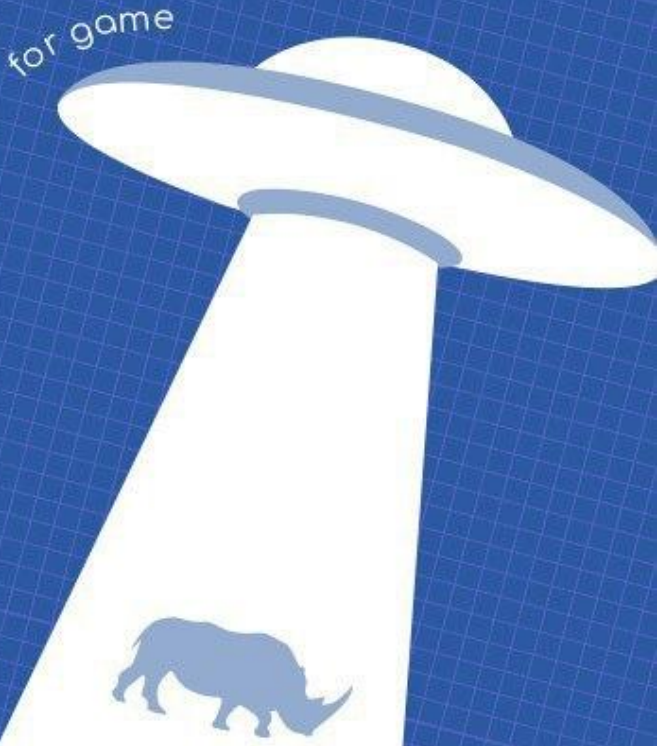
TYPESCRIPT DEFINITIVE GUIDE 3.2



TypeScript
Подробное Руководство

Книга и документация в одном

Дата последнего обновления: 9/8/2020

Глава 10

Совместимость типов на основе вариантности

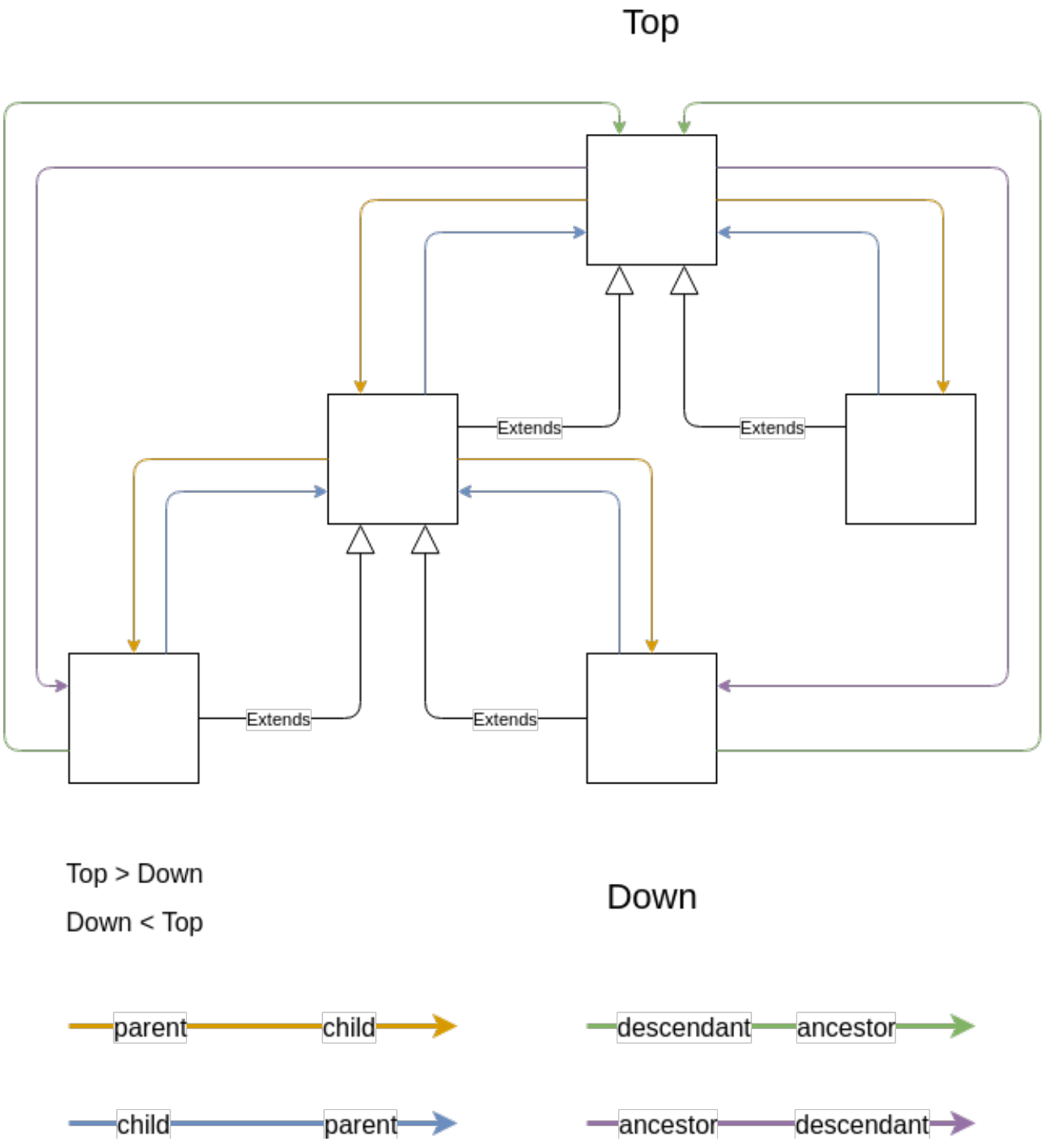
[10.0] Совместимость типов на основе вариантности

Помимо того, что совместимость типов зависит от вида типизации, которая была подробно разобрана в главе “Экскурс в типизацию - Совместимость типов на основе вида типизации”, она также может зависеть от такого механизма, как вариантность. *Вариантность* — это механизм переноса иерархии наследования типов на производные от них типы. В данном случае производные не означает *связанные отношением наследования*. Производные, скорее, означает *определяемые теми типами, с которых переносится наследование*.
Если вы впервые сталкиваетесь с этим понятием и определение вариантности кажется бессмысленным набором слов, то не стоит расстраиваться, эта тема очень простая, в чем вы сами скоро убедитесь.
В основе системы типов могут быть заложены следующие виды вариантности — *ковариантность*, *контравариантность*, *инвариантность*, *бивариантность*. Кроме того, что система типов использует механизм вариантности для своих служебных целей, она также может предоставлять разработчикам возможность управлять им зависящими от конкретного языка способами.
Но прежде чем познакомиться с каждым из этих видов вариантности отдельно, стоит сделать некоторые уточнения касательно иерархии наследования.

[10.1] Иерархия наследования

Иерархия наследования — это дерево, вверху которого расположен *корень*, самый *базовый тип* (*менее конкретный тип*), ниже которого располагаются его *подтипы* (*более*

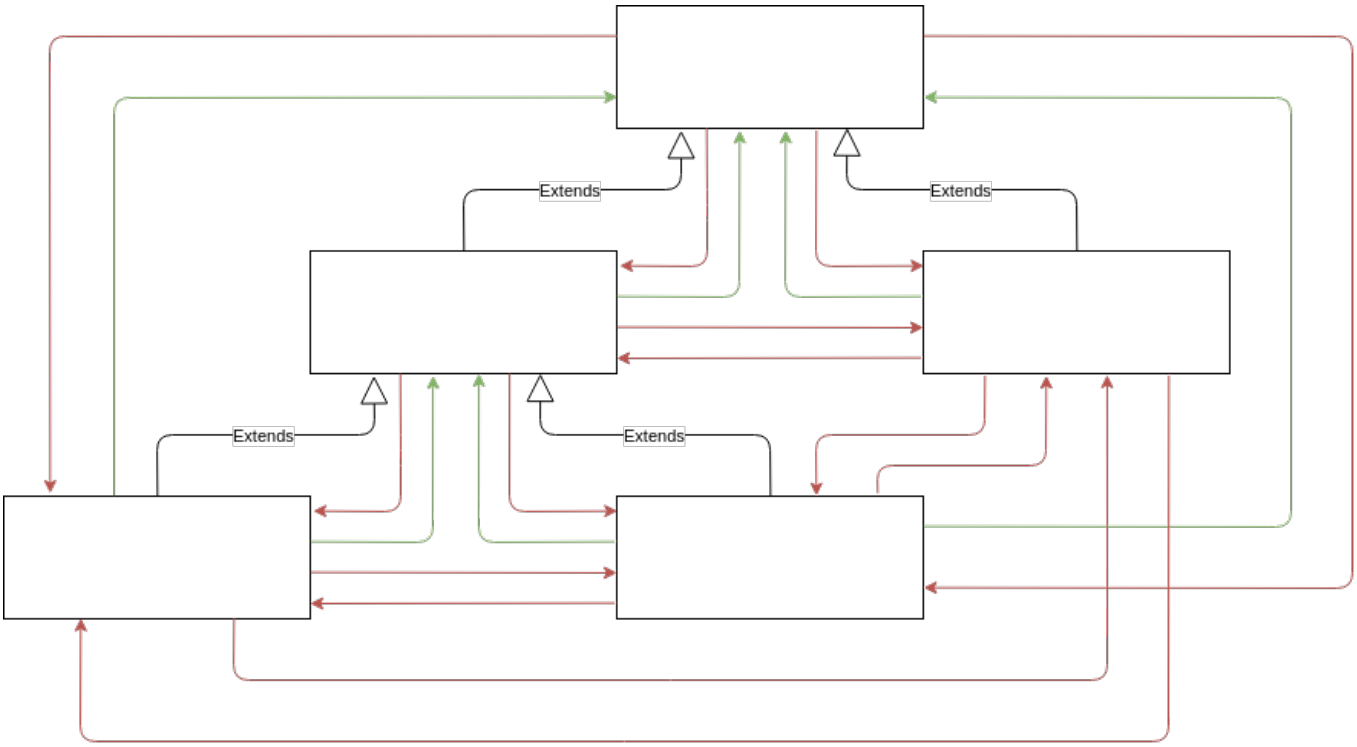
конкретные типы). В случаях преобразования подтипа к базовому типу говорят, что выполняется *восходящее преобразование* (*upcasting*). И наоборот, когда выполняется приведение базового типа к его подтипу, говорят, что выполняется *нисходящее приведение* (*downcasting*). Отношения между супертипом и его подтипом описываются как отношение *родитель-ребенок* (parent-child). Отношения между родителем типа и его ребенком описываются как *предок-потомок* (ancestor-descendant). Кроме того, при логическом сравнении тип, находящийся выше по дереву, больше (`>`) чем тип находящийся ниже по дереву (и наоборот). Можно сказать, что `parent > child`, `child < parent`, `ancestor > descendant`, `descendant < ancestor`. Все это представлено на диаграмме ниже.



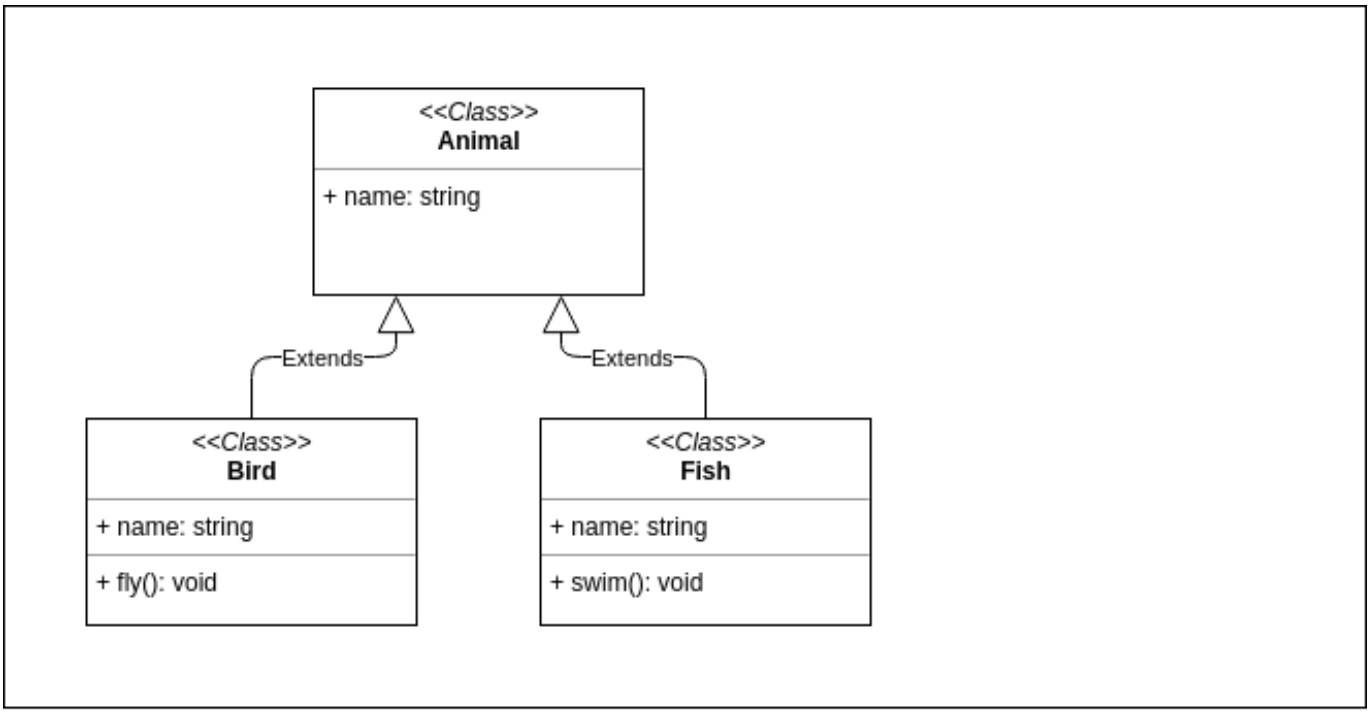
Этого вполне достаточно для того, чтобы приступить к разбору видов вариантности.

[10.2] Ковариантность

Ковариантность — это механизм, позволяющий использовать более конкретный тип там, где изначально предполагалось использовать менее конкретный тип. Простыми словами, совместимыми считаются типы, имеющие отношение `A > B` и `A = B`.



Ковариантность не рекомендуется в местах, допускающих запись. Чтобы понять смысл этих слов, ниже представлена диаграмма, которая иллюстрирует, как через базовый тип можно добавить в массив с подтипом другой, несовместимый подтип и тем самым нарушить типобезопасность программы.



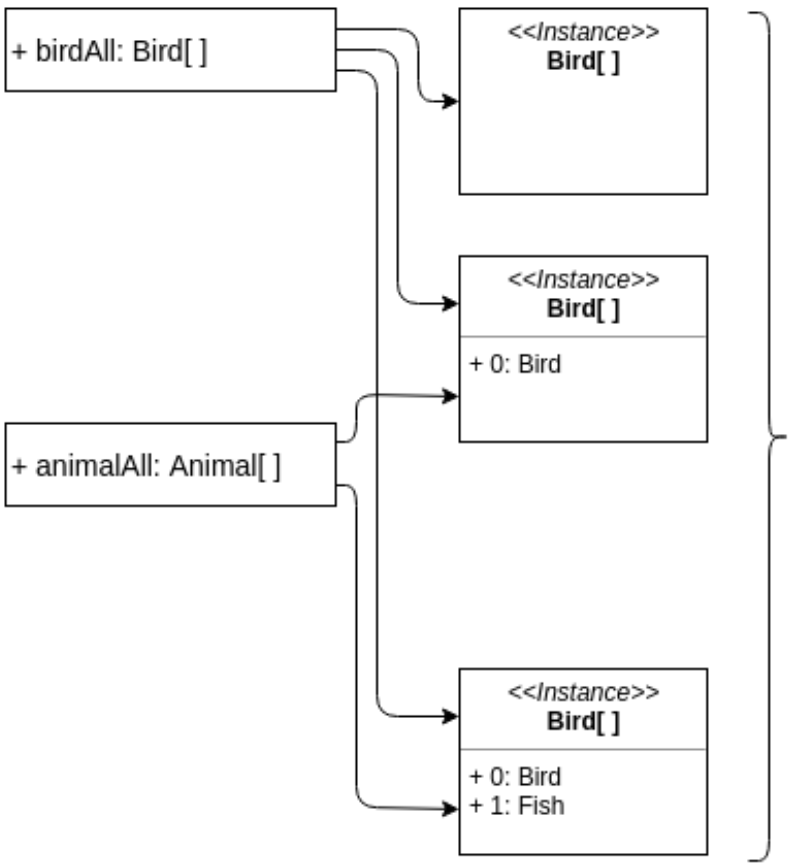
0 Создаем массив с типом Bird[ ] и присваиваем его ссылке с типом Bird[ ]

1 добавляем в массив экземпляр класса Bird

2 Присваиваем массив с типом Bird[ ] ссылке имеющей ,базовый тип Animal[ ]

3 Добавляем в массив с базовым типом Animal [ ] экземпляр класса Fish

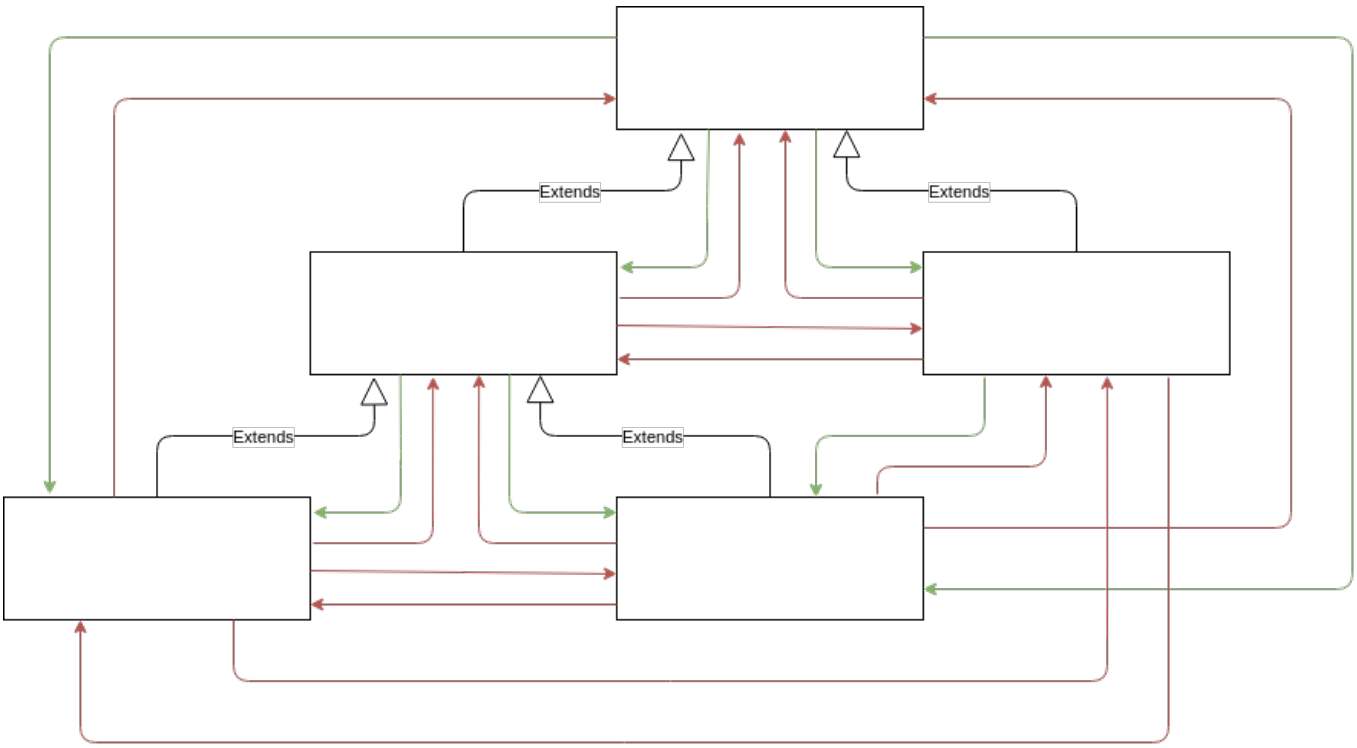
Теперь в массиве с типом Bird[ ] присутствует элемент принадлежащий к типу Fish



Ковариантность рекомендуется применять в местах, допускающих чтение.

[10.3] Контравариантность

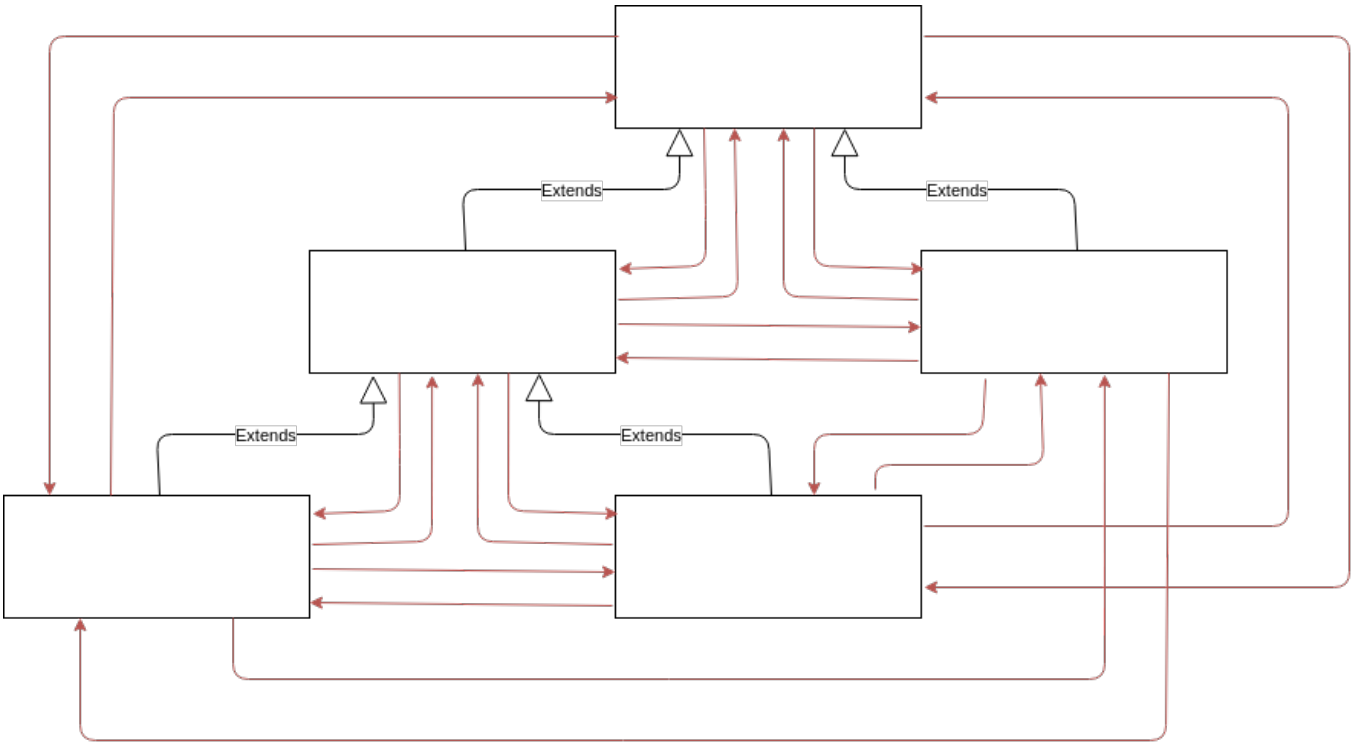
Контравариантность — это противоположный ковариантности механизм, позволяющий использовать менее конкретный тип там, где изначально предполагалось использовать более конкретный тип. Другими словами, совместимыми считаются типы имеющие отношения `A < B` и `A = B`.



Контравариантность не рекомендуется в местах, допускающих чтение, и наоборот, рекомендуется применять в местах, допускающих запись.

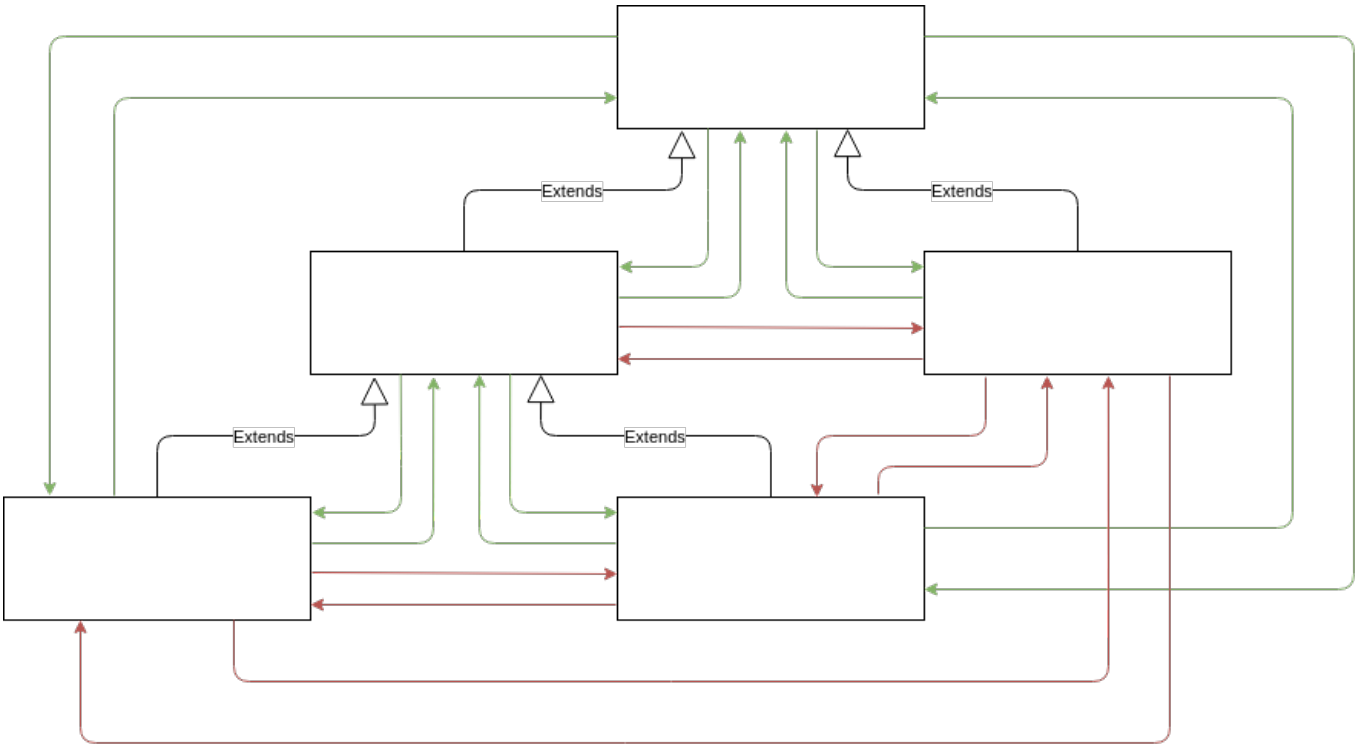
[10.4] Инвариантность

Инвариантность — это механизм, позволяющий использовать только заданный тип. Совместимыми считаются только идентичные типы `A = A`.



[10.5] Бивариантность

Бивариантность — это механизм, который является представлением всех, перечисленных ранее, видов вариантности. В его случае совместимыми считаются любые из перечисленных ранее варианты типы `A > B`, `A < B` и `A = B`.



Бивариантность является самым нетипобезопасным видом вариантности.

Глава 12

Базовый Тип Any

[12.0] Базовый Тип Any

То что *TypeScript* является типизированной надстройкой над *JavaScript*, от которой после компиляции не остаётся и следа, означает, что первый перенял от второго всю его идеологию. Одним из таких моментов является разделение типов данных на типы значения (примитивные) и ссылочные типы.
В *JavaScript* все не объектные типы по своей природе иммутабельные. Иммутабельность означает, что значения нельзя модифицировать, а можно лишь полностью перезаписать новым значением. Из-за этого типы значения были прозваны примитивными типами.

[12.1] Any (any) произвольный тип

Все типы в *TypeScript* являются подтипами `any`, который указывается с помощью ключевого слова `any`. Это означает, что он совместим в обе стороны с любым типом данных и с точки зрения системы типов является *высшим типом* (top type).

typescript

```
let apple: any = 0;
apple = "";
apple = true;
```

Тип `any` в аннотации рекомендуется применять только в самых крайних случаях. К примеру объявление неинициализированной переменной, с заранее неизвестным типом, что является допустимым сценарием на ранних стадиях применения техники `TDD` при создании программы. В остальных случаях настоятельно рекомендуется прежде всего рассматривать кандидатуру обобщенных типов, которые будут рассматриваться позднее.
Тип `any` позволяет работать со значением динамически, что не вызывает ошибок при обращении к членам не описанных в типе данных (незадекларированных), о которых анализатор *TypeScript* ничего не знает.
Примером этого может служить сервис, который работает с сервером посредством *API*. Полученные и сериализованные данные могут храниться как тип `any` прежде чем они будут и преобразованы к конкретному типу.

typescript

```
let data: any = JSON.parse('{"id": "abc"}');
let id = data.id; // ок
```

Если при объявлении переменных `var` и `let` и полей, объявленных в теле класса, не было присвоено значение, компилятором будет выведен тип данных `any` .

typescript

```
var apple; // apple: any
let lemon; // lemon: any

class Fruit {
    name; // name: any
}
```

То же правило касается и параметров функций.

typescript

```
function weight(fruit) { // fruit: any

}
```

Кроме того, если функция возвращает значение, принадлежащее к типу, который компилятор не в состоянии вывести, то возвращаемый этой функцией тип данных будет также будет выведен как тип `any`.

typescript

```
function sum(a, b) { // function sum(a: any, b: any): any
    return a + b;
}
```

Тип `any` является уникальным для *TypeScript*, в *JavaScript* подобного типа не существует.

[12.2] Итог

- Тип данных `any` является супертипом для всех типов в *TypeScript* и указывается с помощью ключевого слова `any`.
- Тип данных `any` указывается тогда, когда тип данных заранее неизвестен.
- Тип данных `any` указывается только в самых крайних случаях.
- Тип данных `any` позволяет обращаться к незадекларированным членам объектов, имитируя динамическое поведение, тем самым перекладывая ответственность за проверку реализации интерфейса на разработчика.

Глава 32

Обобщения (Generics)

Из всего что стало и ещё станет известным о типизированном мире, тем кто только начинает свое знакомство с ним, тема посвященная обобщениям (*generics*) может представлятся наиболее сложной. Хотя данная тема, как и все остальные обладает некоторыми нюансами, каждый из которых будет детально разобран, в реальности рассматриваемые в ней механизмы очень просты и схватываются на лету. Поэтому приготовтесь что к концу главы место занимаемое множеством вопросов касающихся обобщений займет желание сделать все пользовательские конструкции универсальными.

[32.0] Обобщения - общие понятия

Представьте огромный, дорогущий, высокотехнологичный типографский печатный станок выполенный в виде монолита что в свою очередь делает его пригодным для печати только одного номера газеты. То есть для печати сегодняшних новостей необходим один печатный станок, для завтрашних другой и т.д. Подобный станок сравним с *обычным типом* признаки которого после объявления остаются неизменны при его реализации. Другими словами если при существовании типа `A` описание которого включает поле принадлежащее к типу `number` потребуется тип отличие которого будет заключатся лишь в принадлежности поля, то возникнет необходимость в его объявлении.

typescript

```
// простые типы сравнимы с монолитами

// этот станок предназначен для печати газет под номером A
interface A {
    field: number;
}

// этот станок предназначен для печати газет под номером B
interface B {
    field: string;
```

```
}

// и т.д.
```

К счастью в нашей реальности нашли решение не только относительно печатных станков, но и типов. Таким образом появилась парадигма *обобщенного программирования*.
Обобщенное программирование (*Generic Programming*) — это подход, при котором алгоритмы могут одинаково работать с данными, принадлежащими к разным типам данных, без изменения декларации.
В основе обобщенного программирования лежит такое ключевое понятие как *обобщение* . *Обобщение* (*Generics*) - это *параметризированный тип* позволяющий объявлять *параметры типа* являющиеся временной заменой *конкретных типов* которые будут *конкретизированны* в момент создания экземпляра. Параметры типа можно использовать в большинстве операций допускающих работу с обычными типами. Все это вместе дает повод сравнивать обобщенный тип с *правильной версией* печатного станка, чьи заменяемые валы предназначенные для отпечатывания информации на проходящей через них бумаге сопоставимы с *параметрами типа*.
В реальности обобщения позволяют сокращать количество преобразований (приведений) и писать многократно используемый код, при этом повышая его типобезопасность.
Этих примеров должно быть достаточно чтобы сложить отчетливый образ того чем на самом деле являются обобщения, но перед тем как продолжить стоит уточнить значения таких терминов как *обобщенный тип*, *параметризированный тип* и *универсальная конструкция* которые далеко не всем очевидны.
Для понимания этих терминов необходимо представить чертеж бумажного домика в который планируется поселить пойманного на пикнике жука. Когда гипотетический жук мысленно располагается вне границ начерченного жилища сопоставимого с типом, то оно предстает в виде *обобщенного типа*. Когда жук представляется внутри своей будущей обители, то о ней говорят как о *параметризированном типе*. Если же чертеж материализовался, хотя и в форму представленную обычной коробкой из под печенья, то её называют *универсальной конструкцией*.
Другими словами, тип определяющий параметр обозначается как обобщенный тип. при обсуждении типов представляемых параметрами типа необходимо уточнять что они определены в параметрезированном типе. Когда объявление обобщенного типа получило реализацию, то такую конструкцию, будь то класс или функция, называют универсальной (универсальный класс, универсальная функция или метод).

[32.1] Обобщения в TypeScript

В *TypeScript* обобщения могут быть указаны для типов, определяемых с помощью:

- *псевдонимов* (`type`)

- *интерфейсов*, объявленных с помощью ключевого слова `interface`
- *классов* (`class`), в том числе *классовых выражений* (*class expression*)
- *функций* (`function`) определенных в виде как деклараций (*Function Declaration*), так и выражений (*Function Expression*)
- *методов* (*method*)

Обобщения объявляются при помощи пары угловых скобок, в которые через запятую, заключены *параметры типа* называемые также *типо-заполнителями* или *универсальными параметрами* `Type<T0, T1>` .

typescript

```typescript
        /**[0][1] [2] */
interface Type<T0, T1> {}

/**
 * [0] объявление обобщенного типа Type
 * определяющего два параметра типа [1][2]
 */
```

Параметры типа могут быть указаны в качестве типа везде, где требуется аннотация типа, за исключением членов класса (static members). Область видимости параметров типа ограничена областью обобщенного типа. Все вхождения параметров типа будут заменены на конкретные типы, переданные в качестве аргументов типа. Аргументы типа указываются в угловых скобках, в которых через запятую указываются конкретные типы данных `Type<number, string>` .

typescript

```typescript
        /**[0]  [1]      [2] */
let value: Type<number, string>

/**
 * [0] указание обобщенного типа
 * которому в качестве аргументов
 * указываются конкретные типы
 * number [1] и string [2]
 */
```

Идентификаторы параметров типа должны начинаться с заглавной буквы и кроме фантазии разработчика они также ограничены общими для *TypeScript* правилами. Если логическую принадлежность параметра типа возможно установить без какого-либо труда, как например в случае `Array<T>` кричащего что параметр типа `T` представляет тип к которым могут принадлежать элементы этого массива, то идентификаторы параметров типа принято выбирать из последовательности `T`, `S`, `U`, `V` и т.д. Также частая последовательность `T`, `U`, `V`, `S` и т.д.
С помощью `K` и `V` принято обозначать типы соответствующие `Key` / `Value`, а при помощи `P` — `Property`. Идентификатором `Z` принято обозначать полиморфный тип `this`.
Кроме того, не исключены случаи, в которых предпочтительнее выглядят полные имена, как например `RequestService`, `ResponseService`, к которым ещё можно применить *Венгерскую нотацию* - `TRequestService`, `TResponseService`.
К примеру, увидев в автодополнении редактора тип `Array<T>`, в голову сразу же приходит верный вариант, что массив будет содержать элементы принадлежащие к указанному типу `T`. Но, увидев `Animal<T, S>`, можно никогда не догадаться, что это типы данных, которые будут указаны для полей `id` и `arial`. В этом случае было бы гораздо предпочтительней дать говорящие имена `Animal<AnimalID, AnimalArial>` или даже `Animal<TAnimalID, TAnimalArial>` что позволит внутри тела параметрезированного типа `Animal` отличать его парамтры типа от конкретных объявлений.
Указывается обобщение сразу после идентификатора типа. Это правило остается неизменным даже в тех случаях, когда идентификатор отсутствует (как в случае с безымянным классовым или функциональным выражением), или же не предусмотрен вовсе (стрелочная функция).

typescript

```typescript
type Identifier<T> = {};

interface Identifier<T> {}

class Identifier<T> {
    public identifier<T>(): void {}
}

let identifier = class <T> {};

function identifier<T>(): void {}

let identifier = function <T>(): void {};

let identifier = <T>() => {};
```

Но, прежде чем приступить к детальному рассмотрению, нужно уточнить, что правила для функций идентичны, как для функциональных выражений, так и для методов.

Правила для классов ничем не отличаются от правил для классовых выражений. Исходя из этого, все дальнейшие примеры будут приводиться исключительно на классах и функциях.
В случае, когда обобщение указанно псевдониму типа (`type`), область видимости параметров типа ограничена самим выражением.

```typescript
type T1<T> = { f1: T };
```

Область видимости параметров типа при объявлении функции и функционального выражения, включая стрелочное, а также методов, ограничивается их сигнатурой и телом. Другими словами, параметр типа можно указывать в качестве типа при объявлении параметров, возвращаемого значения, а также в своем теле при объявлениях любых конструкций, требующих аннотацию типа.

```typescript
function f1<T>(p1: T): T {
    let v1: T;

    return v1;
}
```

При объявлении классов (в том числе и классовых выражений) и интерфейсов, область видимости параметров типа ограничиваются областью объявления и телом.

```typescript
```

```
interface IT1<T> {
    f1: T;
}

class T1<T> {
    public f1: T;
}
```

В случаях, когда класс/интерфейс расширяет другой класс/интерфейс, который объявлен как обобщенный, потомок обязан указать типы для своего предка. Потомок в качестве аргумента типа своему предку может указать не только конкретный тип, но тип представляемый собственными параметрами типа.

typescript

```
interface IT1<T> {}

interface IT3<T> extends IT1<T> {}
interface IT2 extends IT1<string> {}

class T1<T> {}

class T2<T> extends T1<T> implements IT1<T> {}
class T3 extends T1<string> implements IT1<string> {}
```

Если класс/интерфейс объявлен как обобщенный, а внутри него объявлен обобщенный метод, имеющий идентичный параметр типа, то последний в свой области видимости будет перекрывать первый (более конкретно это поведение будет рассмотрено позднее).

typescript

```
interface IT1<T> {
    m2<T>(p1: T): T;
}

class T1<T> {
    public m1<T>(p1: T): T {
        let v1: T;
```

```
        return p1;
    }
}
```

Принадлежность параметра типа к конкретному типу данных устанавливается в момент передачи аргументов типа. При этом конкретные типы данных указываются в паре угловых скобок, а количество конкретных типов должно соответствовать количеству обязательных параметров типа.

typescript

```
class Animal<T> {
    constructor(readonly id: T) {}
}

var bird: Animal<string> = new Animal('bird'); // Ok
var bird: Animal<string> = new Animal(1); // Error
var fish: Animal<number> = new Animal(1); // Ok
```

Если обобщенный тип указывается в качестве типа данных, то он обязан содержать аннотацию обобщения (исключением является параметры типа по умолчанию, которые рассматриваются далее в главе).

typescript

```
class Animal<T> {
    constructor(readonly id: T) {}
}

var bird: Animal = new Animal<string>('bird'); // Error
var bird: Animal<string> = new Animal<string>('bird'); // Ok
```

Когда все обязательные параметры типа используются в параметрах конструктора, при создании экземпляра класса аннотацию обобщения можно опускать. В таком случае вывод типов определит принадлежность к типам по устанавливаемым значениям. Если

же параметры являются необязательными, и значение не будет передано, то вывод типов определит принадлежность параметров типа к типу данных `unknown` .

typescript

```
class Animal<T> {
    constructor(readonly id?: T) {}
}

let bird: Animal<string> = new Animal('bird'); // Ok -> bird:
Animal<string>
let fish = new Animal('fish'); // Ok -> fish: Animal<string>
let insect = new Animal(); // Ok -> insect: Animal<unknown>
```

Относительно обобщенных типов существуют такие понятия, как *открытый* (open) и *закрытый* (closed) тип. Обобщенный тип данных в момент определения называется *открытым*. Кроме того, типы, которые указаны в аннотации и у которых хотя бы один из аргументов типа является параметром типа, также являются открытыми типами. И наоборот, если все аргументы типа принадлежат к конкретным типам, то такой обобщенный тип является *закрытым* типом.

typescript

```
class T0<T, U> {} //  T0 - открытый тип

class T1<T> {
  public f: T0<number, T>; // T0 - открытый тип
  public f1: T0<number, string>; // T0 - закрытый тип
}
```

Те же самые правила применимы и к функциям, но за одним исключением — вывод типов для примитивных типов определяет принадлежность параметров типа к литеральным типам данных.

typescript

```
function action<T>(value?: T): T | undefined {
    return value;
}

action<number>(0); // function action<number>(value?: number |
undefined): number | undefined
action(0); // function action<0>(value?: 0 | undefined): 0 | undefined

action<string>('0'); // function action<string>(value?: string |
undefined): string | undefined
action('0'); // function action<"0">(value?: "0" | undefined): "0" |
undefined

action(); // function action<unknown>(value?: unknown): unknown
```

Тогда, когда параметры типа не участвуют в операциях при создании экземпляра класса и при этом аннотация обобщения не была указана явно, то вывод типа теряет возможность установить принадлежность к типу по значениям и поэтому устанавливает его принадлежность к типу `unknown` .

typescript

```
class Animal<T> {
    public name: T;

    constructor(readonly id: string) {}
}

let bird: Animal<string> = new Animal('bird#1');
bird.name = 'bird';
// Ok -> bird: Animal<string>
// Ok -> (property) Animal<string>.name: string

let fish = new Animal<string>('fish#1');
fish.name = 'fish';
// Ok -> fish: Animal<string>
// Ok -> (property) Animal<string>.name: string

let insect = new Animal('insect#1');
insect.name = 'insect';
// Ok -> insect: Animal<unknown>
// Ok -> (property) Animal<unknown>.name: unknown
```

И опять, эти же правила верны и для функций.

```typescript
function action<T>(value?: T): T | undefined {
    return value;
}

action<string>('0'); // function action<string>(value?: string |
undefined): string | undefined
action('0'); // function action<"0">(value?: "0" | undefined): "0" |
undefined
action(); // function action<unknown>(value?: unknown): unknown
```

В случаях, когда обобщенный класс содержит обобщенный метод, параметры типа метода будут затенять параметры типа класса.

```typescript
type ReturnParam<T, U> = { a: T, b: U };

class GenericClass<T, U> {
    public defaultMethod<T> (a: T, b?: U): ReturnParam<T, U> {
        return { a, b };
    }

    public genericMethod<T> (a: T, b?: U): ReturnParam<T, U> {
        return { a, b };
    }
}

let generic: GenericClass<string, number> = new GenericClass();
generic.defaultMethod('0', 0);
generic.genericMethod<boolean>(true, 0);
generic.genericMethod('0');

// Ok -> generic: GenericClass<string, number>
// Ok -> (method) defaultMethod<string>(a: string, b?: number):
ReturnParam<string, number>
// Ok -> (method) genericMethod<boolean>(a: boolean, b?: number):
ReturnParam<boolean, number>
// Ok -> (method) genericMethod<string>(a: string, b?: number):
ReturnParam<string, number>
```

Стоит заметить, что в *TypeScript* нельзя создавать экземпляры типов, определенных параметрами типа.

```typescript
interface CustomConstructor<T> {
    new(): T;
}

class T1<T extends CustomConstructor<T>>{
    public getInstance(): T {
        return new T(); // Error
    }
}
```

Кроме того, два типа, определяемые классом или функцией, считаются идентичными, вне зависимости от того, являются они обобщенными или нет.

```typescript
type T1 = {}
type T1<T> = {} // Error -> Duplicate identifier

class T2<T> {}
class T2 {} // Error -> Duplicate identifier

class T3 {
    public m1<T>(): void {}
    public m1(): void {} // Error -> Duplicate method
}

function f1<T>(): void {}
function f1(): void {} // Error -> Duplicate function
```

[32.2] Параметры типа - extends (generic constraints)

Помимо того, что параметры типа можно указывать в качестве конкретного типа данных, они также могут расширять другие типы данных, в том числе и другие параметры типа. Такой механизм требуется, когда значения внутри обобщенного типа должны обладать ограниченным набором признаков.
Ключевое слово `extends` размещается левее расширяемого типа и правее идентификатора параметра типа `<T extends Type>`. В качестве расширяемого типа может быть указан как конкретный тип данных, так и другой параметр типа. При чем в случае, когда один параметр типа расширяет другой, нет разницы в каком порядке они объявляются. Если параметр типа ограничен другим параметром типа, то такое ограничение называют *неприкрытым ограничением типа* (*naked type constraint*),

typescript

```
class T1 <T extends number> {}
class T2 <T extends number, U extends T> {} // неприкрытое ограничение
типа
class T3 <U extends T, T extends number> {}
```

Механизм расширения требуется в тех случаях, при которых параметр типа должен обладать некоторыми характеристиками, требующимися для выполнения каких-либо операций над этим типом.
Для примера рассмотрим случай, когда в коллекции `T` (`Collection<T>`) объявлен метод получения элемента по имени (`getItemByName`).

typescript

```
class Collection<T> {
    private itemAll: T[] = [];

    public add(item: T): void {
        this.itemAll.push(item);
    }
```

```
    public getItemByName(name: string): T {
        return this.itemAll.find(item => item.name === name); // Error -
> Property 'name' does not exist on type 'T'
    }
}
```

При операции поиска в массиве возникнет ошибка. Ошибка возникнет потому, что в типе `T` не описано свойство `name`.
Для того чтобы ошибка исчезла, тип `T` должен расширить тип, в котором описано свойство `name`. В таком случае предпочтительней будет вариант объявления интерфейса `IName` с последующим его расширением.

typescript

```
interface IName {
    name: string;
}

class Collection<T extends IName> {
    private itemAll: T[] = [];

    public add(item: T): void {
        this.itemAll.push(item);
    }

    public getItemByName(name: string): T {
        return this.itemAll.find(item => item.name === name); // Ok
    }
}

abstract class Animal {
    constructor(readonly name: string) {}
}

class Bird extends Animal {}
class Fish extends Animal {}

let birdCollection: Collection<Bird> = new Collection();
birdCollection.add(new Bird('raven'));
birdCollection.add(new Bird('owl'));

let raven: Bird = birdCollection.getItemByName('raven'); // Ok

let fishCollection: Collection<Fish> = new Collection();
fishCollection.add(new Fish('shark'));
fishCollection.add(new Fish('barracuda'));

let shark: Fish = fishCollection.getItemByName('shark'); // Ok
```

Пример, когда параметр типа расширяет другой параметр типа, будет рассмотрен немного позднее.
Также не лишним будет заметить, что когда параметр типа расширяет другой тип, то в качестве аргумента типа можно будет передать только совместимый с ним тип.

typescript

```
interface Bird { fly(): void; }
interface Fish { swim(): void; }

interface IEgg<T extends Bird> { child: T; }
```

```
let v1: IEgg<Bird>; // Ok
let v2: IEgg<Fish>; // Error -> Type 'Fish' does not satisfy the
constraint 'Bird'
```

Кроме того, расширять можно любые предназначенные для расширения типы, полученные любым доступным путем.

typescript

```
interface IAnimal {
    name: string;
    age: number;
}

let animal: IAnimal;

class Bird<T extends typeof animal> {} // T extends IAnimal
class Fish<K extends keyof IAnimal> {} // K extends "name" | "age"
class Insect<V extends IAnimal[K], K extends keyof IAnimal> {} // V
extends string | number
class Reptile<T extends number | string, U extends number & string> {}
```

Помимо прочего одна важная и не очевидная особенность связанна с расширение параметром типа типа `any` . Может показаться что в таком случае над параметром типа будет возможно производить любые допустимые типом `any` операции. Но поскольку `any` допускает совершать над собой любые операции, то для повышения типобезопасности подобное поведение было отмененно для типов представляемых параметрами типа.

typescript

```
class ClassType<T extends any> {
    private f0: any = {}; // Ok
    private field: T = {}; // Error [0]

    constructor(){
        this.f0.notExistsMethod(); // Ok [1]
        this.field.notExistsMethod(); // Error [2]
    }
}
```

```
/**
 * Поскольку параметр типа расширяющий тип any
 * подрывает типобезолпасность программы, то вывод
 * типов такой параметр расценивает как принадлежащий
 * к типу unknown запрещающий любые операции над собой.
 *
 * [0] тип unknown не совместим с объектным типом {}.
 * [1] Ok на этапе компиляции и Error вов ремя выполнения.
 * [2] тип unknown не описывает метода notExistsMethod().
 */
```

[32.3] Параметра типа - значение по умолчанию = (generic parameter defaults)

Помимо прочего *TypeScript* позволяет указывать для параметров типа значение по умолчанию.
Значение по умолчанию указывается с помощью оператора равно `=`, слева от которого располагается параметр типа, а справа конкретный тип, либо другой параметр типа `T = Type`. Параметры, которым заданы значения по умолчанию, являются необязательными параметрами. Необязательные параметры типа должны быть перечислены строго после обязательных. Если параметр типа указывается в качестве типа по умолчанию, то ему самому должно быть задано значение по умолчанию, либо он должен расширять другой тип.

typescript

```
class T1<T = string> {} // Ok
class T2<T = U, U> {} // Error -> необязательное перед обязательным
class T3<T = U, U  = number> {} // Ok

class T4<T = U, U extends number> {} // Error -> необязательное перед
обязательным
class T5<U extends number, T = U> {} // Ok.
```

Кроме того, можно совмещать механизм установки значения по умолчанию и механизм расширения типа. В этом случае оператор равно `=` указывается после расширяемого типа.

```typescript
class T1 <T extends T2 = T3> {}
```

В момент, когда тип `T` расширяет другой тип, он получает признаки этого типа. Именно поэтому для параметра типа, расширяющего другой тип, в качестве типа по умолчанию можно указывать только совместимый с ним тип.
Чтобы было проще понять, нужно представить два класса, один из которых расширяет другой. В этом случае переменной с типом суперкласса можно в качестве значения присвоить объект его подкласса, но — не наоборот.

```typescript
class Animal {
    public name: string;
}

class Bird extends Animal {
    public fly(): void {}
}

let bird: Animal = new Bird(); // Ok
let animal: Bird = new Animal(); // Error
```

Тот же самый механизм используется для параметров типа.

```typescript
class Animal {
    public name: string;
}
```

```
class Bird extends Animal {
    public fly(): void {}
}

class T1 <T extends Animal = Bird> {} // Ok
// -------(   Animal   ) = Bird

class T2 <T extends Bird = Animal> {} // Error
// -------(   Bird   ) = Animal
```

Необходимо сделать акцент на том, что вывод типов обращает внимание только при работе с аргументами типа. Чтобы было более понятно вспомним ещё раз что механизм ограничения парамтров типа с помощью ключевого слова `extends` наделяет их указанными признаками служащими не только для ограничения аргументов типа, но и для выполнения опреаций в области видимости. Другими словами вывод типов при работе с аргументами типа берет во внимание как ограничивающий тип, так и тип указанный в качестве типа по умолчанию.

typescript

```
// ограничение типа T типом string
declare class A<T extends string>{
    constructor(value?: T)
}

// тип string устанавливается типу T в качестве типа по умолчанию
declare class B<T = string>{
    constructor(value?: T)
}


/**[0] */
let a0 = new A(); // Ok
let a1 = new A(`ts`); // Ok
let a2 = new A(0); // Error ->  Argument of type '0' is not assignable
to parameter of type 'string | undefined'.

/**[1] */
let b0 = new B(); // Ok
let b1 = new B(`ts`); // Ok
let b2 = new B(0); // Ok

/**
 * Вывод типов беретпри работе с аргументами типа
 * берет во внимание как тип расширяющий его [0]
```

```
 * так и тип указанный в качестве значения по умолчанию [1].
 */
```

Но при работе со значениями принадлежащих к типу `T` вывод типов берет в расчет только ограничивающий тип, но не тип выступающий в качестве типа по умолчанию.

typescript

```typescript
// ограничение типа T типом string
class A<T extends string>{
    constructor(value?: T){
        value?.toLowerCase(); // Ok /**[0] */
    }
}

// тип string устанавливается типу T в качестве типа по умолчанию
class B<T = string>{
    constructor(value?: T) {
        value?.toLowerCase(); // Error /**[1] */
    }
}

/**
 * Вы вод типов берет в расчет только ограничивающий тип [0]
 * но не тип указанный в качестве типа по умолчанию [1].
 *
 * [0] value наделено признаками типа T.
 * [1] у value отсутствуют какие-либо признаки.
 *
 */
```

Не будет лишним также рассмотреть отличия этих двух механизмов при работе вывода типов.

typescript

```typescript
// ограничение типа T типом string
declare class A<T extends string>{
    constructor(value?: T)
}
```

```
// тип string устанавливается типу T в качестве типа по умолчанию
declare class B<T = string>{
    constructor(value?: T)
}


let a0 = new A(); // Ok -> let a0: A<string>
let b0 = new B(); // Ok -> let b0: B<string>

let a1 = new A(`ts`); // Ok -> let a1: A<"ts">
let b1 = new B(`ts`); // Ok -> let b1: B<string>

let a2 = new A<string>(`ts`); // Ok -> let a2: A<string>
let b2 = new B<string>(`ts`); // Ok -> let b2: B<string>

let a3 = new A<number>(0); // Error
let b3 = new B<number>(0); // Ok -> let b3: B<number>
```

[32.4] Параметры типа - как тип данных

Параметры типа, указанные в угловых скобках при объявлении обобщенного типа, изначально не принадлежат ни к одному типу данных. Несмотря на это, компилятор расценивает параметры типа, как совместимые с такими типами как `any` и `never` и самим собой.

typescript

```
function f0<T>(p: any): T { // Ok, any совместим с T
    return p;
}

function f1<T>(p: never): T { // Ok, never совместим с T
    return p;
}

function f2<T>(p: T): T { // Ok, T совместим с T
    return p;
}
```

Если обобщенная коллекция, в качестве аргумента типа, получает тип данных объединение (`Union`), то все её элементы будут принадлежать к типу объединения. Простыми словами, элемент из такой коллекции не будет, без явного преобразования, совместим ни с одним из вариантов, составляющих тип объединение.

typescript

```typescript
interface IName { name: string; }

interface IAnimal extends IName {}

abstract class Animal implements IAnimal {
  constructor(readonly name: string) {}
}

class Bird extends Animal {
    public fly(): void {}
}

class Fish extends Animal {
    public swim(): void {}
}

class Collection<T extends IName> {
    private itemAll: T[] = [];

    public add(item: T): void {
        this.itemAll.push(item);
    }

    public getItemByName(name: string): T {
        return this.itemAll.find(item => item.name === name); // Ok
    }
}

let collection: Collection<Bird | Fish> = new Collection();
  collection.add(new Bird('bird'));
  collection.add(new Fish('fish'));

var bird: Bird = collection.getItemByName('bird'); // Error -> Type
'Bird | Fish' is not assignable to type 'Bird'
var bird: Bird = collection.getItemByName('bird') as Bird; // Ok
```

Но операцию приведения типов можно поместить (сокрыть) прямо в метод самой коллекции и тем самым упростить её использование. Для этого метод должен быть обобщенным, а его параметр типа, указанный в качестве возвращаемого из функции типа, расширять параметр типа самой коллекции.

typescript

```
// ...

class Collection<T extends IName> {
    private itemAll: T[] = [];

    public add(item: T): void {
        this.itemAll.push(item);
    }

    // 1. параметр типа U должен расширять параметр типа T
    // 2. возвращаемый тип указан как U
    // 3. возвращаемое значение нуждается в явном преобразовании к типу
U
    public getItemByName<U extends T>(name: string): U {
        return this.itemAll.find(item => item.name === name) as U; // Ok
    }
}

let collection: Collection<Bird | Fish> = new Collection();
 collection.add(new Bird('bird'));
 collection.add(new Fish('fish'));

var bird: Bird = collection.getItemByName('bird'); // Ok
var birdOrFish = collection.getItemByName('bird'); // Bad, var
birdOrFish: Bird | Fish
var bird = collection.getItemByName<Bird>('bird'); // Ok, var bird: Bird
```

Сокрытие приведения типов прямо в методе коллекции повысило “привлекательность” кода. Но, все же, в случаях, когда элемент коллекции присваивается конструкции без явной аннотации типа, появляется потребность вызывать обобщенный метод с аргументами типа.
Кроме того, нужно не забывать, что два разных объявления параметров типа несовместимы, даже если у них идентичные идентификаторы.

typescript

```
class Identifier<T> {
    array: T[] = [];

    method<T>(param: T): void  {
        this.array.push(param); // Error, T объявленный в сигнатуре
функции не совместим с типом T объявленном в сигнатуре класса
```

```
    }
}
```