он позволяет сделать запрос к свойству или методу объекта и если его нет, то просто вернуть undefined. Это позволяет избегать ошибок

ЕСЛИ МЫ ТОЧНО ЗНАЕМ, ЧТО ОНО СУЩЕСТВУЕТ

Для указания того, что сущность точно существует мы используем оператор **Non-Null and Non-Undefined** - **“!”**

Даже если TS будет предупреждать вас об ошибке, то этот оператор отключит это поведение. Использовать его стоит только тогда, когда вы **на все 100% уверены в наличии сущности**. Иначе будут ошибки в рантайме

```

1 interface User {
2     login: string;
3     addr: string | undefined;
4     parents?: {
5         mother?: string;
6         father?: string;
7     };
8 }
9
10 function sendUserData(obj: User, db?: string): void {
11     console.log(obj.parents!.father?.toLowerCase(), db!.toLowerCase());
12 }

```

Мы говорим коду, что свойство parents и аргумент db точно будут.

А значит к ним можно применить дальнейшие операции.

Иногда этот оператор позволяет подкорректировать логику TS и сказать, что функция точно будет синхронной. И другие ситуации, когда TS не уверен в существовании чего-либо на момент использования:

```

1 let dbName: string;
2 sendUserData(user, "ervervefvf");
3
4 console.log(dbName!);
5
6 function sendUserData(obj: User, db?: string): void {
7     dbName = "12345";
8     console.log(obj.parents!.father?.toLowerCase(), db!.toLowerCase());
9 }

```

В 4й строке TS не знает о том, что значение переменной уже назначено, поэтому показывает ошибку. Мы можем это исправить, так как знаем, что эта функция работает синхронно

В литературе прием выше называется **Definite Assignment Assertion** или **Утверждение определенного назначения**

НЕИЗМЕНЯЕМЫЕ СУЩНОСТИ

TS позволяет на уровне синтаксиса сказать, что **свойства объекта, массивы или кортежи являются неизменяемыми**. Любая операция, направленная на это, будет воспринята как ошибка. Для этого используется модификатор **readonly**

```
● ● ●  
1 interface User {  
2     readonly login: string;  
3 }  
4  
5 const user: User = {  
6     login: "first!"  
7 };  
8  
9 user.login = 'Error!' // Error
```

```
● ● ●  
1 const basicPorts: readonly number[] = [3000, 3001, 5555];  
2 basicPorts[0] = 5; // Error
```

При использовании **readonly** на массиве, он и его содержимое становятся неизменяемыми. Такие методы как **pop()**, **push()** и тп работать не будут. В кортежах тоже самое:

```
● ● ●  
1 const basicPorts: readonly [number, ...string[]} = [3000, '3001', '5555'];  
2 basicPorts[0] = 5; // Error  
3 basicPorts.push(566); // Error
```

Нужно помнить! Такие ограничения существуют только на этапе разработки, внутри TS. После компиляции эти свойства и массивы будут редактируемые. Но вот на этапе разработки они позволяют вам избежать ошибок, особенно, когда вы работаете в команде

АЛЬТЕРНАТИВНЫЙ СИНТАКСИС

Существуют альтернативные варианты записи при помощи дженериков. Эта тема будущих уроков, но добавим её и сюда. Такой синтаксис встречается реже, так как оператор удобнее. Но в некоторых ситуациях они так же применимы:

```
● ● ●  
1 const userFreeze: Readonly<User> = {  
2   login: "first!",  
3   password: "qwerty",  
4   age: 50,  
5 };  
6  
7 userFreeze.age = 4484;  
8 userFreeze.password = "dfffd";
```

Все свойства в объекте стали неизменяемыми за счет применения Readonly на интерфейсе User

Для массивов другая конструкция:

```
● ● ●  
1 const basicPorts: ReadonlyArray<number> = [3000, 3001, 5555];  
2 basicPorts[0] = 5; // Error  
3 basicPorts.push(566); // Error
```

ПЕРЕЧИСЛЕНИЯ В TYPESCRIPT

В разработке бывают ситуации, когда что-то ограничено перечислением нескольких вариантов. И программа выбирает один из них. Например:

- Варианты движения объекта: *вверх, вниз, влево, вправо*
- Список доступных валют: *€, \$, ₽*
- Список доступных анимаций: *fadeIn, fadeDown, swipeLeft, swipeRight*

И мы хотим предоставить ей и разработчикам только ограниченный выбор. Для этого используются **перечисления (Enum)**. Эта структура существует **только внутри TS**. Она позволяет избегать опечаток и применения сторонних вариантов:

```
● ○ ●
1 enum Directions {
2     TOP,
3     RIGHT,
4     LEFT,
5     BOTTOM,
6 }
```

Стандартный enum (числовой)
Автоматическая нумерация значений

```
● ○ ●
1 enum TimingFunc {
2     EASE = "ease",
3     EASE_IN = "ease-in",
4     LINEAR = "linear",
5 }
```

Строковый enum

```
● ○ ●
1 enum TimingFunc {
2     EASE = 1,
3     EASE_IN = 2,
4     LINEAR = 10
5 }
```

Числовой enum
с установленными значениями

```
● ○ ●
1 function frame(elem: string, dir: Directions, tFunc: TimingFunc): void {
2     if (dir === Directions.RIGHT) {
3         console.log(tFunc);
4     }
5 }
6
7 frame("id", Directions.RIGHT, TimingFunc.LINEAR);
```

Применение перечисления в функции
Enum ограничивает варианты и не дает опечататься

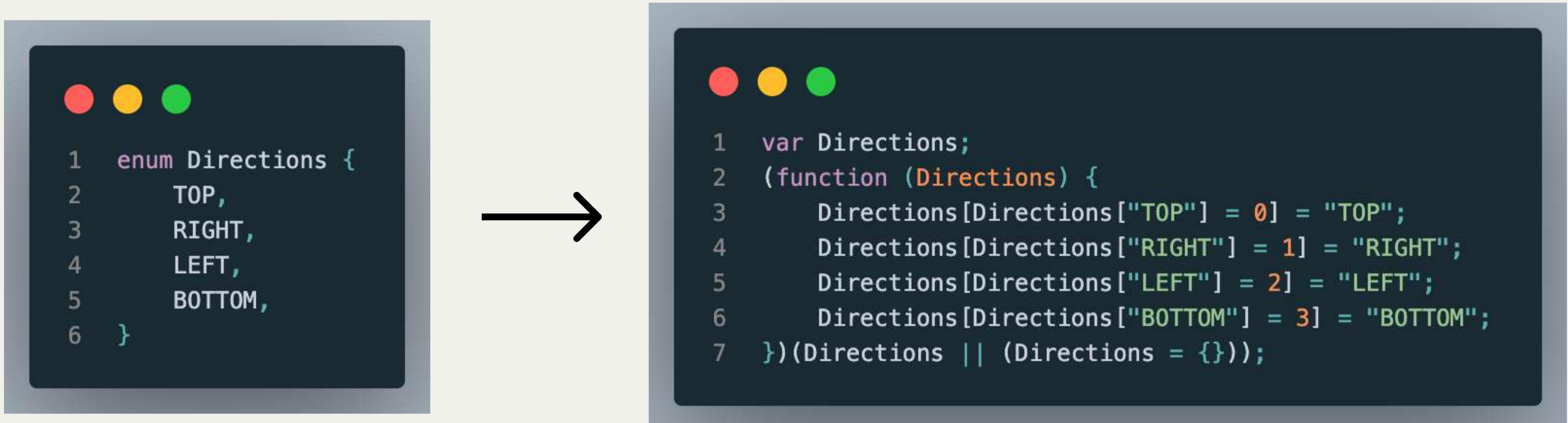
Существует вариант гетерогенного enum'а - это **комбинация строковых и числовых значений**. Но в реальности использовать такое **не стоит**. Лучше задуматься об рефакторинге кода в таком случае.

В строковых перечислениях нельзя заниматься вычислениями для получения новых значений. В числовых - можно:

```
● ○ ●
1 enum TimingFunc {
2     EASE = 1,
3     EASE_IN = 2,
4     LINEAR = EASE * 2,
5 }
```

ПЕРЕЧИСЛЕНИЯ ПОСЛЕ КОМПИЛЯЦИИ

После преобразования .ts файлов в .js перечисления превращаются в **функции с вычислением и записью значений**:



Есть вариант немного ускорить их работу в конечном коде, если вычисления внутри enum занимают какое-то время. Для этого используются константные перечисления:



```
1 const enum TimingFunc {  
2     EASE = "ease",  
3     EASE_IN = "ease-in",  
4     LINEAR = "linear",  
5 }
```

После компиляции они не формируют функции. Компилятор сделал все вычисления, нашел все ссылки на это перечисление и заменил их значениями в конечном коде. Остаются лишь комментарии где это произошло

Этот прием имеет свои подводные камни и может приводить к багам. Так что почти всегда вы будете встречать стандартный вариант. Лучше пожертвовать несколькими миллисекундами, но не получать ошибок

ТИП UNKNOWN

Бывают ситуации, когда мы можем получить сущность неизвестного типа. Для работы с ними существует **тип unknown**, который является **типовозащищенным аналогом any**:

```
● ● ●
1 let smth: any;
2
3 smth = 'str';
4
5 let data: string[] = smth;
6 data.find(e => e);
```

При использовании **any** мы не получим ошибку в 5й строке. Но в рантайме она будет

```
● ● ●
1 let smth: unknown;
2
3 smth = 'str';
4
5 let data: string[] = smth;
6 data.find(e => e);
```

При использовании **unknown** ошибка будет. Нельзя применять метод неизвестно к чему

Опасность **any** в том, что это **любой тип**. В нем нет никакой строгости. Никакие проверки типов в нем не выполняются. А вот **unknow** – **это неизвестный тип**.

В **any** может быть **что угодно**, а в **unknown** – **мы не знаем что может быть**. К чему угодно может применяться **что угодно**, а к неизвестному – **ничего**

```
● ● ●
1 const someValue: any = 10;
2 someValue.method(); // Ok
```

```
● ● ●
1 const someValue: unknown = 10;
2 someValue.method(); // Error
```

Для работы с этим типом необходимо использовать **сужение типов**. Так мы поймем что это и сможем правильно с ним работать:

```
● ● ●
1 function fetchData(data: unknown): void {
2     if (typeof data === "string") {
3         console.log(data.toLowerCase());
4     }
5     data // тут unknown
6 }
```

ПРИМЕНЕНИЕ UNKNOWN

Данный тип можно использовать для работы с функциями, которые возвращают что-угодно. Например, `JSON.parse()`. Так мы избежим ошибок и правильно будем работать с данными:

```

 1 const userData =
 2   '{"isBirthdayData": true, "ageData": 40, "userNameData": "John"}';
 3
 4 function safeParse(s: string): unknown {
 5   return JSON.parse(s);
 6 }
 7
 8 const data = safeParse(userData);
 9
10 function transferData(d: unknown): void {
11   if (typeof d === "string") {
12     console.log(d.toLowerCase());
13   } else if (typeof d === "object" && d) {
14     console.log(data);
15   } else {
16     console.error("Some error");
17   }
18 }
19
20 transferData(data);

```

В JSON строке могут быть разные данные и благодаря функции `safeParse` мы получим не что угодно, а неизвестно что. А функция `transferData` использует сужение типов для правильной работы

В TS, в конструкции **try/catch** ошибка, приходящая в `catch` будет типа **unknown**. Это происходит из-за того, что компилятор не знает, какую именно ошибку разработчик “выкинет” из блока `try`. Это может быть строка, экземпляр определенного класса или что-то еще. Так что здесь тоже стоит применять сужение типов:

```

 1 try {
 2   if (1) {
 3     throw new Error("error");
 4   }
 5 } catch (e) {
 6   if (e instanceof Error) {
 7     console.log(e.message);
 8   } else if (typeof e === "string") {
 9     console.log(e);
10   }
11 }

```

Работа с `union` и `intersection` типами:

- Если тип `unknown` составляет тип **объединение** (`union`), то он перекроет **все типы**, за исключением типа `any`
- Если тип `unknown` составляет тип **пересечение** (`intersection`), то он **будет перекрыт всеми типами**

```

 1 type T0 = any | unknown; // any
 2 type T1 = number | unknown; // unknown
 3 type T2 = any & unknown; // any
 4 type T3 = number & unknown; // number

```

ЗАПРОСЫ ТИПОВ

Механизм, который позволяет получить тип определенной сущности называется **запрос типа** (type query). Чаще всего он необходим, когда мы четко **понимаем**, какой тип нам нужен в этой ситуации и он нигде не будет дальше **повторяться**

Реализуется через оператор `typeof` + сущность:

```
● ● ●  
1 const dataFromControl = {  
2     water: 200,  
3     el: 350,  
4 };  
5  
6 function checkReadings(data: typeof dataFromControl): boolean {  
7     const dataFromUser = {  
8         water: 200,  
9         el: 350,  
10    };  
11}
```

Теперь аргумент имеет нужный тип, который совпадает с типом объекта `dataFromControl`. В теле функции появятся нужные подсказки и проверки:

```
● ● ●  
1 if (data.el === dataFromUser.el && data.water === dataFromUser.water) {  
2     return true;  
3 } else return false;  
4 }
```

Как пример, можно использовать, когда в коде уже есть сущность, от которой мы отталкиваемся вместо дублирования типа:

```
● ● ●  
1 const PI = 3.14;  
2 let PIClone: typeof PI;
```

УТВЕРЖДЕНИЕ ТИПОВ

В реальном коде случаются ситуации, когда полученное откуда-то значение не соответствует тому типу, который мы ожидаем. Для того, чтобы попросить TS **пересмотреть** свое отношение к определенному типу, используется механизм **утверждения типа**



```

1 const fetchData = (url: string, method: "GET" | "POST"): void => {
2   console.log(method);
3 }
4
5 const requestOptions = {
6   url: "https://someurl.com",
7   method: "GET",
8 };
9
10 fetchData("qqq", "GET");
11 fetchData(reqOptions.url, reqOptions.method); // Error

```

Свойство `reqOptions.method` имеет тип `string`, а значит не подходит под аннотацию аргумента с литералами. Если мы **точно** знаем, что значение свойства подходит, то мы можем утвердить это значение оператором **as**:



```

1 const fetchData = (url: string, method: "GET" | "POST"): void => {
2   console.log(method);
3 }
4
5 const requestOptions = {
6   url: "https://someurl.com",
7   method: "GET",
8 };
9
10 fetchData("qqq", "GET");
11 fetchData(reqOptions.url, reqOptions.method as "GET");

```

Что **важно** помнить:

- 👉 В таких операциях **вы берете риск на себя**. TS позволяет вам уточнить строку в её литерал (конкретное значение). **Но при изменении значения свойства в объекте** TS вам не укажет на ошибку
- 👉 TS **не позволит** вам создавать нелогичные конструкции. Вы просите пересмотреть отношение, но **не указываете воспринимать** так-то. Доступно лишь утверждение **более специализированных типов**: строка - её литерал, число - его литерал и тп. В обратную сторону операция не имеет смысла



```

1 fetchData(reqOptions.url, reqOptions.method as 5); // Error

```

- 👉 Необходимо отличать **утверждение типов** от **преобразования типов**. От конструкции с `as` после компиляции **ничего не останется**. Это просто просьба того, каким типом считать эту сущность при разработке. Преобразование - это превращение числа в строку и тп.

АЛЬТЕРНАТИВНЫЕ ВАРИАНТЫ

Эту ситуацию можно решить еще двумя другими способами, так же утверждением типов. **Первый** - утвердить значение еще на этапе объекта:



```

1 const reqOptions = {
2   url: "https://someurl.com",
3   method: "GET" as "GET",
4 };

```

Это удобно, так как и утверждение и значение находятся в одном месте. **Но**, если свойство используется в разных места по разному, это может навредить. Иногда приходится утверждать тип в момент использования. **Второй** - превратить **весь объект в литерал типа**:



```

1 const reqOptions = {
2   url: "https://someurl.com",
3   method: "GET",
4 } as const;

```

Напоминание! Если переменная с примитивом создается без аннотации и через `const`, то вывод типов присваивает ей **литеральный тип**. С объектами так не работает, но можно сделать через `as const`

Существует **альтернативный синтаксис** утверждения типов, через угловые скобки. Он **не так удобен и не работает** в некоторых технологиях, где идет конфликт по скобкам (например React):



```

1 fetchData(reqOptions.url, <"GET">reqOptions.method);

```

Часто утверждение типов можно встретить при работе с DOM-деревом, когда мы хотим уточнить, с каким элементом мы работаем:



```

1 const box = document.querySelector(".box") as HTMLElement;
2 const input = document.querySelector("input") as HTMLInputElement;
3 // Альтернативный:
4 const input = <HTMLInputElement>document.querySelector("input");

```

Мы “уточняем” более специализированный интерфейс, ведь **HTMLElement** это **частный случай Element** (будет рассмотрено в следующих уроках)

Важно помнить, что элемент может быть не найден на странице, так что использование различных конструкций по предотвращению ошибок обязательно (`try/catch`, `if` и тп.)

СОЗДАНИЕ ЛИТЕРАЛОВ ЧЕРЕЗ КОНСТРУКТОР

В JS есть возможность создавать сущности через **конструкторы**. Это **не самый лучший подход**, так как создание экземпляра занимает больше времени, чем простое объявление литерала. Код усложняется. И можно получать довольно странные ошибки:

```
> const x = 500;
  const y = new Number(500);

  console.log(x == y);
  console.log(x === y)
  true
  false
```

В TS тоже есть такая возможность. При этом тип переменной будет указан с большой буквы.

Так что в TypeScript существует типы, **представляющие конструкторы одноименных типов из JavaScript** (`Number`, `String`, `Boolean`, `Symbol`, `BigInt`), и также существуют типы, представляющие примитивные значения **литералов** (`number`, `string`, `boolean`, `symbol`, `bigint`)

```
1 const num = 5; // type: 5
2 const num: number = 5; // type: number
3 const num = new Number(5); // type: Number
4 const num = Number(5); // type: number
```

В работе это тоже может дать свои особенности, так как тип `number` неявно преобразуется в тип `Number`, но **не наоборот**:

```
1 let n: number = 5;
2 let N: Number = new Number(5);
3
4 N = n; // Ok
5 n = N; // Error
```

Отсюда сделаем **небольшие выводы**:

- 👉 Не используйте конструкторы для создания значений. Это может привести к непонятным ошибкам
- 👉 На собеседованиях бывают вопросы с подвохом, где спрашивают сколько примитивных типов есть в ТС. Сюда можно и относить эти типы с большой буквы, которые являются аналогами конструкторов в JS
- 👉 В продвинутой литературе вы можете встретить эти типы. Будьте внимательны и обращайте внимание на регистр первой буквы

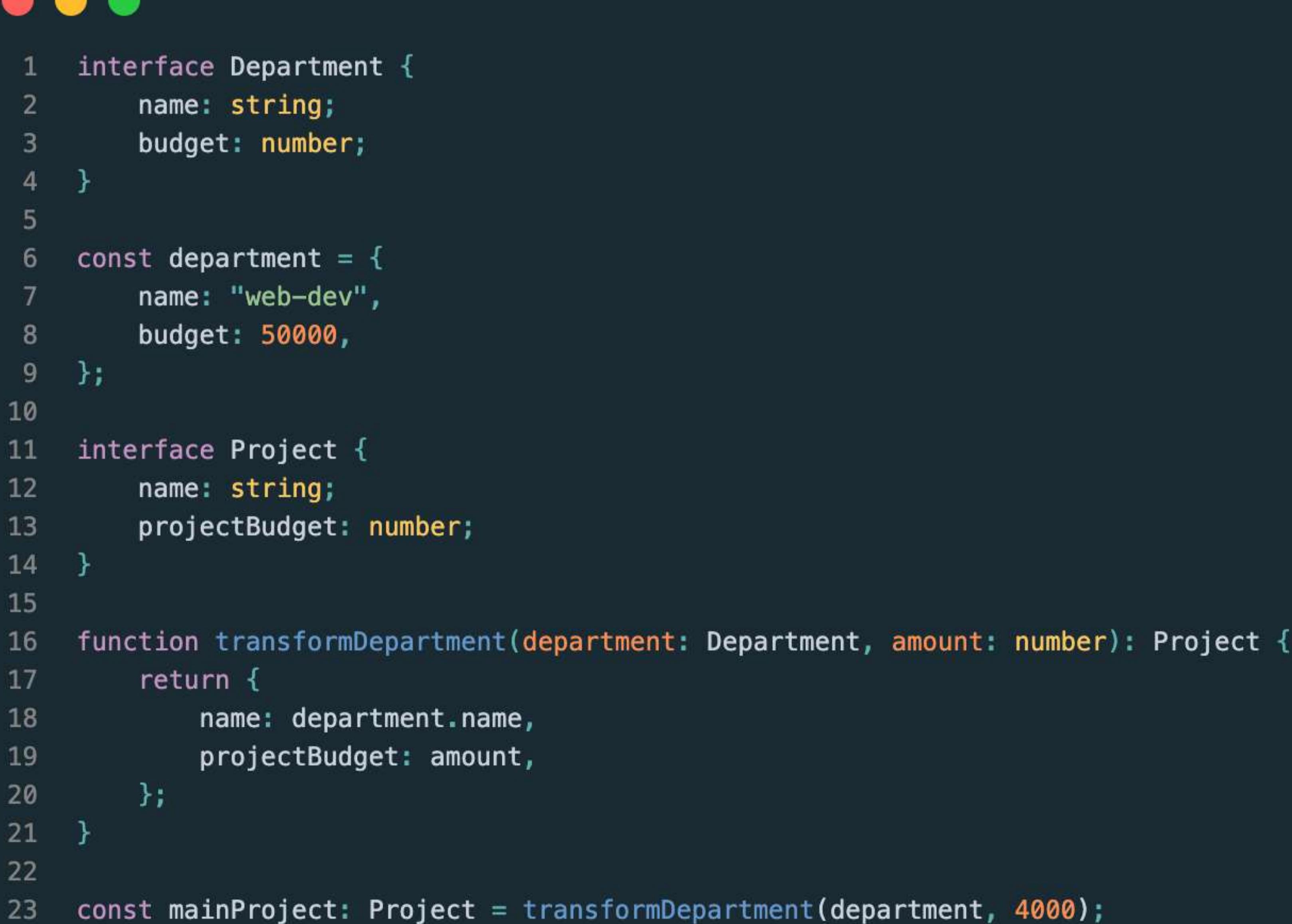
ПРЕОБРАЗОВАНИЕ ТИПОВ

Самые банальные преобразования работают точно так же, как и в нативном JS. Они валидны за счет того, что TS четко понимает последовательность действий.



```
1 const num = 5;
2 const strNum: string = num.toString();
3
4 const str = "5";
5 const numStr: number = +str;
6 // и другие методы
```

С объектами все немного сложнее. Если мы говорим про трансформацию одного объекта в другой, то чаще всего мы используем трансформирующие функции. Они позволяют гибко настраивать преобразование и легко его изменять в будущем:



```
1 interface Department {
2     name: string;
3     budget: number;
4 }
5
6 const department = {
7     name: "web-dev",
8     budget: 50000,
9 };
10
11 interface Project {
12     name: string;
13     projectBudget: number;
14 }
15
16 function transformDepartment(department: Department, amount: number): Project {
17     return {
18         name: department.name,
19         projectBudget: amount,
20     };
21 }
22
23 const mainProject: Project = transformDepartment(department, 4000);
```

А интерфейсы позволяют нам контролировать содержание этих объектов, типизировать их содержимое

ЗАЩИТНИК ТИПА (TYPE GUARD)

Когда мы изучали тему **сужения типов**, то мы создавали различные условия для их определения. Правила, которые позволяют выводу типов определить **суженный диапазон типов** для значения называются **защитниками типа, type guards**

```

● ● ●

1 function printMsg(msg: string[] | number | boolean): void {
2   if (Array.isArray(msg)) { // <= type guard
3     msg.forEach((m) => console.log(m));
4   } else if (typeof msg === "number") { // <= type guard
5     console.log(msg);
6   } else {
7     console.log(msg);
8   }
9 }
```

Для соблюдения принципа DRY (не повторяем код) мы такие правила можем вынести в отдельную функцию. В TS можно использовать дополнительный синтаксис для таких функций и создавать **пользовательские защитники типа**, которые возвращают предикат (лат. *praedicatum*, логическое значение: *true* или *false*)

```

● ● ●

1 function isNumber(n: unknown): n is number {
2   return typeof n === "number";
3 }
```

```

● ● ●

1 function isNumber(n: string[] | number | boolean): n is number {
2   return typeof n === "number";
3 }
```

Функция `inNumber` вернет `true` только если аргумент будет числом. Оператор `is` позволяет сказать, что будет возвращено **логическое значение**, где проверяется, что `n` это число. Дальше используем её в условии:

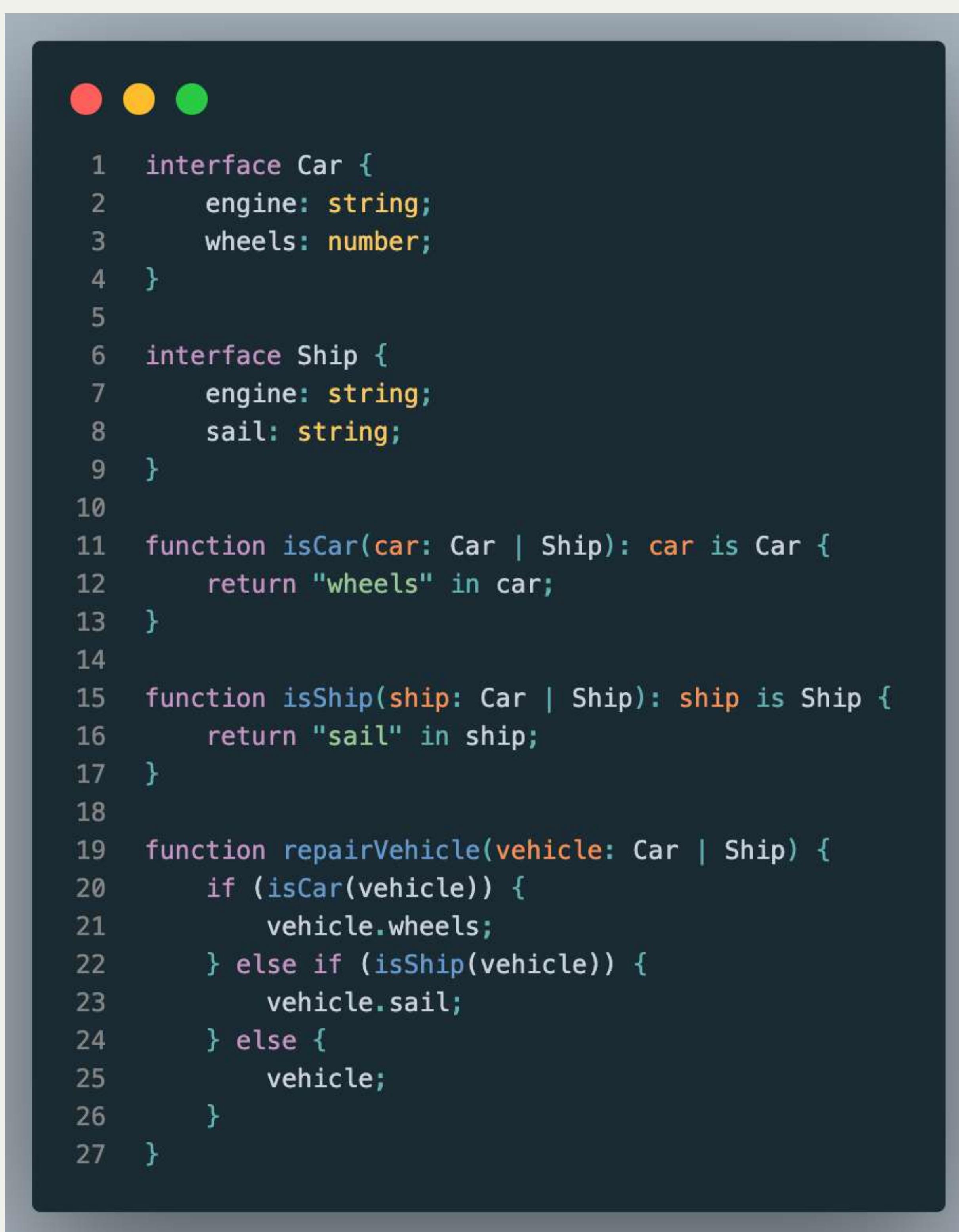
```

● ● ●

1 if (Array.isArray(msg)) {
2   msg.forEach((m) => console.log(m));
3 } else if (isNumber(msg)) {
4   console.log(msg);
5 } else {
```

ЗАЩИТНИК ТИПА (TYPE GUARD)

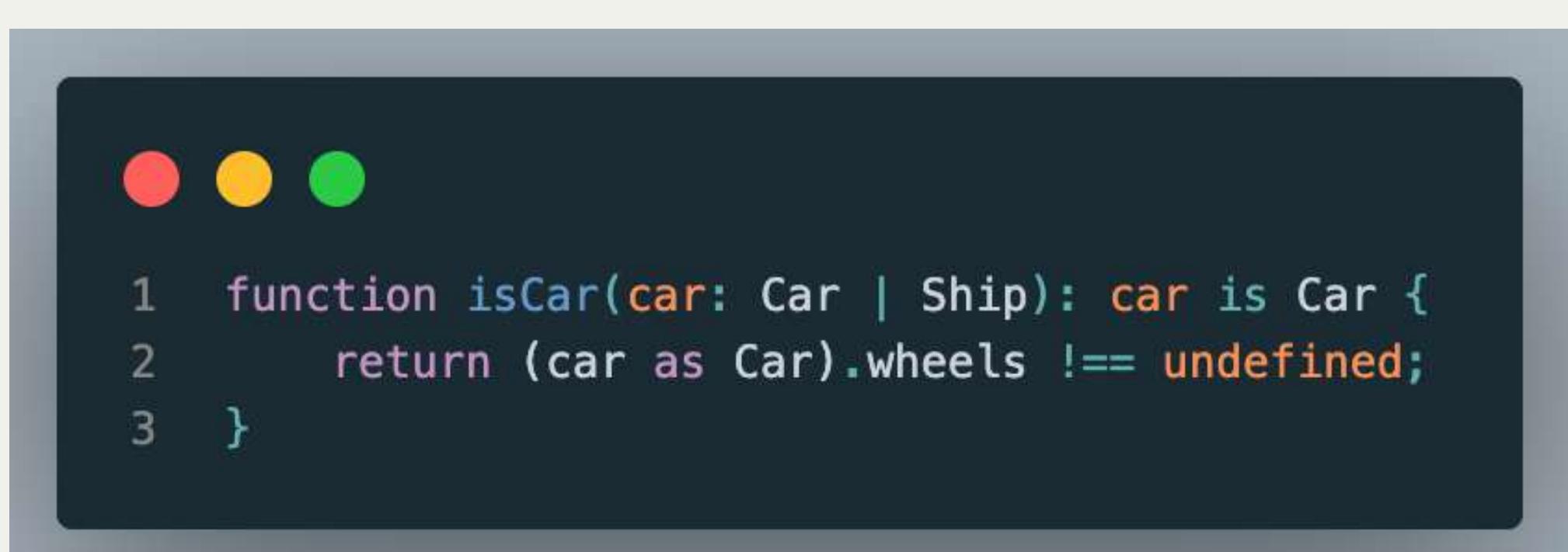
Более продвинутые защитники определяют, к какому объекту относится эта сущность. Например, по интерфейсу:



```
● ● ●

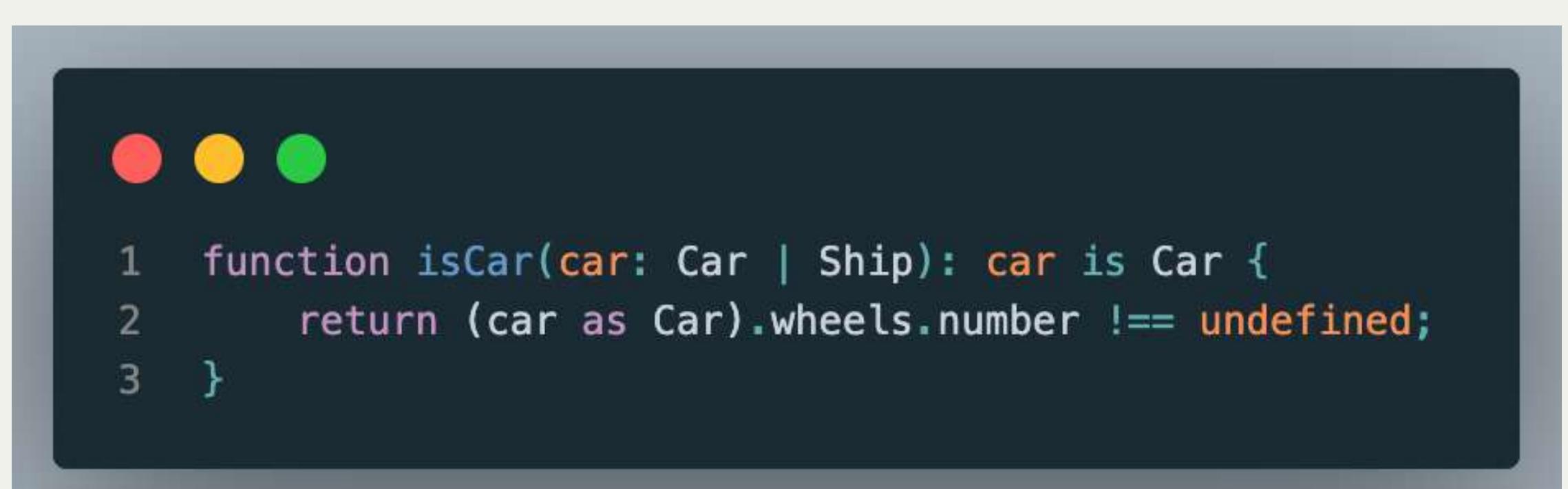
1 interface Car {
2   engine: string;
3   wheels: number;
4 }
5
6 interface Ship {
7   engine: string;
8   sail: string;
9 }
10
11 function isCar(car: Car | Ship): car is Car {
12   return "wheels" in car;
13 }
14
15 function isShip(ship: Car | Ship): ship is Ship {
16   return "sail" in ship;
17 }
18
19 function repairVehicle(vehicle: Car | Ship) {
20   if (isCar(vehicle)) {
21     vehicle.wheels;
22   } else if (isShip(vehicle)) {
23     vehicle.sail;
24   } else {
25     vehicle;
26   }
27 }
```

Тут мы узнаем, есть ли такое свойство в интерфейсе и возвращаем true, если оно есть. Если нет - false. За счет такой проверки, внутри функции мы получаем корректные подсказки о том, что это за объект. Но вы можете создавать **любые условия** внутри пользовательских защитников для ваших целей. Более продвинутый вариант функции выше, позволяет работать со вложенными структурами:



```
● ● ●

1 function isCar(car: Car | Ship): car is Car {
2   return (car as Car).wheels !== undefined;
3 }
```



```
● ● ●

1 function isCar(car: Car | Ship): car is Car {
2   return (car as Car).wheels.number !== undefined;
3 }
```

ПЕРЕГРУЗКА ФУНКЦИЙ

Для того, чтобы наглядно прописать все варианты использования функции и задокументировать их, применяется **перегрузка**. Мы описываем их и что будет возвращено из каждого:

```
● ● ●  
1 function calculateArea(side: number): Square;  
2 function calculateArea(a: number, b: number): Rect;  
3 function calculateArea(a: number, b?: number): Square | Rect { ... }
```

Необходимо соблюдать несколько правил:

- 👉 Перегрузка записывается до основного тела функции
- 👉 Аргументы могут называться другими именами, это допустимо
- 👉 Все перегрузки должны быть совместимы с главной функцией. То есть, нельзя сделать вот так:

```
● ● ●  
1 function calculateArea(side: string): Square; // Error  
2 function calculateArea(a: number, b: number): Rect;  
3 function calculateArea(a: number, b?: number): Square | Rect { ... }
```

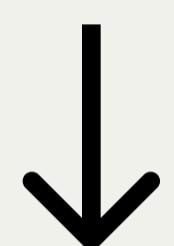
При использовании этого приема подсказки будут более **информативны и специфичнее**. Особенno стоит использовать данный прием, если функция сложная и имеет много **необязательных** аргументов

Перегрузка **никак не влияет** на работу или скорость выполнения функции. После компиляции полностью исчезает

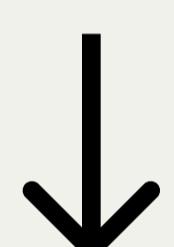
РАБОТА С DOM В TS

Особенностью работы с DOM-структурой в TS является то, что он предоставляет множество специальных интерфейсов для этого. Существует интерфейс **документа (Document)**, **различных событий (Event, MouseEvent и тп)** и **элементов на странице**. Последние подчиняются строгой иерархии в зависимости от специфиности:

Node



Element



HTMLElement



HTMLInputElement
HTMLParagraphElement
HTMLAnchorElement
и другие

Любой узел на странице: текстовый, содержимое изображения, любые элементы... Содержит только самые базовые свойства и методы

Любые элементы на странице, в том числе и невидимые на странице. Содержит базовые свойства и методы, присущие всем элементам. Получаем его без утверждения типа

Любые html-элементы на странице. Содержит более специфичные свойства и методы, присущие элементам

Определенные html-элементы на странице. Содержат специфичные для них свойства и методы в дополнение к общим

Все эти интерфейсы нужны для четкого указания с чем мы работаем и правильного доступа к нужным свойствам/методам

```

1 const box = document.querySelector(".box") as HTMLElement;
2 const p = document.querySelector(".paragraph") as HTMLParagraphElement;
3 const input = document.querySelector("a"); // Автоматическое определение
  
```

Если мы указываем селектор, по которому **и так понятно, что за элемент будет получен** - TS автоматически подставит нужный тип интерфейса. Утверждение в таком случае **не нужно**. Такая же ситуация и при создании новых элементов через команду `createElement()`

Помните, утверждение типа избавляет вас от `null` только в TS. Всегда есть шанс не получить элемент на странице. Так что не забывайте про стандартные средства защиты (`try/catch`, `if` и тд.)

ОБОБЩЕНИЯ

В программировании постоянно создаются конструкции, которые можно **переиспользовать** в разных ситуациях с разными данными. В стандартном JS мы просто передавали в функцию аргумент и в теле уже с ним работали. В TS необходимо **указать тип аргумента**

```
● ● ●
1 function processingData(data) {
2   ...
3
4   return data;
5 }
```

Функция, принимающие данные в виде строки, числа, логического значения или массива строк

Плохой вариант её типизации



```
● ● ●
1 function processingData(data: string | number | boolean | string[]): string | number | boolean | string[] {}
```

Для таких ситуаций в TS существует **механизм обобщений (generics, дженерики)** Он позволяет поставить “заглушку”, которая будет заменена при использовании:

```
● ● ●
1 function processingData<T>(data: T): T {
2   ...
3
4   return data;
5 }
6
7 let res1 = processingData(1); // type number
8 let res2 = processingData("1"); // type string
```

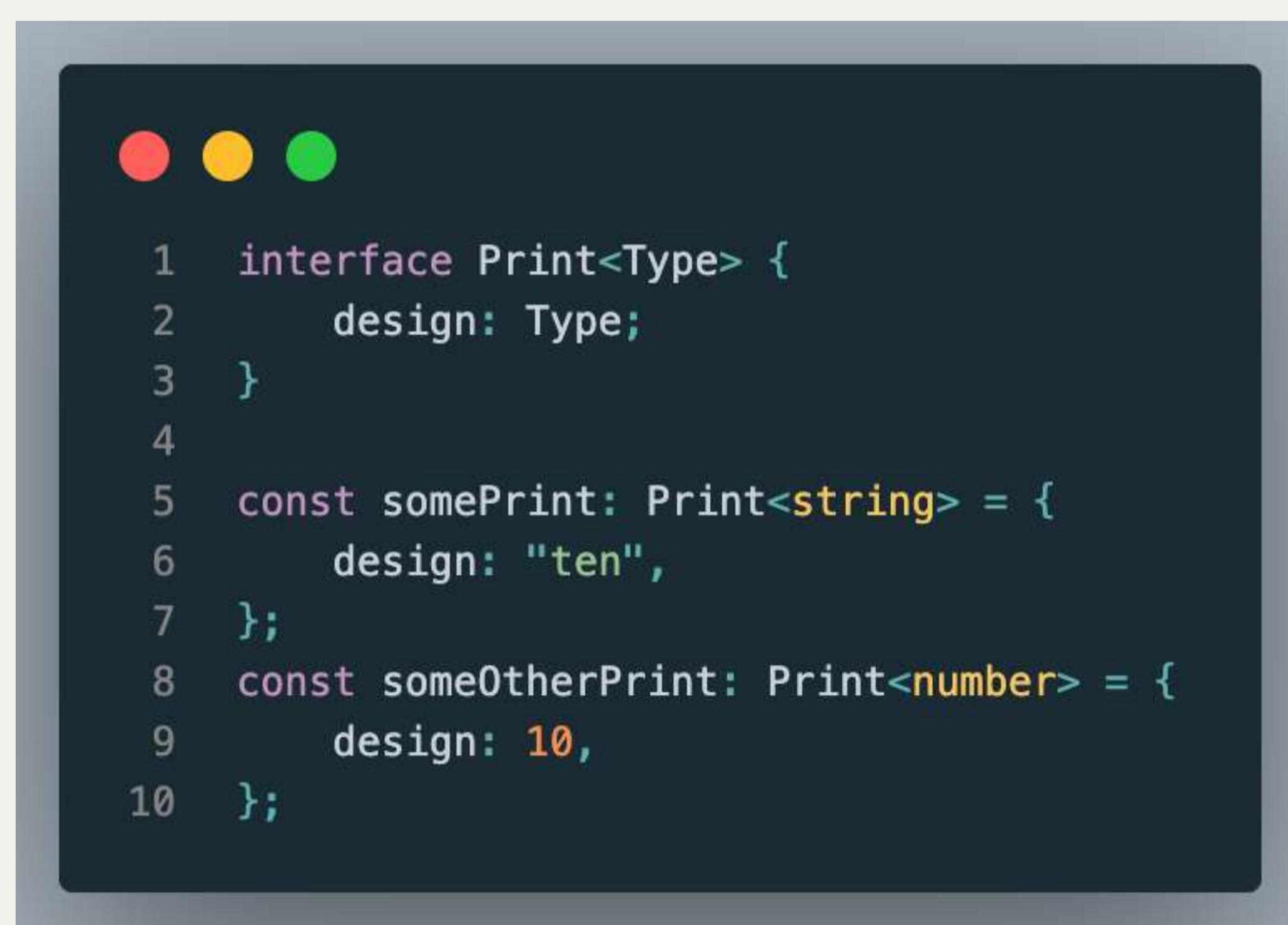
T - это и есть такая заглушка, которая записывается в угловых скобках и дальше используется для типизации аргумента и возвращаемого значения. При вызове функции мы можем подставить **любой тип** и TS на выходе будет знать, что помещается в результат
Существует **альтернативный синтаксис использования дженерика**, при вызове мы четко говорим что нужно передать в аргумент:

```
● ● ●
1 const res3 = processingData<number>(10);
```

ПРАВИЛА ИСПОЛЬЗОВАНИЯ ОБОБЩЕНИЙ

- 👉 Обобщения можно создавать **для типов, интерфейсов, функций, методов и классов**

Для **перечислений** (enum) этого делать нельзя. Про каждый из вариантов мы поговорим в других уроках. Пример с интерфейсом:



```

1  interface Print<Type> {
2      design: Type;
3  }
4
5  const somePrint: Print<string> = {
6      design: "ten",
7  };
8  const someOtherPrint: Print<number> = {
9      design: 10,
10 };

```

- 👉 Для дженериков можно использовать **любые названия** и обозначения. Все зависит от сложности:

В простых случаях, обычно, используются буквы **T, U, V, S**. Если там будет property – то **P**, если ключ/значение – то **K/V**

Идентификаторы всегда прописываются с **большой буквы**

В сложных случаях, обычно, эту заглушку описывают более подробно:

ReferralSystem<UserID, UserReferrals>

(количество неограничено, прописываются через запятую)

- 👉 Существуют **встроенные в TS дженерики** для самых разных целей. Мы касались этой темы при изучении `readonly` и вот пример создания массива через них:



```

1  Array<T>
2  const arrOfStrings: Array<string> = ['Hello', 'World!'];

```

На самом деле, когда мы прописываем `string[]`, то это удобное сокращение от `Array<string>`, которое срабатывает внутри TS

ФУНКЦИИ-ОБОБЩЕНИЯ

Для создания второго (и следующих) аргумента, который тоже будет определен во время вызова функции, достаточно прописать его через запятую:

```
● ● ●
1 function processingData<T, S>(data: T, options: S): T {
2     return data;
3 }
4
5 const res1 = processingData(1, "fast");
6 const res2 = processingData("1", "slow");
7 const res3 = processingData<number, string>(10, "slow");
```

Для выполнения разных операций с разными типами данных используем **сужение типов**:

```
● ● ●
1 function processingData<T, S>(data: T, options: S): string {
2     switch (typeof data) {
3         case "string":
4             return `${data}, speed: ${options}`;
5         case "number":
6             return `${data.toFixed()}, speed: ${options}`;
7         default:
8             return "Not valid";
9     }
10 }
```

Из заглушки можно сформировать **массив** и им типизировать **аргументы**. В таком случае передаем массив нужных типов:

```
● ● ●
1 function processingData<T, S>(data: T[], options: S): T[] {
2     return data;
3 }
4 let res1 = processingData([1.004], "fast");
5 let res2 = processingData(["1"], "slow");
6 const res3 = processingData<number, string>([10], "slow");
```

Для методов точно такие же правила, как и для функций. А если они описаны в интерфейсе, то есть три варианта синтаксиса:

```
● ● ●
1 interface DataSaver {
2     processing: <T>(data: T) => T;
3 }
4
5 const saver: DataSaver = {
6     processing: <T>(data: T) => {
7         // ... Some actions
8         console.log(data);
9         return data;
10    },
11 };
12
13 saver.processing(5);
```

```
● ● ●
1 interface DataSaver {
2     processing: <T>(data: T) => T;
3 }
4
5 const saver: DataSaver = {
6     processing(data) {
7         // ... Some actions
8         console.log(data);
9         return data;
10    },
11 };
12
13 saver.processing(5);
```

```
● ● ●
1 interface DataSaver {
2     processing: <T>(data: T) => T;
3 }
4
5 const saver: DataSaver = {
6     processing: (data) => {
7         // ... Some actions
8         console.log(data);
9         return data;
10    },
11 };
12
13 saver.processing(5);
```

ФУНКЦИИ-ОБОБЩЕНИЯ КАК АННОТАЦИИ

Функции-обобщения можно использовать в качестве аннотаций. Например, создаем отдельную функцию-шаблон, а потом используем в нужных местах:

```
● ● ●  
1 function processing<T>(data: T): T {  
2     return data;  
3 }  
4  
5 let newFunc: <T>(data: T) => T = processing;  
6  
7 interface DataSaver {  
8     processing: <T>(data: T) => T;  
9 }  
10  
11 const saver: DataSaver = {  
12     processing: processing  
13 };
```

Если мы хотим оптимизировать саму аннотацию, то можно вынести её в отдельный интерфейс:

```
● ● ●  
1 interface ProcessingFn {  
2     <T>(data: T): T  
3 }  
4  
5 function processing<T>(data: T): T {  
6     return data;  
7 }  
8  
9 let newFunc: ProcessingFn = processing;  
10  
11 interface DataSaver {  
12     processing: ProcessingFn;  
13 }
```

ОБОБЩЕННЫЕ TYPES И ИНТЕРФЕЙСЫ

Создание **обобщенного type** так же позволяет подставлять нужный тип уже во время использования. Синтаксис использует все те же идентификаторы в угловых скобках:

```
1 type User<T> = {  
2   login: T;  
3   age: number;  
4 };
```

Синтаксис

```
1 const user: User<string> = {  
2   login: 'str',  
3   age: 54  
4 }
```

Использование

```
1 const user: User<'str'> = {  
2   login: 'str',  
3   age: 54  
4 }
```

Использование с литералом

Type так же позволяет создавать **generic helper types** за счет поддержки **литеральных значений**:

```
1 type OrNull<Type> = Type | null;  
2  
3 type OneOrMany<Type> = Type | Type[];  
4  
5 const data: OneOrMany<number[]> = [5];
```

В целом, обобщенные type в практике вы будете встречать **редко**. Куда чаще используются интерфейсы, которые позволяют помещать в объекты разные данные:

```
1 interface User<T> {  
2   login: T;  
3   age: number;  
4 };
```

```
1 interface User<ParentsData> {  
2   login: string;  
3   age: number;  
4   parents: ParentsData  
5 };
```

Урок: Generics types and interfaces, constraints

GENERIC CONSTRAINTS

То, что мы можем передать в дженерик **любые данные** - это и преимущество и недостаток. В обобщенных функциях мы использовали сужение типов для работы с разными данными, но для ограничения входящих данных есть механизм **ограничений**, **generic constraints**

Проблема следующего кода в том, что при создании объекта мы можем поместить в свойство `parents` все что угодно. А по задумке это должен быть объект со свойствами **mother** и **father**:

```

1 interface User<ParentsData> {
2   login: string;
3   age: number;
4   parents: ParentsData
5 };
6
7 const user: User<{mother: string, father: string}> = {
8   login: 'str',
9   age: 54,
10  parents: {mother: 'Anna', father: 'no data'}
11}

```

Эту проблему можно решить, если создать отдельный интерфейс и типизировать свойство `parents`. **Но**, проблема в том, что тогда в этот объект **не сможет** попасть никакое другое свойство. А мы бы хотели сделать его **расширяемым**, но с **двумя обязательными** свойствами:

```

1 interface ParentsOfUser {
2   mother: string; father: string
3 }
4
5 interface User {
6   login: string;
7   age: number;
8   parents: ParentsOfUser;
9 }
10
11 const user: User = {
12   login: "str",
13   age: 54,
14   parents: { mother: "Anna", father: "no data" } // Никаких других свойств
15 };

```

Для решения этой задачи и создан механизм ограничения, который позволит “ограничить” идентификатор в дженерике. В данном случае мы можем сказать, что он будет **только объектом любого размера с двумя обязательными свойствами**. Для этого используем `extends`:

```

1 interface ParentsOfUser {
2   mother: string;
3   father: string;
4 }
5
6 interface User<ParentsData extends ParentsOfUser> {
7   login: string;
8   age: number;
9   parents: ParentsData;
10 }
11
12 const user: User<{ mother: string; father: string; married: boolean }> = {
13   login: "str",
14   age: 54,
15   parents: { mother: "Anna", father: "no data", married: true },
16 };

```

Урок: Generics types and interfaces, constraints

Для ограничений можно использовать и **примитивные типы**, в том числе и union. Например функция, которая принимает только строку или число:

```
● ● ●  
1 const depositMoney = <T extends number | string>(amount: T): T => {  
2   console.log(`req to server with amount: ${amount}`)  
3   return amount;  
4 }  
5  
6 depositMoney(500);  
7 depositMoney('500');  
8 depositMoney(true) // Error
```

Альтернативный вариант с использованием обычного union типа также можно использовать. Но тут будет **повторение** кода:

```
● ● ●  
1 const depositMoney = (amount: number | string): number | string => {  
2   console.log(`req to server with amount: ${amount}`);  
3   return amount;  
4 };  
5  
6 depositMoney(500);  
7 depositMoney("500");
```

ОБОБЩЕННЫЕ КЛАССЫ

Обобщенные классы действуют по тому же принципу: абстрактные идентификаторы при создании экземпляра будут заменены чем угодно:

```
● ● ●
1 class User<T, S> {
2     name: T;
3     age: S;
4
5     constructor(name: T, age: S) {
6         this.name = name;
7         this.age = age;
8     }
9 }
10 const ivan = new User("Ivan", 30);
11 console.log(ivan);
```

Альтернативный синтаксис позволяет прописать типы вручную при создании экземпляра. Это полезно, когда данные приходят из **других источников**, а не прописаны вручную:

```
● ● ●
1 const ivan = new User<string, number>("Ivan", 30); // Не очень полезно
2 const nameData = 'Alex';
3 const ageData = 50;
4 const alex = new User<string, number>(nameData, ageData); // Теперь неправильные данные не пройдут
```

Методы классов могут быть дженериками. Причем одинаковые названия идентификаторов **не пересекаются**:

```
● ● ●
1 class User<T, S> {
2     name: T;
3     age: S;
4
5     constructor(name: T, age: S) {
6         this.name = name;
7         this.age = age;
8     }
9
10    sayMyFullName<T>(surname: T): string {
11        if (typeof surname !== "string") {
12            return `I have only name: ${this.name}`;
13        } else {
14            return `${this.name} ${surname}`;
15        }
16    }
17 }
```

При наследовании классов мы должны указывать то, **какими типами данных были идентификаторы у родительского класса**. Только так мы можем передавать их наследнику. Сам же наследник тоже может быть дженериком:

```
● ● ●
1 class AdminUser<T> extends User<string, number> {
2     rules: T;
3 }
```

ВСТРОЕННЫЕ ОБОБЩЕНИЯ

В TS существует довольно много **встроенных дженериков**, часть из которых для внутренних нужд языка, а часть для рутинных задач в разработке. Первый, с которым мы знакомились - это дженерик, запрещающий менять данные и его разновидность ReadonlyArray:



```

1 const arr: Array<number> = [1, 2, 3]; // Обычный массив
2
3 const roarr: ReadonlyArray<string> = ["dsds"]; // Неизменяемый массив

```



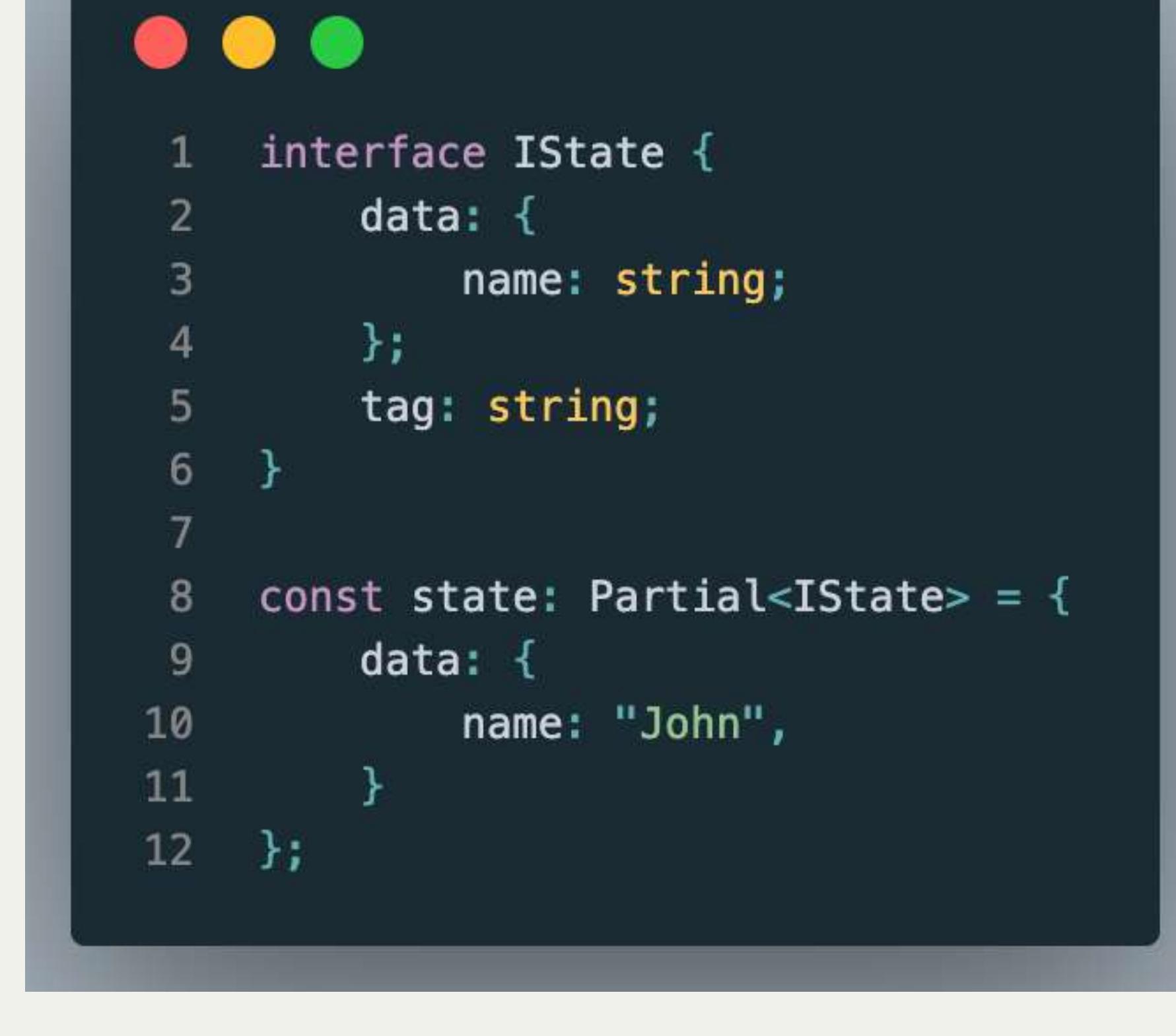
```

1 interface IState {
2   data: {};
3   tag: string;
4 }
5
6 function action(state: Readonly<IState>) {
7   state.data = ""; // Error
8 }

```

В варианте выше Readonly дженерик запрещает изменения только на **первом уровне вложенности** в объекте. Свойства внутри state.data менять уже можно

Дженерик **Partial** добавляет всем свойствам объекта модификатор вопросительного знака (optional), делая их **необязательными**:

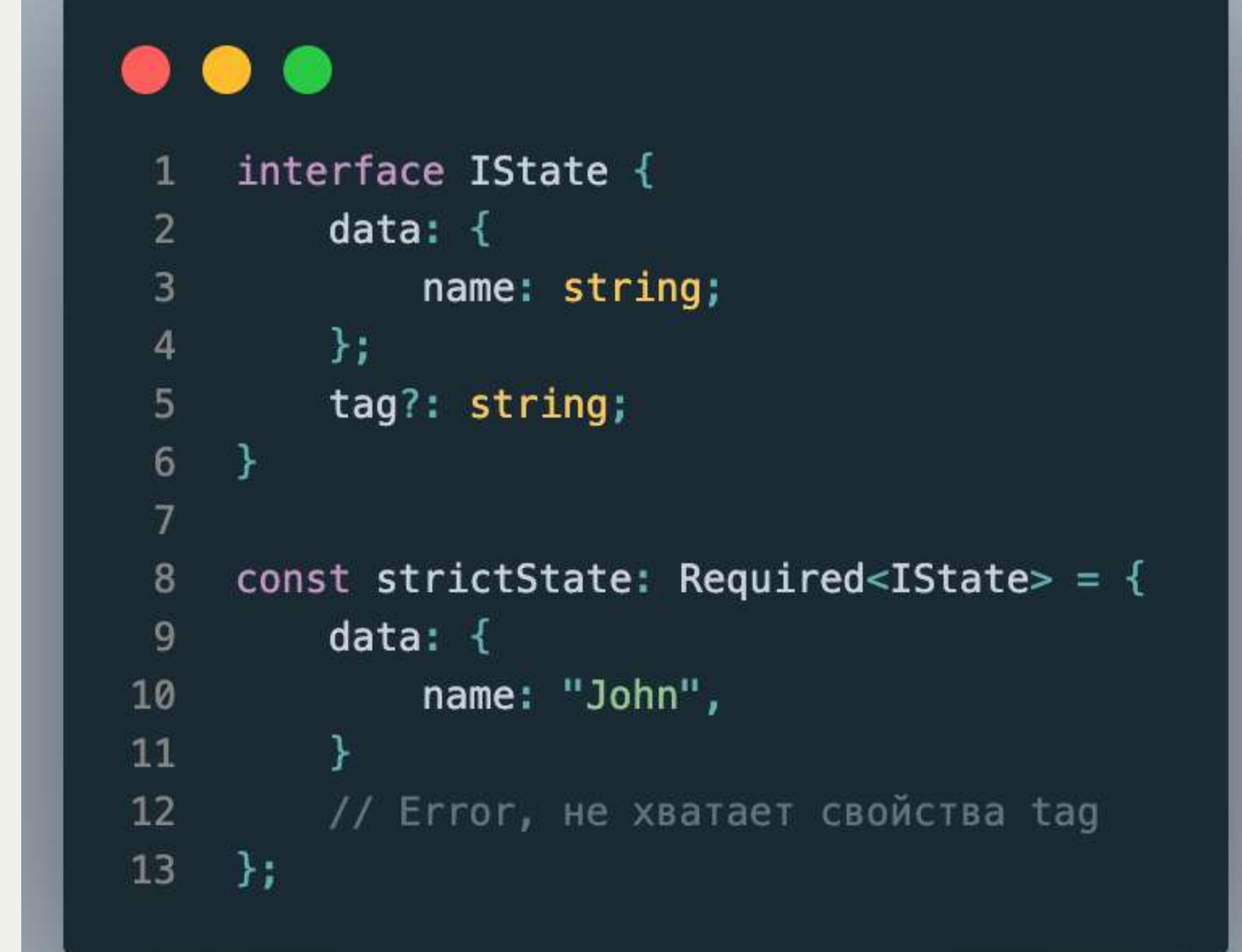


```

1 interface IState {
2   data: {
3     name: string;
4   };
5   tag: string;
6 }
7
8 const state: Partial<IState> = {
9   data: {
10     name: "John",
11   }
12 };

```

Дженерик **Required** - это полная противоположность **Partial**. Он берет объект и удаляет у всех свойств модификатор необязательности (optional), делает все поля **обязательными**:



```

1 interface IState {
2   data: {
3     name: string;
4   };
5   tag?: string;
6 }
7
8 const strictState: Required<IState> = {
9   data: {
10     name: "John",
11   }
12   // Error, не хватает свойства tag
13 };

```

Строго говоря, эти дженерики **более правильно называть типами**, но на уровне кода вы уже знаете что это такое. Большую часть из них мы изучим дальше в уроках

ПОЛУЧИТЬ КЛЮЧИ ОБЪЕКТНОГО ТИПА

keyof - это оператор, позволяющий получить все свойства объекта, его ключи. Мы работаем **с типами**, а не с литералами, так что его можно применить к **interface**, **type** и **class**:

```
1 interface ICompany {
2   name: string;
3   debts: number;
4 }
5
6 type CompanyKeys = keyof ICompany;
7 const keys: CompanyKeys = 'debts'; // 'debts' или 'name'
```

Результат работы - это **union type** из свойств. Часто такой прием используется с **дженериками** для того, чтобы создать связь между типом и его свойствами. В таком случае мы сможем использовать **только существующие** свойства и получим подсказки:

```
1 function printDebts<T, K extends keyof T, S extends keyof T>(
2   company: T,
3   name: K,
4   debts: S
5 ) {
6   console.log(`Company ${company[name]}, debts: ${company[debts]}`);
7 }
8
9 const google: ICompany = {
10   name: "Google",
11   debts: 500000,
12 };
13
14 printDebts(google, "name", 'debts'); // аргументы №2 и №3 строго ограничены свойствами
```

ПОЛУЧЕНИЕ ТИПА

typeof - это оператор, позволяющий получить тип определенной сущности. Мы его уже использовали для механизма **сужения типов** и **запроса типов**:

```
1 function printMsg(msg: string[] | number | boolean): void {
2   if (Array.isArray(msg)) {
3     msg.forEach(m => console.log(m));
4   } else if (typeof msg === "number") {
5     console.log(msg);
6   } else {
7     console.log(msg);
8   }
9   console.log(msg);
10 }
```

```
1 const dataFromControl = {
2   water: 200,
3   el: 350,
4 };
5
6 function checkReadings(data: typeof dataFromControl): boolean {
7   const dataFromUser = {
8     water: 200,
9     el: 350,
10   };
11   // ...
12 }
```

В теме манипуляций с типами он играет большую роль за счет того, что позволяет получить **тип любого литерала** (конкретного значения) и работать дальше с ним как с **типовом**:

```
1 const google = {
2   name: "Google",
3   debts: 500000,
4 };
5
6 type GoogleKeys = keyof google; // Error! google - это объект, а не тип
```

```
1 const google = {
2   name: "Google",
3   debts: 500000,
4 };
5
6 type GoogleKeys = keyof typeof google; // "name" | "debts"
```

Когда необходимо поработать с чем-то, **как с типом**, то сначала применяем на нем `typeof`, а после уже выполняем манипуляции

Урок: Indexed Access Types

INDEXED ACCESS TYPES

Для получения **типа значения** в определенном свойстве используется прием **Indexed Access Types** (дословно: получение типа по индексу):

```

1 interface ICompany {
2   name: string;
3   debts: number;
4 }
5
6 type CompanyDebtsType = ICompany["debts"]; // number

```

При использовании будем получать подсказки о доступных свойствах. Можно применять на объектных типах, в том числе и для получения значений **вложенных свойств**:

```

1 interface ICompany {
2   name: string;
3   debts: number;
4   management: {
5     owner: string;
6   };
7 }
8
9 type CompanyOwnerType = ICompany["management"]["owner"]; // string

```

Получение **одного конкретного типа** из массива типов:

```

1 interface ICompany {
2   name: string;
3   debts: number;
4   departments: Department[];
5 }
6
7 type CompanyDepartmentsType = ICompany["departments"][number]; // Department

```

Для использования переменной при получении типа используется оператор **typeof**. Алгоритм такой же: получаем **тип** из переменной и применяем его:

```

1 interface ICompany {
2   name: string;
3   debts: number;
4 }
5 const debts = "debts";
6 type CompanyDebtsType = ICompany[typeof debts]; // number

```

Если переменная объявлена как **let**, то её сначала нужно привести к **литеральному типу**:

```

1 let debts: "debts" = "debts";
2 // Или
3 let debts = "debts" as "debts";

```

ФОРМИРУЕМ ТИПЫ ЧЕРЕЗ УСЛОВИЕ

Типы можно формировать при помощи **тернарного оператора**, где мы дословно спрашиваем: “этот тип наследуется(или совместим с) от этого? Если да - будет этот тип, нет - другой”:

`SomeType extends OtherType ? TrueType : FalseType;`



```
1 type Example = "string" extends "Hello" ? string : number; // number, так как литеральные типы разные
```

- 👉 Условные типы всегда предполагают использование ограничения, то есть ключевого слова **extends**. Проверяемый тип должен чем-то ограничиваться для проверки. В целом, это и есть условие
- 👉 Мы работаем с типами. При использовании литерала будет **ошибка**. Конкретные значения сначала необходимо преобразовать в тип с помощью оператора **typeof**:



```
1 const str: string = "Hello";
2 type Example = "string" extends str ? string : number; // Error
3 type Example = "string" extends typeof str ? string : number; // Ok, string, тк тип переменной string
```

- 👉 Базовый синтаксис не очень полезен в работе. Поэтому условные типы вы почти всегда встретите в комбинации с дженериками. Именно там раскрывается главная суть:



```
1 type FromUserOrFromBase<T extends string | number> = T extends string
2     ? IDataFromUser
3     : IDataFromBase;
4
5 interface User<T extends "created" | Date> {
6     created: T extends "created" ? "created" : Date;
7 }
```

РЕШЕНИЕ ОШИБКИ В ФУНКЦИЯХ

В примере ниже TS **не может определить**, что же функция в итоге должна вернуть. Эта информация появится только на этапе запуска функции, ведь она зависит **от типа приходящего аргумента**. Отсюда ошибка в том, что компилятор не сможет сопоставить **тип возвращаемого значения с типом, возвращаемым в теле функции**:

```
● ● ●  
1  function calculateDailyCalories<T extends string | number>(  
2      numOrStr: T  
3  ): T extends string ? IDataFromUser : IDataFromBase {  
4      if (typeof numOrStr === "string") {  
5          const obj: IDataFromUser = {  
6              weight: numOrStr,  
7          };  
8          return obj; // Error  
9      } else {  
10         const obj: IDataFromBase = {  
11             calories: numOrStr,  
12         };  
13         return obj; // Error  
14     }  
15 }
```

Такое поведение **соответствует дизайну TS**. Решить её можно прямым указанием того, чем является возвращаемое значение в теле функции:

```
● ● ●  
1  return obj as T extends string ? IDataFromUser : IDataFromBase; // Ok
```

INFER И ВЕТКИ УСЛОВИЙ

Формирование условных типов можно **разветвлять**, добиваясь нужного результата и комбинируя условия:

```
● ● ●  
1 type GetStringType<T extends "hello" | "world" | string> = T extends "hello"  
2     ? "hello"  
3     : T extends "world"  
4     ? "world"  
5     : string;
```

Редкий гость в TS - это оператор **infer**. Он позволяет “**вытащить**” определенный тип из какой-либо сущности:

```
● ● ●  
1 type GetFirstType<T> = T extends Array<infer First> ? First : T;  
2  
3 type Ex = GetFirstType<number[]>; // number
```

```
● ● ●  
1 type UnwrappedPromise<T> = T extends Promise<infer Return> ? Return : T;
```

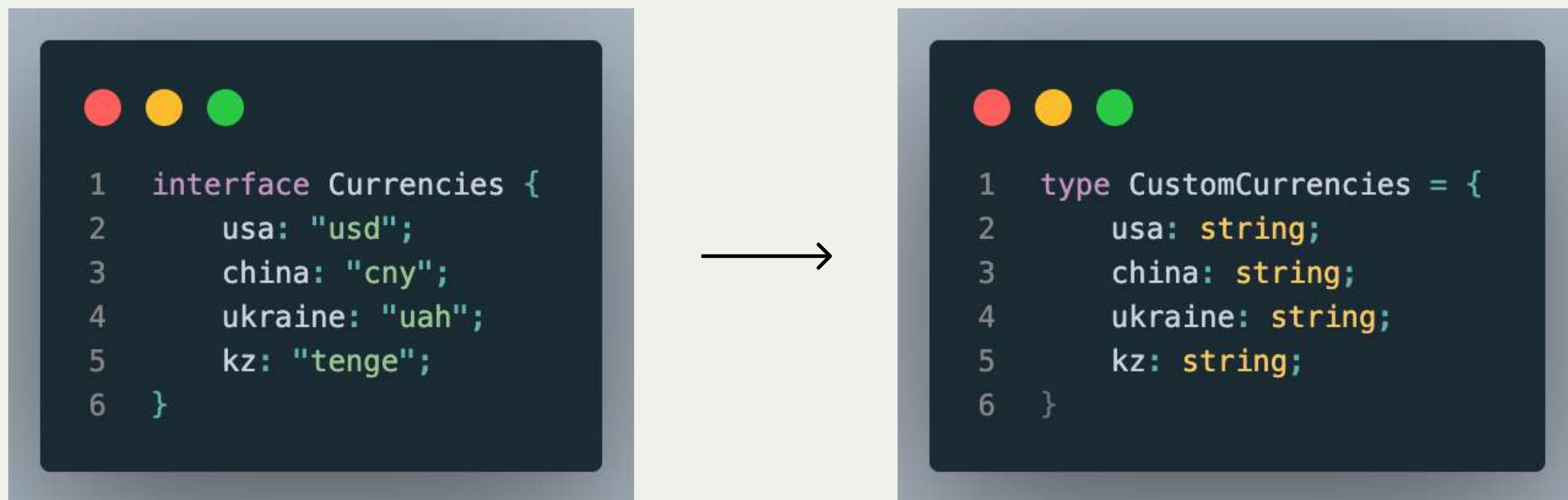
Реализация типа, принимающего любой тип и возвращающего массив этого типа:

```
● ● ●  
1 type ToArray<Type> = Type extends any ? Type[] : never;  
2 type ExArray = ToArray<number>; // number[]  
3 type ExArray2 = ToArray<Ex | string>; // number[] | string[]
```

Прием, когда как аргумент передается юнион тип, иногда называется распределительные условные типы.

MAPPED TYPES

TS позволяет формировать объектные типы путем **перебора и модификации исходного типа**. Этот механизм называется **Mapped types** (**сопоставление типов**). К примеру, мы можем сформировать:



Создание вручную такого типа ведет к нескольким проблемам:

- ! Если в списке будут все страны мира, то вы потратите огромное количество времени на изменение данных **вручную**
- ! При этом копия будет занимать огромное количество места в коде
- ! Но главное то, что у вас не будет **зависимости** одного типа от другого. Если вы удалите свойство в целевом интерфейсе, то это никак не повлияет на копию

Для оптимизации работы воспользуемся **Mapped types**, синтаксис которых:

```

type СопоставимыйТип = {
  [ПроизвольныйИдентификатор in Множество]: ПроизвольныйТипДанных;
};

```

Сформированный тип должен быть обязательно задан через **type!**
В самом простом варианте его можно применять к обычным литералам:



```

1 type Keys = "name" | "age" | "role";
2
3 type User = {
4   [K in Keys]: string;
5 };
6
7 const alex: User = {
8   name: "Alex",
9   age: "25",
10  role: "admin",
11 };

```

Но в практике чаще всего **mapped types** комбинируется с дженериками. Для создания нового типа на базе интерфейса валют создадим тип:



```

1 type CreateCustomCurr<T> = {
2     [P in keyof T]: string;
3 };
4
5 type CustomCurrencies = CreateCustomCurr<Currencies>;

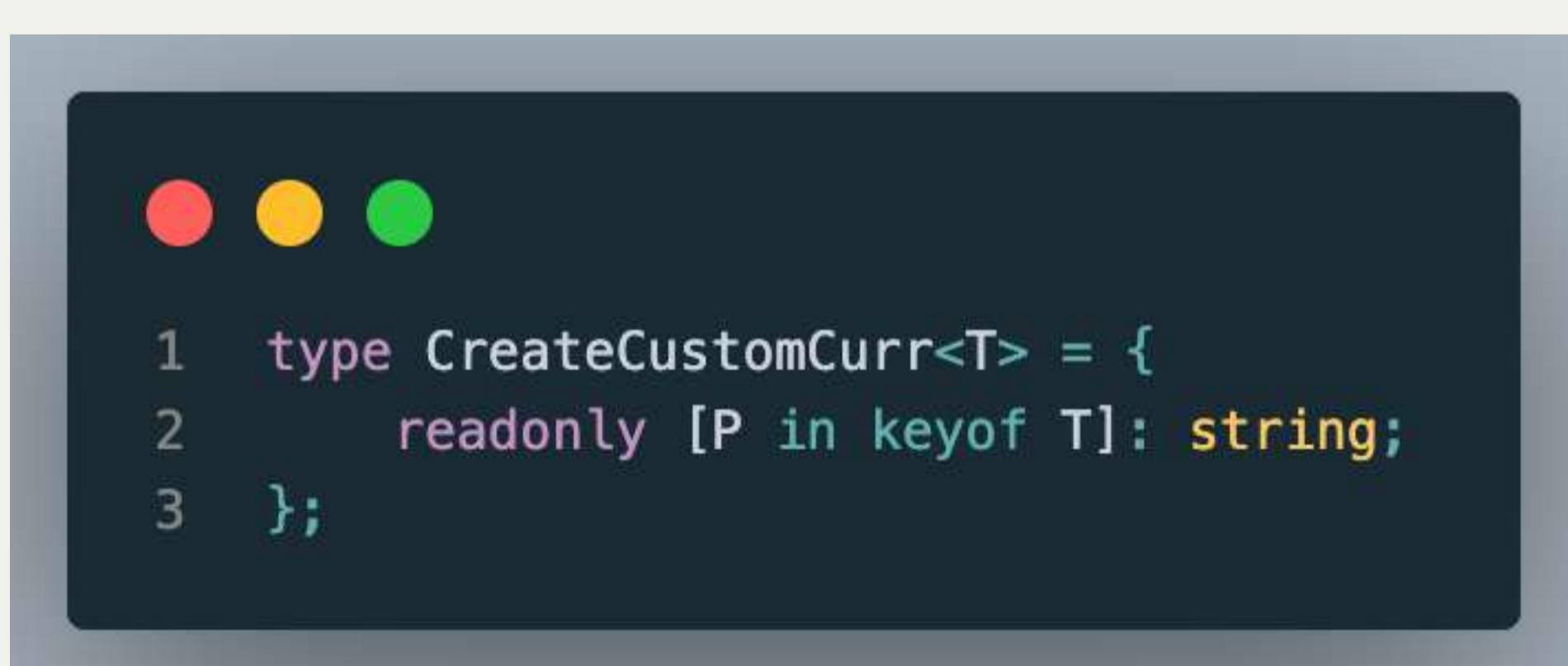
```

Где **P** - это свойства, которые берутся из ключей приходящего в дженерик типа. **keyof T** - получение этих ключей. Вместо **string** может быть **какой угодно тип**, необходимый вам.

Таким образом мы установили **связь** между типами и удаление одного из свойств в **Currencies** приведет к изменению типа **CustomCurrencies**

МОДИФИКАЦИИ И ОПЕРАТОРЫ +/-

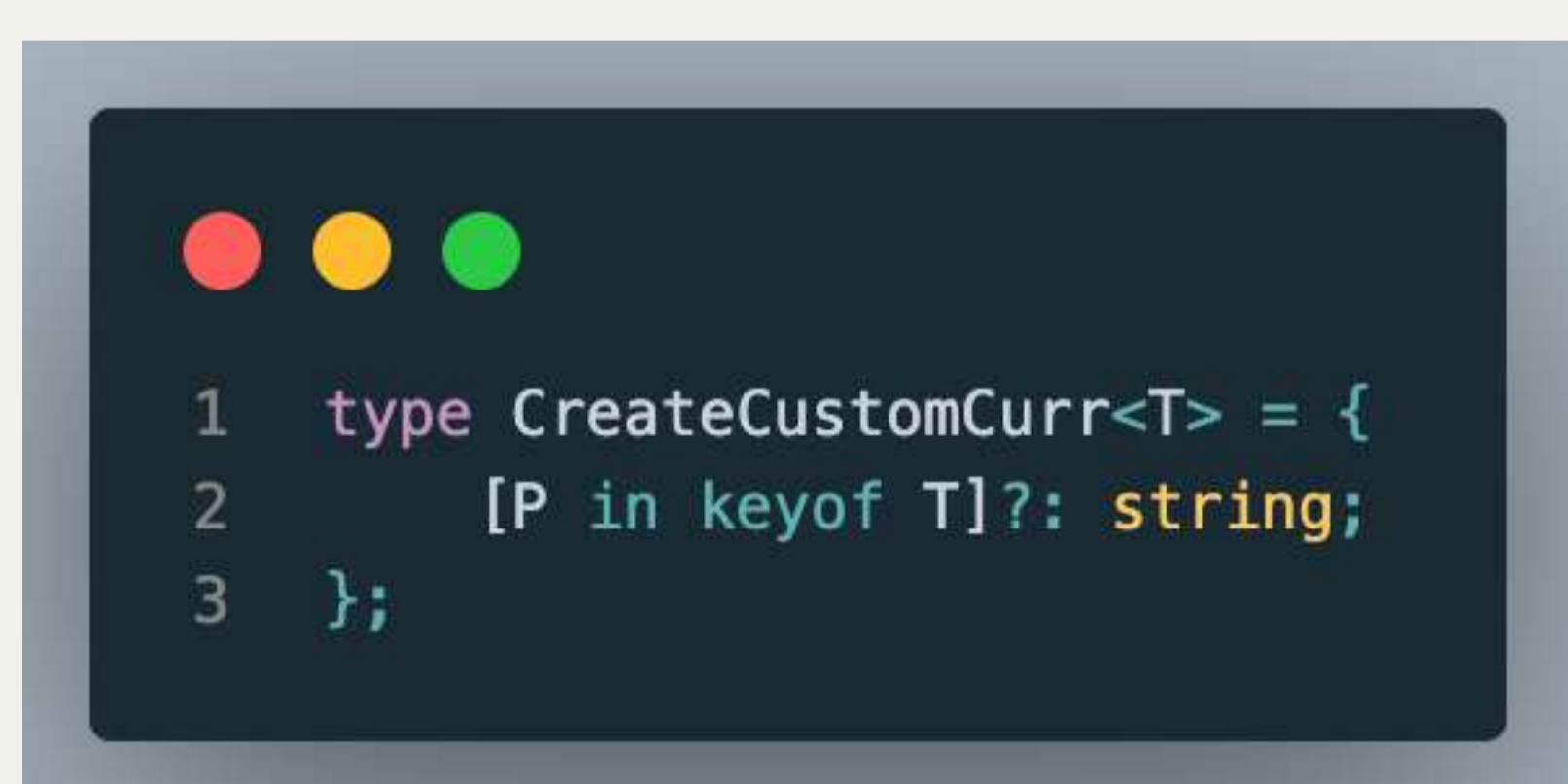
Во время формирования нового типа к свойствам можно добавлять модификаторы **readonly** и/или **optional**:



```

1 type CreateCustomCurr<T> = {
2     readonly [P in keyof T]: string;
3 };

```



```

1 type CreateCustomCurr<T> = {
2     [P in keyof T]?: string;
3 };

```

Так же существуют операторы “+” и “-”, которые добавляют или убирают эти модификаторы из **исходного** типа. Оператор “+” аналогичен записи выше, когда идет простое добавление



```

1 type CreateCustomCurr<T> = {
2     +readonly [P in keyof T]-?: string;
3 };

```

TEMPLATE LITERAL TYPES

При создании новых типов есть механизм, похожий на стандартную интерполяцию строк в JS. Он позволяет формировать **литеральные типы по тем же синтаксическим правилам**(косые кавычки, обрамление переменных в \${}):

```
1 type MyAnimation = "fade";
2
3 type MyNewAnimation = `${MyAnimation}In`; // "fadeIn"
```

При использовании **union type** будет формироваться так же тот же тип, но с модификациями:

```
1 type MyAnimation = "fade" | "swipe";
2
3 type MyNewAnimation = `${MyAnimation}In`; // "fadeIn" | "swipeIn"
```

```
1 type MyAnimation = "fade" | "swipe";
2 type Direction = "in" | "out";
3
4 type MyNewAnimation = `${MyAnimation}${Direction}`; // "fadein" | "fadeout" | "swipein" | "swipeout"
```

Для удобных модификаций были добавлены специальные дженерики по работе со строками. Передача в них **не строки** ведет к ошибке

Uppercase<StringType>
Lowercase<StringType>
Capitalize<StringType>
Uncapitalize<StringType>

```
1 type MyAnimation = "fade" | "swipe";
2 type Direction = "in" | "out";
3
4 type MyNewAnimation = `${MyAnimation}${Capitalize<Direction>}`; // "fadeIn" | "fadeOut" | "swipeIn" | "swipeOut"
```

Этот механизм можно использовать внутри **Mapped types** для изменения названий свойств:

```
1 interface Currencies {
2   usa: "usd";
3   china: "cny";
4   ukraine: "uah";
5   kz: "tenge";
6 }
7
8 type CreateCustomCurr<T> = {
9   [P in keyof T as `custom${Capitalize<string & P>}`]: string;
10 };
11
12 type CustomCurrencies = CreateCustomCurr<Currencies>;
```

ВСТРОЕННЫЕ ВСПОМОГАТЕЛЬНЫЕ ТИПЫ

Вспомогательные типы используются для самых разных целей при формировании новых типов. Это раздел **Utility types** в документации с полным их перечнем

Тип **Omit** необходим для того, чтобы **исключить указанные свойства из другого типа**. Второй аргумент может быть только `string | number | symbol`

```
● ● ●
1 interface Currencies {
2     usa: "usd";
3     china: "cny";
4     ukraine: "uah";
5     kz: "tenge";
6 }
7
8 type CurrWithoutUSA = Omit<Currencies, "usa">; // исключение
```

Тип **Pick** необходим для **фильтрации типа по заданным свойствам**. Остаются только **указанные**. Указывать можно только существующие в целевом типе. Если их несколько - то в union типе:

```
● ● ●
1 type CurrUSAAndUkraine = Pick<Currencies, "usa" | "ukraine">; // фильтрация по свойству
```

Тип **Exclude** позволяет **убирать из union типа те, которые соответствуют условию**:

```
● ● ●
1 type CountriesWithoutUSA = Exclude<keyof Currencies, "usa">; // "china" | "ukraine" | "kz"
2
3 type MyAnimation = "fade" | "swipe";
4 type FadeType = Exclude<MyAnimation, "swipe">; // удаление из union type
```

Тип **Extract** **выбирает типы**, которые соответствуют условию. Это **Exclude** наоборот:

```
● ● ●
1 type MyAnimation = "fade" | "swipe";
2 type SwipeType = Extract<MyAnimation, "swipe">; // выбор подходящего типа, "swipe"
```

Тип **Record** позволяет **сконструировать другой тип в формате ключ-значение**. Это удобный способ сказать TS: тут будет объект, ключами которого будет это, значениями – это:

```
● ● ●
1 type PlayersNames = "alex" | "john";
2 type CustomCurrencies = CreateCustomCurr<Currencies>;
3 type GameDataCurr = Record<PlayersNames, CustomCurrencies>;
4
5 const gameData: GameDataCurr = {
6     alex: {
7         customChina: "test",
8         customKz: "test",
9         customUkraine: "test",
10        customUsa: "test",
11    },
12    john: {
13        customChina: "test",
14        customKz: "test",
15        customUkraine: "test",
16        customUsa: "test",
17    },
18};
```

ДОПОЛНИТЕЛЬНЫЕ ВСПОМОГАТЕЛЬНЫЕ ТИПЫ

Тип **ReturnType** необходим для получения типа, который **возвращает** переданная ему функция. Используется именно тип функции, так что не забываем применять `typeof`:

```
1 function calculate(a: number, b: number): number {
2     return a * b;
3 }
4
5 type CalculateRT = ReturnType<typeof calculate>; // number
```

Тип **Parameters** позволяет получать **тип аргументов функции** в виде кортежа:

```
1 function calculate(a: number, b: number): number {
2     return a * b;
3 }
4
5 type CalculatePT = Parameters<typeof calculate>; // [a: number, b: number]
6 type CalculatePTFirst = Parameters<typeof calculate>[0]; // number
```

Тип **ConstructorParameters** позволяет получать **тип аргументов в классах**:

```
1 class Example {
2     constructor(a: number) {}
3 }
4
5 type T0 = ConstructorParameters<typeof Example>; // [a: number]
```

РАБОТА С JSON И ЗАПРОСАМИ

Чтобы мы не делали, максимально типизировать парсинг json-строки в TS **мы не сможем**. Эта операция происходит в **рантайме**, то есть при запуске кода и в самой строке может быть **что угодно**. Так что такой код особого смысла не имеет:

```
1 const jsonTest = '{ "name": "Test", "data": "dfdg"}';
2
3 interface JSONTest {
4     name: string;
5     data: number;
6 }
7
8 const objFromJson: JSONTest = JSON.parse(jsonTest);
```

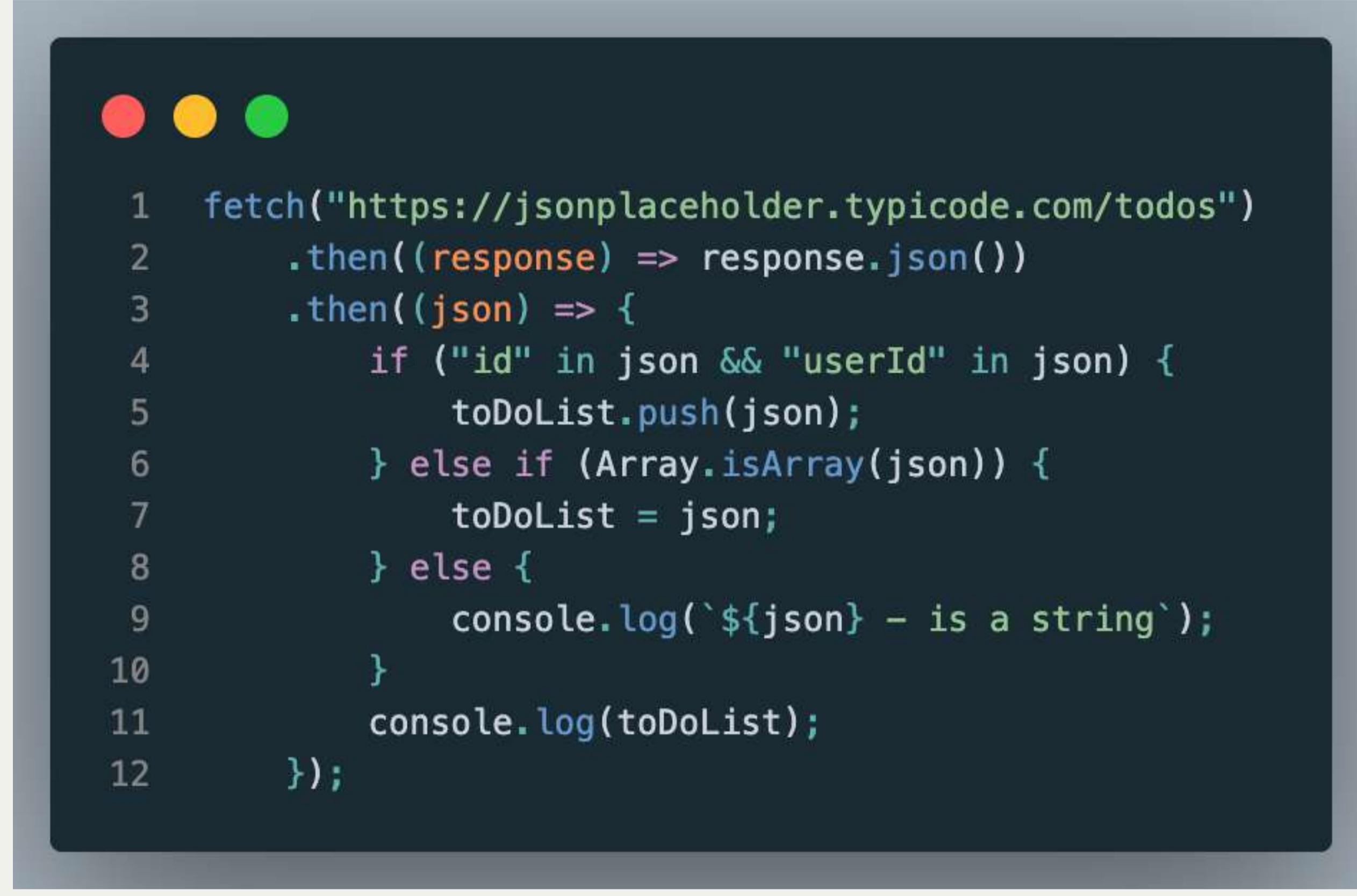
Результат работы будет **any** и подсказок никаких TS не даст. Для правильной обработки таких данных **можно использовать стандартные техники**: проверка на наличие нужных свойств, значений, типов и тп. В TS можно **вместо any вернуть unknown** и обрабатывать его через сужение типов:

```
1 const userData =
2     '{"isBirthdayData": true, "ageData": 40, "userNameData": "John"}';
3
4 function safeParse(s: string): unknown {
5     return JSON.parse(s);
6 }
7
8 const data = safeParse(userData);
9
10 function transferData(d: unknown): void {
11     if (typeof d === "string") {
12         console.log(d.toLowerCase());
13     } else if (typeof d === "object" && d) {
14         console.log(data);
15     } else {
16         console.error("Some error");
17     }
18 }
```

При работе с запросами и получении данных с сервера та же схема. **Получили данные и проверяете их**:

```
1 fetch("https://jsonplaceholder.typicode.com/todos/1")
2     .then((response) => response.json())
3     .then((json) => {
4         if ("id" in json) {
5             toDoList.push(json);
6         }
7         console.log(toDoList);
8     });
9 
```

Такие условия в запросах можно расширять по вашим нуждам.
Проверять на тип данных и содержимое в них:

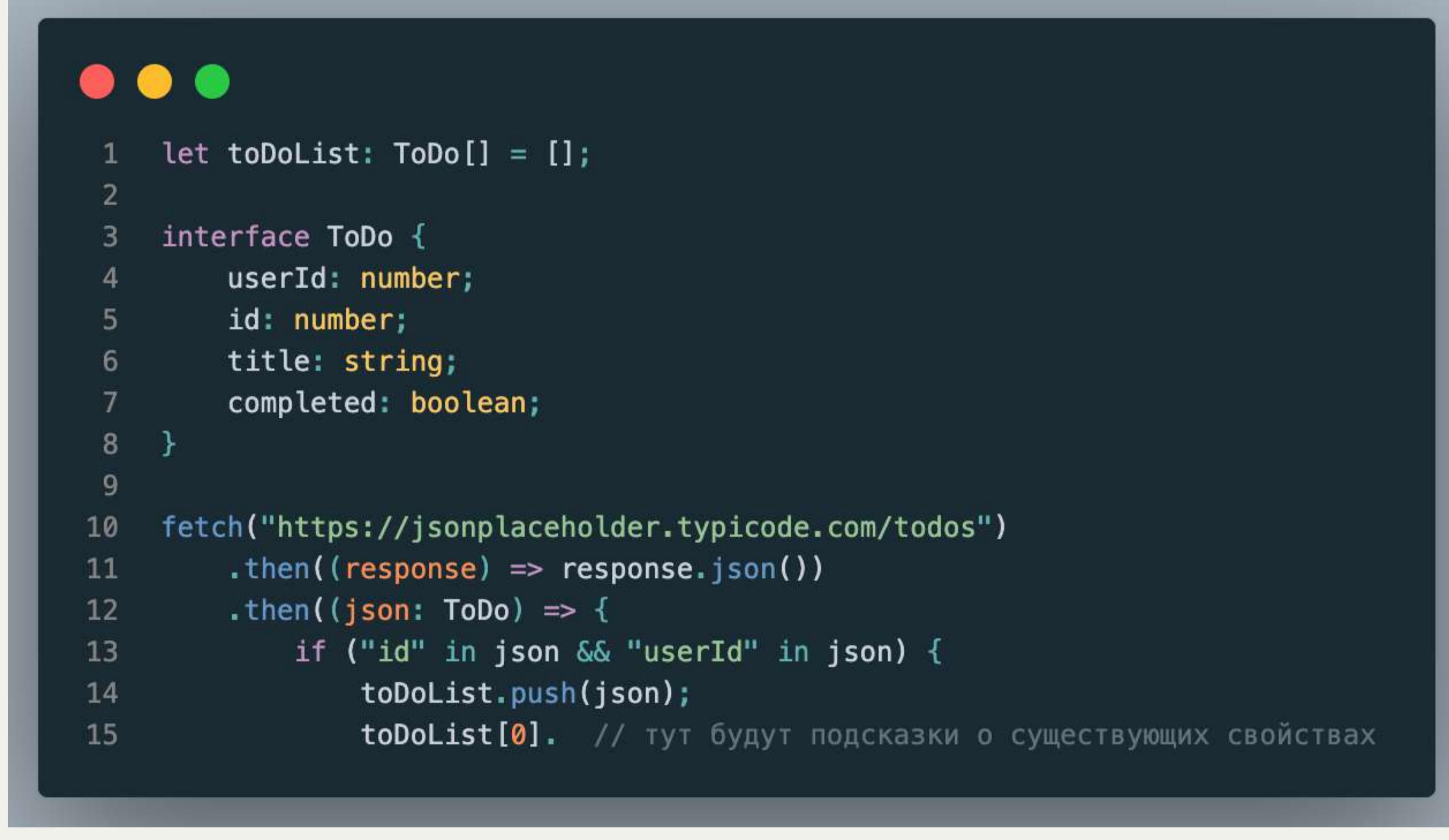


```

1  fetch("https://jsonplaceholder.typicode.com/todos")
2      .then((response) => response.json())
3      .then((json) => {
4          if ("id" in json && "userId" in json) {
5              toDoList.push(json);
6          } else if (Array.isArray(json)) {
7              toDoList = json;
8          } else {
9              console.log(`$ {json} - is a string`);
10         }
11     }
12   );

```

Типизация кода внутри .then **НИКАК НЕ УБЕРЕЖЕТ НАС ОТ ОШИБОК** в рантайме. Но она может повысить **семантику** вашего кода: сказать всем разработчикам, что именно вы тут ожидаете и получить подсказки внутри функции. Так что тут на ваше усмотрение



```

1  let toDoList: ToDo[] = [];
2
3  interface ToDo {
4      userId: number;
5      id: number;
6      title: string;
7      completed: boolean;
8  }
9
10 fetch("https://jsonplaceholder.typicode.com/todos")
11     .then((response) => response.json())
12     .then((json: ToDo) => {
13         if ("id" in json && "userId" in json) {
14             toDoList.push(json);
15             toDoList[0]. // тут будут подсказки о существующих свойствах

```

ИНТЕРФЕЙС PROMISE

При выполнении запроса с `fetch` можно обнаружить, что он нам возвращает `Promise<Response>` А значит, что в TS существует интерфейс **Promise, принимающий тот тип, который он будет возвращать при успешном выполнении**

Для примера можно создать свой пример со строкой:



```

1  const promise = new Promise<string>((resolve, reject) => {
2      resolve("Test"); // resolve принимает только string
3  });
4
5  promise.then((value) => {
6      console.log(value.toLowerCase()); // value - string
7  });

```

Во время запроса TS автоматически подставляет туда интерфейс `Response`, который содержит методы `.json()` и тп.

Урок: Awaited

AWAITED

Вспомогательный тип `Awaited` был введен в язык последним и нужен для формирования типа, который возвращает указанный промис:

```
1 type FromPromise = Awaited<Promise<number>>; // number, а не Promise<number>
```

Этот тип также позволяет рекурсивно раскрывать промисы:

```
1 type FromPromise = Awaited<Promise<Promise<number>>>; // number
```

Пример использования для получения типа:

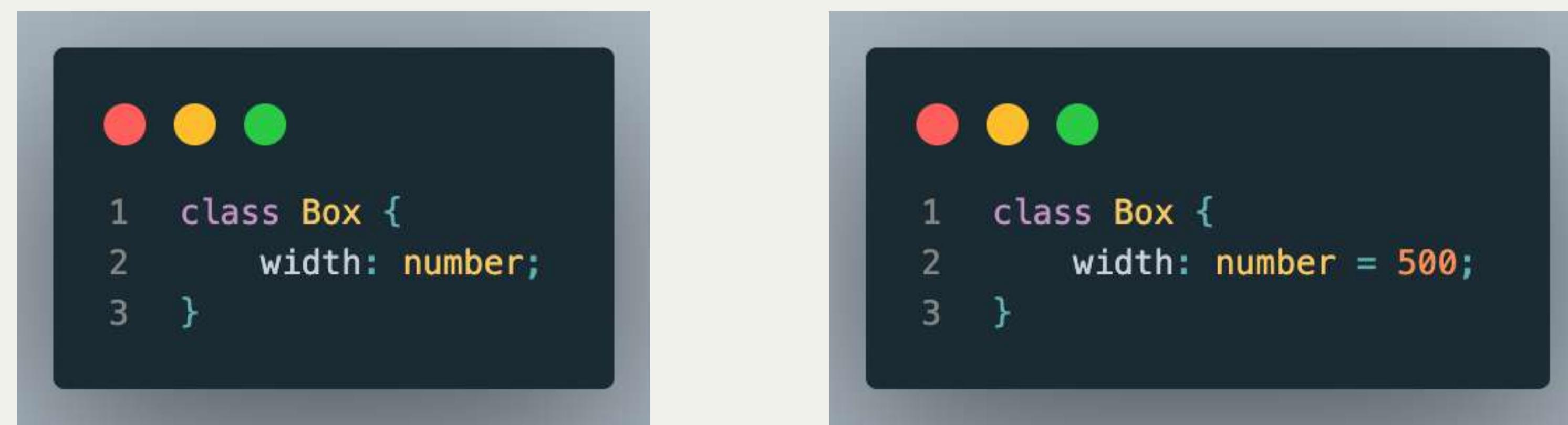
```
1 interface User {
2     name: string;
3 }
4
5 async function fetchUsers(): Promise<User[]> {
6     const users: User[] = [
7         {
8             name: "Alex",
9         },
10    ];
11
12    return users;
13 }
14
15 const users = fetchUsers(); // users type Promise<User[]>
16
17 type FetchUsersReturnType = Awaited<ReturnType<typeof fetchUsers>>; // User[]
```

До версии 4.5 в коде TS вы можете встретить ручное создание этого типа:

```
1 type UnwrappedPromise<T> = T extends Promise<infer Return> ? Return : T;
2 type FetchDataReturnType = UnwrappedPromise<ReturnType<typeof fetchUsers>>;
```

БАЗОВЫЙ СИНТАКСИС КЛАССОВ В ТС

Классы в ТС создаются **по тем же правилам**, что и в JS, но с учетом типизации. Она проникла почти во все возможности, так что стоит их разобрать по отдельности. Современный ТС использует возможность “**Class fields**”, которая позволяет, кроме всего прочего, создавать свойства сразу без конструктора:



```
1 class Box {  
2     width: number;  
3 }  
  
1 class Box {  
2     width: number = 500;  
3 }
```

Сразу создали свойство и указали его тип. Можно задать значение по умолчанию прямо тут

При первоначальной настройке такой код будет **предупреждать об ошибке**. По умолчанию, каждое свойство должно быть проинициализировано в конструкторе. На практике это необходимо далеко не всегда, так что есть два способа избавиться от ошибки:

- 👉 Использовать оператор “!”, чтобы убедить компилятор, что мы знаем что делаем: `width!: number;`;
- 👉 В файле `tsconfig.json` найти свойство `strictPropertyInitialization` и установить его в позицию **false**

В реальных проектах последняя опция часто выключена, так как свойства инициализируются другими способами

Далее можно создать конструктор или/и другие методы для работы с учетом типизации. В конструкторе так же можно устанавливать начальное значение, но способ выше удобнее:



```
1 class Box {  
2     width: number;  
3     height: number;  
4  
5     constructor(width: number) {  
6         this.width = width;  
7         this.height = 500;  
8     }  
9 }
```

Важный нюанс! Для того, чтобы применялись все настройки файла `tsconfig.json`, необходимо запускать команду `tsc` без добавления аргументов, в том числе пути к файлу

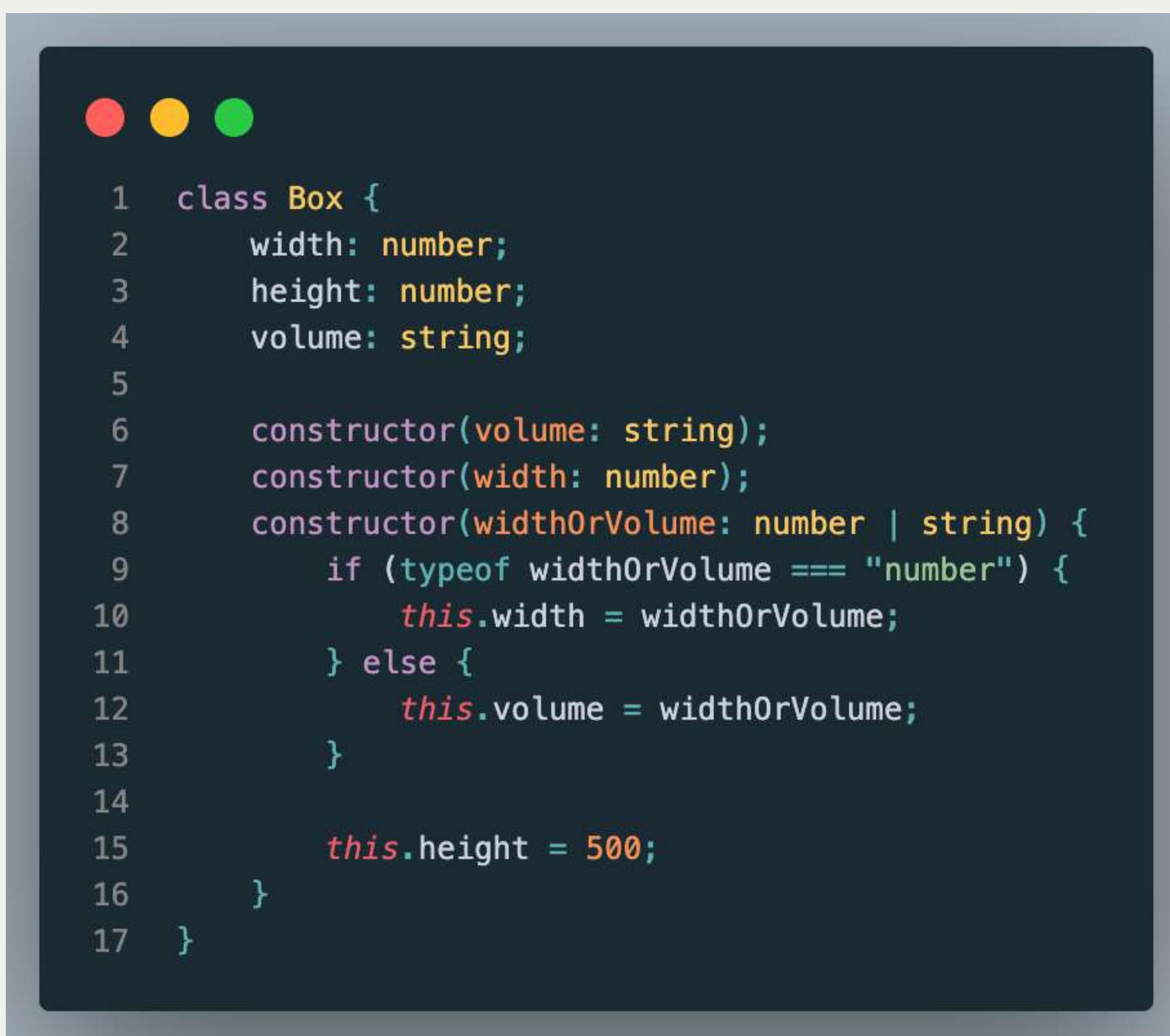
КОНСТРУКТОРЫ КЛАССОВ В ТС

Метод **constructor** необходим для создания экземпляра класса и присвоения начальных значений. Он вызывается, когда создается конструкция с `new`

Это метод, а значит к нему можно применять: **тиปизацию всех аргументов и перегрузки**. Но! У него нельзя изменить возвращаемое значение. Это приведет к **ошибке**.

Конструктор всегда возвращает **инстанс (экземпляр) класса!**

Перегрузки доступны для конструкторов в таком же формате, как и для обычных функций. Но при большом количестве аргументов лучше воздержаться от их использования:

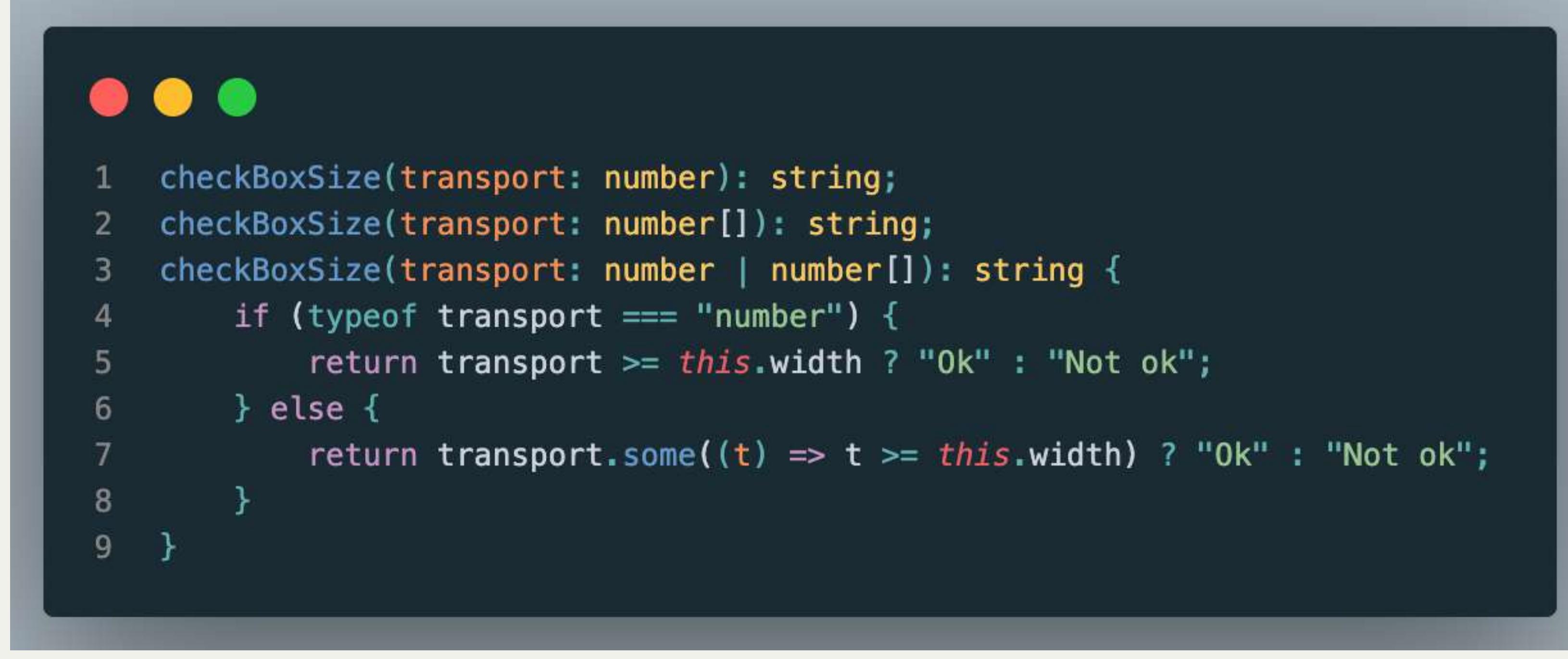


```
● ● ●
1  class Box {
2      width: number;
3      height: number;
4      volume: string;
5
6      constructor(volume: string);
7      constructor(width: number);
8      constructor(widthOrVolume: number | string) {
9          if (typeof widthOrVolume === "number") {
10              this.width = widthOrVolume;
11          } else {
12              this.volume = widthOrVolume;
13          }
14          this.height = 500;
15      }
16  }
17 }
```

Еще одно ограничение состоит в том, что конструкторы **не могут использовать обобщения**. Но сами классы и другие методы могут это делать

МЕТОДЫ КЛАССОВ В ТС

Методы работают **по тем же правилам**, что и обычные функции. К ним можно применять перегрузки, обобщения, типизацию возвращаемого значения и аргументов. Например, метод проверки размера посылки:

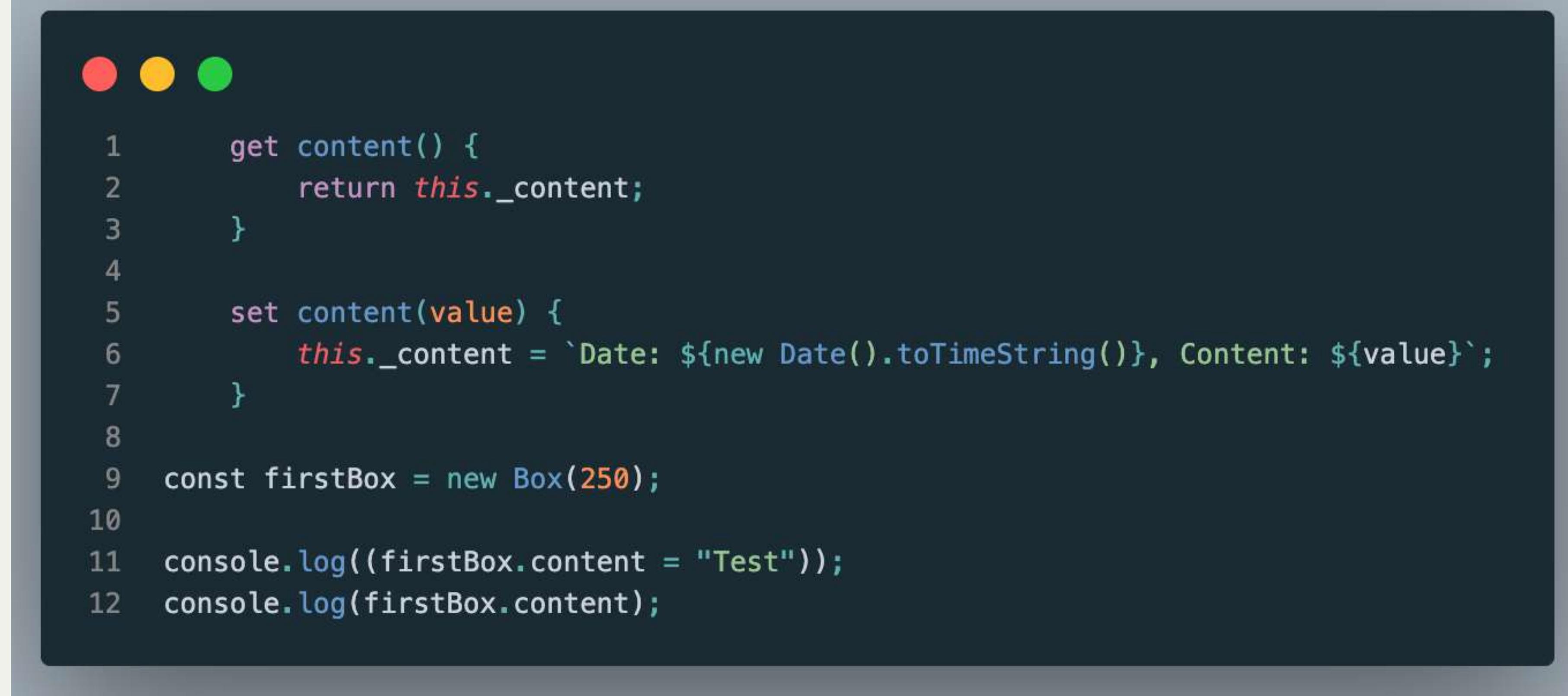


```

1  checkBoxSize(transport: number): string;
2  checkBoxSize(transport: number[]): string;
3  checkBoxSize(transport: number | number[]): string {
4      if (typeof transport === "number") {
5          return transport >= this.width ? "Ok" : "Not ok";
6      } else {
7          return transport.some((t) => t >= this.width) ? "Ok" : "Not ok";
8      }
9  }

```

Для методов класса доступно так же ключевое слово **get/set**, превращающее их в **свойства-аксессоры**. Это позволяет инкапсулировать нужные свойства, делать проверки на моменте установки значения и возвращать наружу интерфейс работы с классом:



```

1  get content() {
2      return this._content;
3  }
4
5  set content(value) {
6      this._content = `Date: ${new Date().toTimeString()}, Content: ${value}`;
7  }
8
9  const firstBox = new Box(250);
10
11 console.log(firstBox.content = "Test");
12 console.log(firstBox.content);

```

Свойство **_content** “скрыто” внутри класса. Мы изучим способы создать приватное свойство дальше по курсу. Но пока его можно спокойно получить через **firstBox._content**, так что символ “**_**” - это лишь **словесное указание** другим разработчикам.

- 👉 Если в паре с **get** не существует **set**, то это свойство автоматически становится **readonly**
- 👉 Тип аргумента **value** внутри **set** устанавливается **автоматически** на основании того типа, который возвращает **get**. Можно поменять при необходимости
- 👉 **get/set** свойства **не могут быть асинхронными**. Если нужна асинхронность - используйте обычный метод. Пример выше можно переписать, если дата приходит асинхронно (например после запроса на сервер):



```

1  async content(value: string) {
2      const date = await new Date().toTimeString();
3      this._content = `Date: ${date}, Content: ${value}`;
4      console.log(this._content);
5      return this._content;
6  }

```

НАЧАЛЬНОЕ ЗНАЧЕНИЕ И INDEX SIGNATURES

Начальное значение свойств в классах можно задавать как при помощи class fileds, так и в конструкторе. Скорее всего в своей работе вы встретите и то и то. А что использовать - будет диктовать стиль проекта или ваши предпочтения:



```
1 class Box {
2     height: number = 500;
3 }
```



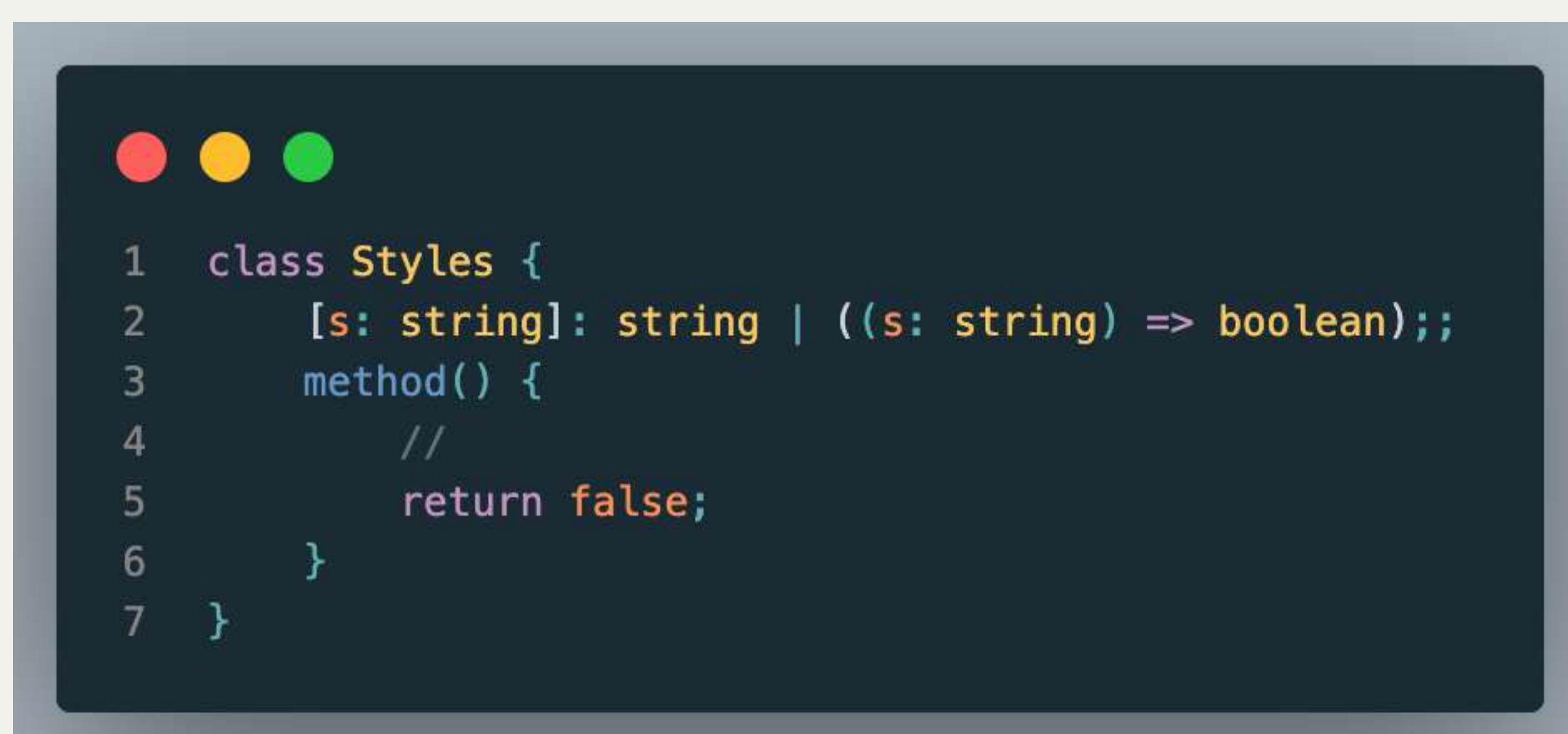
```
1 constructor() {
2     this.height = 500;
3 }
```

Если мы знаем, какого типа будут все свойства, но не знаем их количество, то можно применить Index Signatures:



```
1 class Styles {
2     [s: string]: string;
3 }
4
5 const style = new Styles();
6 style.color = 5; // error
7 style.font = "Roboto";
```

Но с классами эта возможность не так удобна. Если захотим добавить метод - придется добавлять тип:



```
1 class Styles {
2     [s: string]: string | ((s: string) => boolean);
3     method() {
4         //
5         return false;
6     }
7 }
```

Так что в реальном коде можно такую возможность и не встретить

НАСЛЕДОВАНИЕ КЛАССОВ В ТС

Наследование – это когда мы можем создать цепочку классов, где есть связь родитель – потомок. Потомок будет содержать **все свойства и методы родителя**, и на усмотрение разработчика содержать что-то дополнительное или немного по другому реализует родительский функционал. Для наследования мы используем ключевое слово **extends**:

```

1 class Box {
2   width: number;
3   height: number = 500;
4   // ...

```



```

1 class PresentBox extends Box {
2   wrap: string;
3   height: number = 600;
4   // ...

```

Потомки могут конструироваться по другому, а значит и будут иметь свой конструктор. Не забывайте использовать **суперконструктор** перед использованием this:

```

1 class PresentBox extends Box {
2   wrap: string;
3   height: number = 600;
4
5   constructor(wrap: string, width: number) {
6     super(width);
7     this.wrap = wrap;
8   }

```

В наследуемом классе свойства и методы **должны совпадать** по типам с аналогами в родителе. В методах возвращаемое значение должно быть таким же, а новые аргументы обязательно с **модификатором опциональности**:

```

1 // Parent
2 async content(value: string) {
3   const date = await new Date().toTimeString();

```



```

1 // Derived
2 async content(value: string, text?: string) {
3   const date = await new Date().toTimeString();

```

Чтобы сказать, что метод был “перезаписан” и в потомке он уже имеет другой функционал, существует модификатор override:

```

1 // Derived
2 override async content(value: string, text?: string) {
3   const date = await new Date().toTimeString();

```

Его суть в двух моментах:

- 👉 Четко сказать разработчику, что это перезаписанный метод родителя
- 👉 Если из родителя исчезает этот метод – в потомке будет **ошибка**. Так мы убедимся, что потомок не будет использовать собственный метод и всегда сможем это подправить

ИМПЛЕМЕНТАЦИЯ В КЛАССАХ

Простым языком, **имплементация** - это описание того, что должно быть внутри класса при помощи одного или нескольких интерфейсов. В свою очередь, это позволяет вам строить определенные **связи и зависимости**. Реализуется это через ключевое слово **implements**:

```

1 interface IUser {
2   login: string;
3   password: string;
4 }

```

```

1 class UserForm implements IUser {
2   login: string;
3   password: string;
4 }

```

Теперь в классе должны быть все свойства и методы, которые были в интерфейсе. Сам класс **можно** дополнять другими методами или свойствами.

При этом мы создали **связь** и если в интерфейсе пользователь что-то изменится, то TS подскажет нам, что и в классе необходимо что-то изменить.

Имплементацию можно проводить сразу от **несколько интерфейсов**:

```

1 interface Validation {
2   valid: boolean;
3   isValid: (data: string) => boolean;
4 }
5
6 class UserForm implements IUser, Validation {
7   login: string;
8   password: string;
9   valid: boolean = false;
10
11  isValid(login: string) {
12    return login.length > 3;
13  }
14 }

```

Если не указать тип у аргумента `login`, то будет ошибка. Все потому, что класс **автоматически не перенимает** аннотации аргументов или все необязательные свойства интерфейса(ов)! Вы **сами** должны добавить аннотации типов и необязательные свойства по необходимости

МОДИФИКАТОРЫ ВИДИМОСТИ

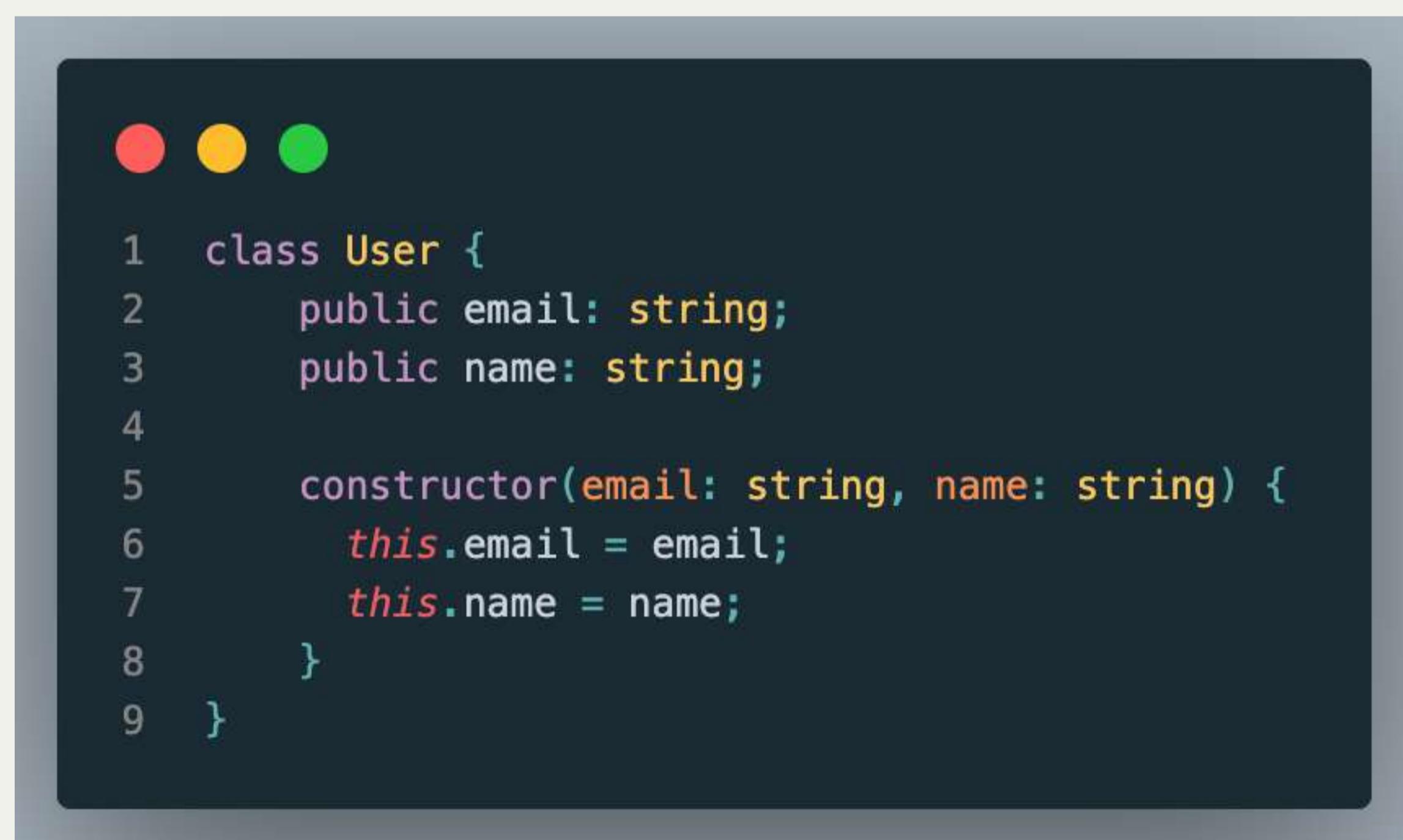
TS предоставляет модификаторы для **свойств и методов класса**. Они позволяют указать разработчикам на их принадлежность и “открытость”. Они **будут удалены** из кода после компиляции, так что вводят ограничения только в TS

По умолчанию любые свойства и методы являются **публичными**, то есть доступными снаружи экземпляра класса. Так же это можно указать модификатором **public**:



```
1 class User {
2     public email: string;
3     public name: string;
4 }
```

Обычно их не указывают, за исключением определенного стиля проекта. Или **сокращенной записи свойств**, где TS сам создаст все нужные данные и присвоит их:



```
1 class User {
2     public email: string;
3     public name: string;
4
5     constructor(email: string, name: string) {
6         this.email = email;
7         this.name = name;
8     }
9 }
```

То же самое:



```
1 class User {
2     constructor(public email: string, public name: string) {}
3 }
```

Если мы хотим запретить разработчикам работать с методом или свойством “снаружи”, то мы можем использовать модификатор **private**:

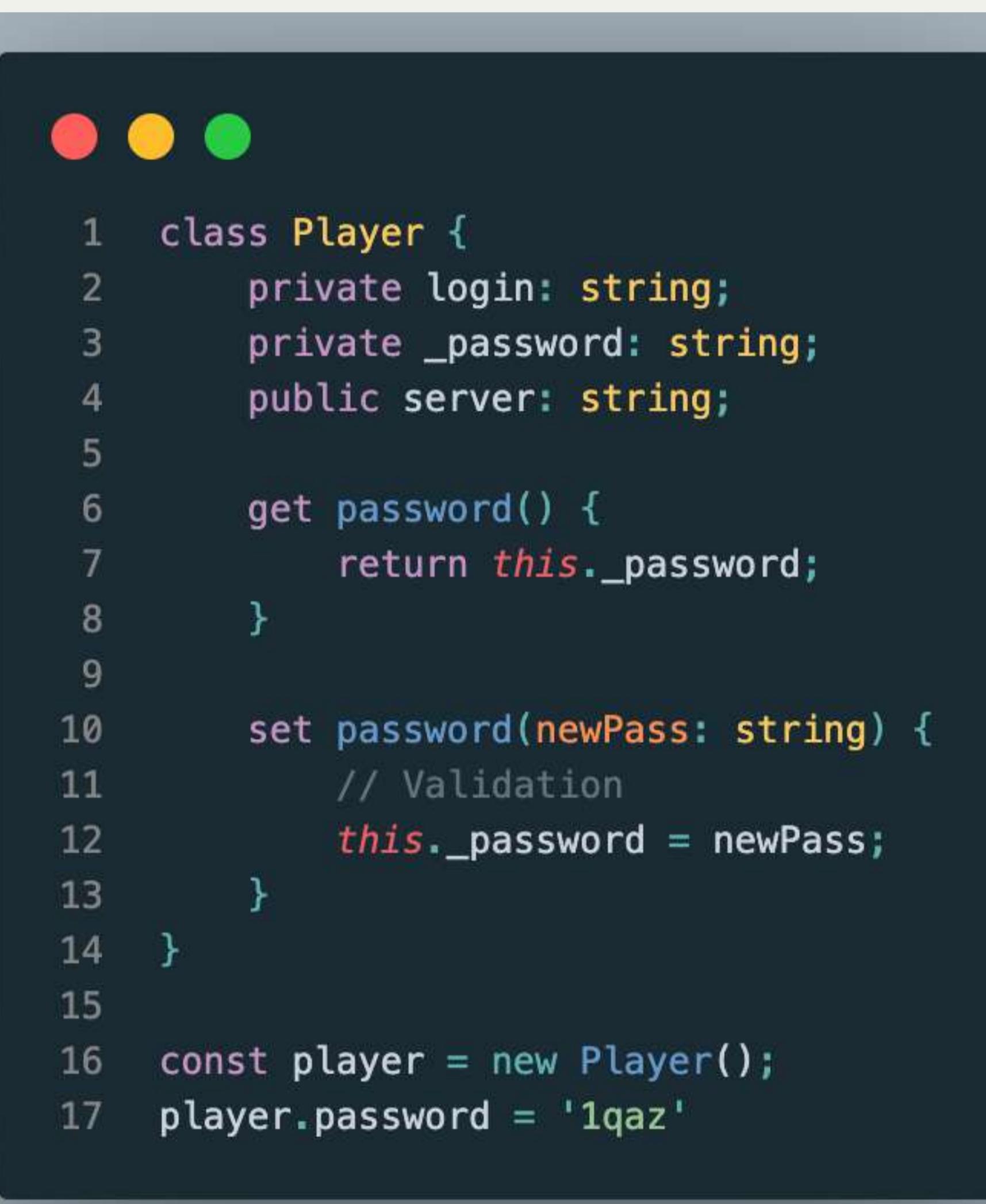


```

1 class Player {
2     private login: string;
3     private password: string;
4     public server: string;
5 }
6
7 const player = new Player();
8 player.login = '1qaz'; // Error
9 player.server = 'first' // Ok

```

Теперь его можно использовать **только внутри самого класса**. Как один из вариантов работы со скрытым свойством “снаружи” - это использование **get/set**:

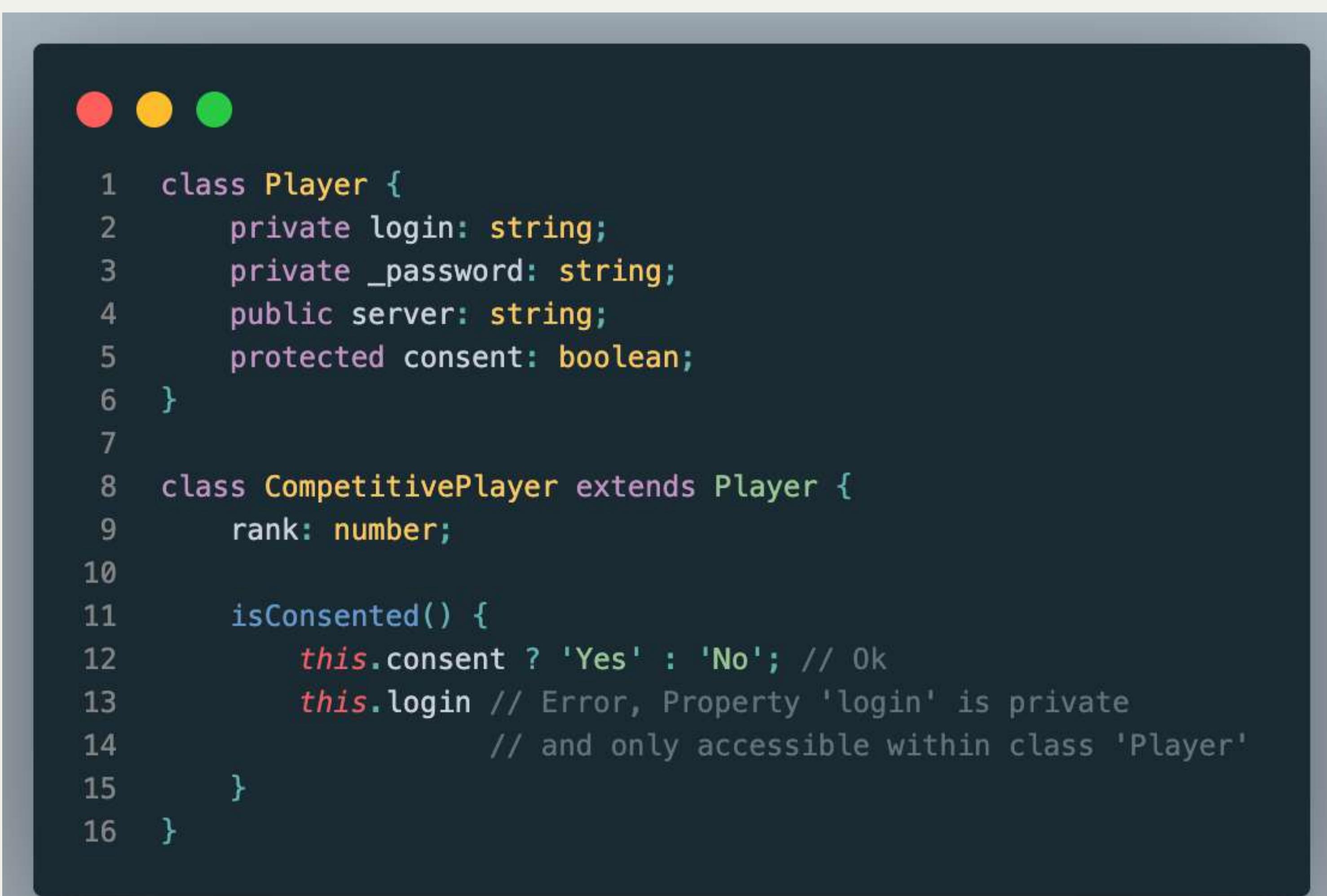


```

1 class Player {
2     private login: string;
3     private _password: string;
4     public server: string;
5
6     get password() {
7         return this._password;
8     }
9
10    set password(newPass: string) {
11        // Validation
12        this._password = newPass;
13    }
14 }
15
16 const player = new Player();
17 player.password = '1qaz'

```

При работе с **наследованием**, **private**-свойства/методы **не будут видны у потомков**. Если вы хотите сделать так, чтобы снаружи свойство не было доступно, но **появлялось у всех потомков**, необходимо использовать модификатор **protected**:



```

1 class Player {
2     private login: string;
3     private _password: string;
4     public server: string;
5     protected consent: boolean;
6 }
7
8 class CompetitivePlayer extends Player {
9     rank: number;
10
11     isConsented() {
12         this.consent ? 'Yes' : 'No'; // Ok
13         this.login // Error, Property 'login' is private
14             // and only accessible within class 'Player'
15     }
16 }

```

ПРИВАТНЫЕ ПОЛЯ

В **современном стандарте JS** уже существует возможность создавать приватные свойства и методы на уровне языка. Для этого используется символ решетки:



```
1 class Player {
2     #login: string;
3     private _password: string;
4     public server: string;
5     protected consent: boolean;
6 }
7
8 const player = new Player();
9 player.#login // Error
```

Этот функционал останется **и после компиляции**, но в разных вариантах, в зависимости от установленного стандарта. Модификаторы в TS появились намного раньше этой возможности, да и необходимы для совместимости со старыми стандартами

Что использовать: **модификатор private или #** - решение за вами, но большинство разработчиков примерно такого мнения:

- 👉 На **бэкенде** обычно хватает модификатора, так как очень мало кода, который намеренно будет лезть в другие объекты и что-то делать. Более практично контролировать работу самих разработчиков, что и делает TS
- 👉 На **фроненде** есть такие скрипты, которые могут намеренно что-то менять в объектах. Так что имеет смысл самые ценные и уязвимые данные пометить через #. Тогда функционал “недоступности” останется и в конечном коде.
Помните, что это не скрывает эти данные в прямом смысле этого слова. Они все равно видны в конечном скрипте. Это лишь **ограничивает функциональный доступ к ним**

СТАТИЧНЫЕ СВОЙСТВА И МЕТОДЫ

В **современном** стандарте JS уже существует возможность создавать статичные свойства и методы класса. Для этого используется ключевое слово **static**. Это позволяет создать у класса **сущность, общую для каждого экземпляра**. Общая настройка, общий метод, что-то, что в общем характеризует этот класс.



```

1 class Player {
2     static game: string = "COD";
3 }
4 const game = Player.game; // "COD"

```

Для имен статичных методов и свойств **не могут** использоваться встроенные, например `name`. К ним можно применять модификаторы видимости (`private`, `protected`). В таком случае, использовать их можно только внутри самих классов в таком формате:



```

1 class Player {
2     private static game: string = "COD";
3
4     static getGameName() { // Make it static too
5         return Player.game;
6     }
7 }
8
9 const game = Player.getGameName();

```

Классическим примером статичных методов и свойств является класс `Math`: `Math.random()`, `Math.ceil()`, `Math.PI` и тд.

Но в TS **не существует** такого понятия, как статичные классы. Даже если класс состоит только из `static` сущностей, то мы не можем создать конструкцию `static class Test {}`

Это не имело бы особого смысла, так как в JS больше вариативности. Вместо `static` свойства вы можете использовать переменную, а вместо метода - функцию, например. Так что пользуйтесь `static` с умом, когда действительно сущность привязана к классу.

Для их инициализации статичных свойств существуют **статичные блоки**. Они позволяют установить значение только один раз, при первой инициализации объекта



```

1 function setName() {
2     return 'COD';
3 }
4
5 class Player {
6     private static game: string;
7
8     static {
9         Player.game = setName();
10    }
11 }

```

КАК НЕ ПОТЕРЯТЬ КОНТЕКСТ

При усложнении функционала всегда можно потерять контекст в классах. В TS существует возможность четко сказать, чем должен быть контекст и не получать такие ситуации:

```
1 class Player {
2     #login: string;
3
4     constructor(login: string) {
5         this.#login = login;
6     }
7
8     logIn() {
9         return `Player ${this.#login} online!`;
10    }
11 }
12
13 const player = new Player("Test");
14 const test = player.logIn;
15 test(); // Error: Cannot read properties of undefined (reading '#login')
```

Проблему можно решить при помощи **bind** или **стрелочной функции**. Но до запуска кода вы не узнаете об ошибке. Поэтому в TS есть вариант сразу сказать, **чем будет контекст** прямо первым аргументом в функции. И вы увидите ошибку на этапе разработки:

```
1 class Player {
2     #login: string;
3
4     constructor(login: string) {
5         this.#login = login;
6     }
7
8     logIn(this: Player) { // this type
9         return `Player ${this.#login} online!`;
10    }
11 }
12
13 const player = new Player("Test");
14 const test = player.logIn;
15 test(); // Error: The 'this' context of type 'void' is not assignable to method's 'this' of type 'Player'
```

Иногда мы из метода возвращаем контекст, то есть ссылку на экземпляр объекта. В таких случаях **не стоит жестко типизировать возвращаемое значение**, так как оно может сломать логику:

```
1 connect(): Player {
2     // Do stmh
3     return this;
4 }
```

Неправильно: даже в наследуемых классах будет возвращать Player, что может привести к ошибке

```
1 connect() {
2     // Do stmh
3     return this;
4 }
```

Правильно: всегда будет возвращаться **текущий** экземпляр класса, даже в потомках

При помощи контекста мы можем проверить, к какому классу относится экземпляр и написать свой **защитник типа**:

```
1 isPro(): this is CompetitivePlayer {
2     return this instanceof CompetitivePlayer;
3 }
4
5 const somePlayer: Player | CompetitivePlayer = new CompetitivePlayer("Test");
6 somePlayer.isPro() ? console.log(somePlayer) : console.log(somePlayer);
```

АБСТРАКТНЫЕ КЛАССЫ

В TS существует возможность создать абстрактные классы - **концепт, некий шаблон чего-то**. Таким образом в нем могут находиться и шаблонные методы-типы, которые должны присутствовать и мы их должны будем еще создать и готовые, которые сразу можно пустить в работу:

```
1 abstract class AbstractVehicle {
2     model: string;
3     capacity: number;
4     abstract startEngine: (time: Date) => string;
5     stopEngine(time: Date): string {
6         this.startEngine(new Date());
7         return 'Engine Stopped'
8     }
9 }
10
11 class Vehicle extends AbstractVehicle {
12     startEngine = (time: Date) => {
13         return 'Started'
14     };
15 }
```

Метод, указанный через `abstract` должен быть реализован у потомка в соответствии с заданным типом. А метод `stopEngine` уже готов к использованию, как и два свойства. Готовые методы **невозможно** было бы передать при имплементации интерфейса, ведь в нем содержатся только типы:

```
1 interface IEngine {
2     model: string;
3     capacity: number;
4     startEngine: (time: Date) => string;
5 }
6
7 class Vehicle implements IEngine {
8     model: string;
9     capacity: number;
10    startEngine = (time: Date) => {
11        return 'Started'
12    };
13 }
```

Так что абстрактные классы могут дать немного больше свободы. У них есть два **ограничения**:

- 👉 У них **нельзя создать экземпляры**, только отнаследоваться от них
- 👉 Нельзя создавать **отдельно абстрактные методы** без объявления всего класса абстрактным. Получится так, что экземпляр класса создать можно, не известно чем будет абстрактный метод. Так что это запрещено

После компиляции сами классы-абстракции остаются, но без абстрактных методов. Они появляются только у потомков