TypeScript
Подробное Руководство

Книга и документация в одном

Дата последнего обновления: 9/9/2020

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*.

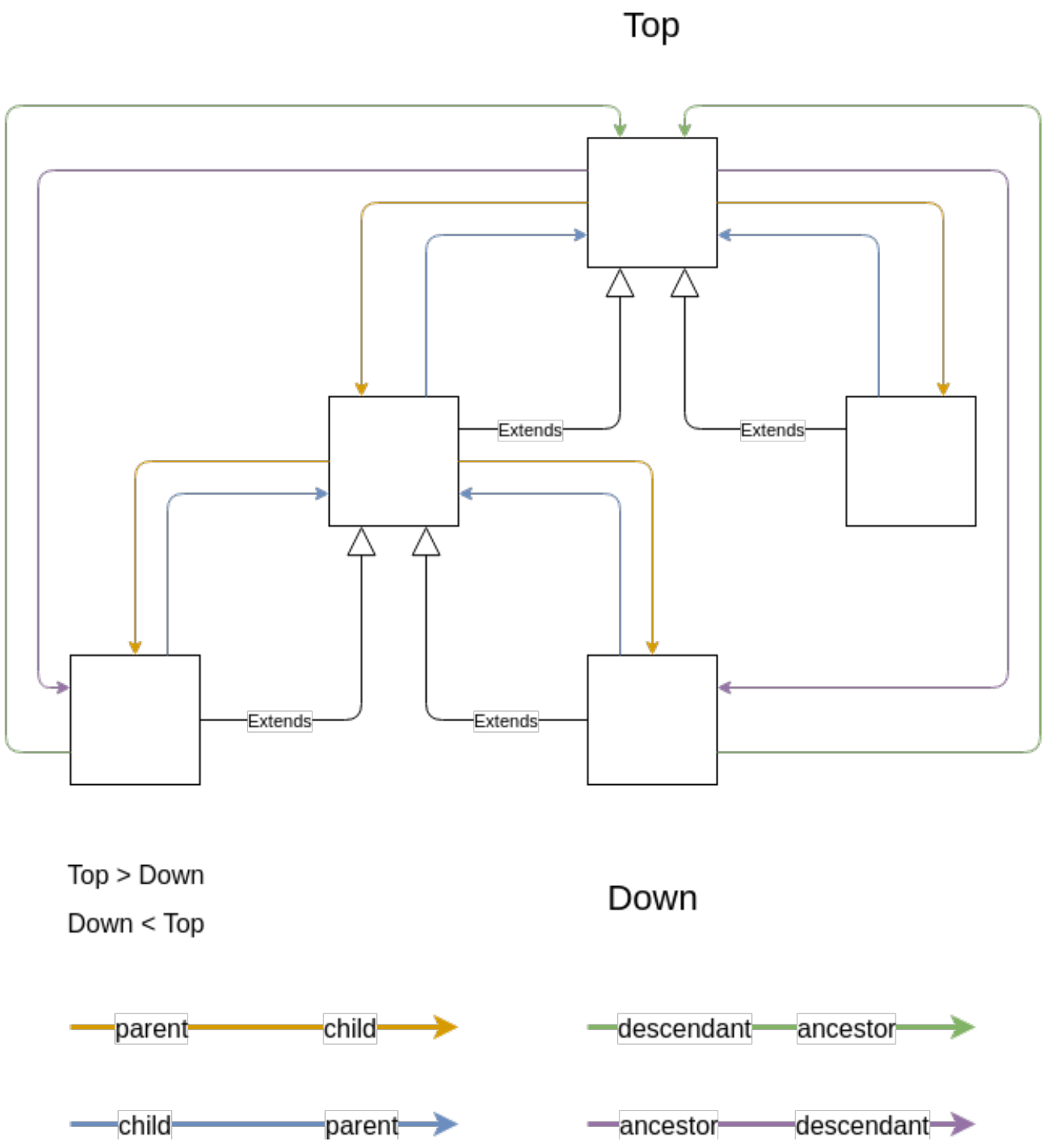
Если вы впервые сталкиваетесь с этим понятием и определение вариантности кажется бессмысленным набором слов, то не стоит расстраиваться, эта тема очень простая, в чем вы сами скоро убедитесь.

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

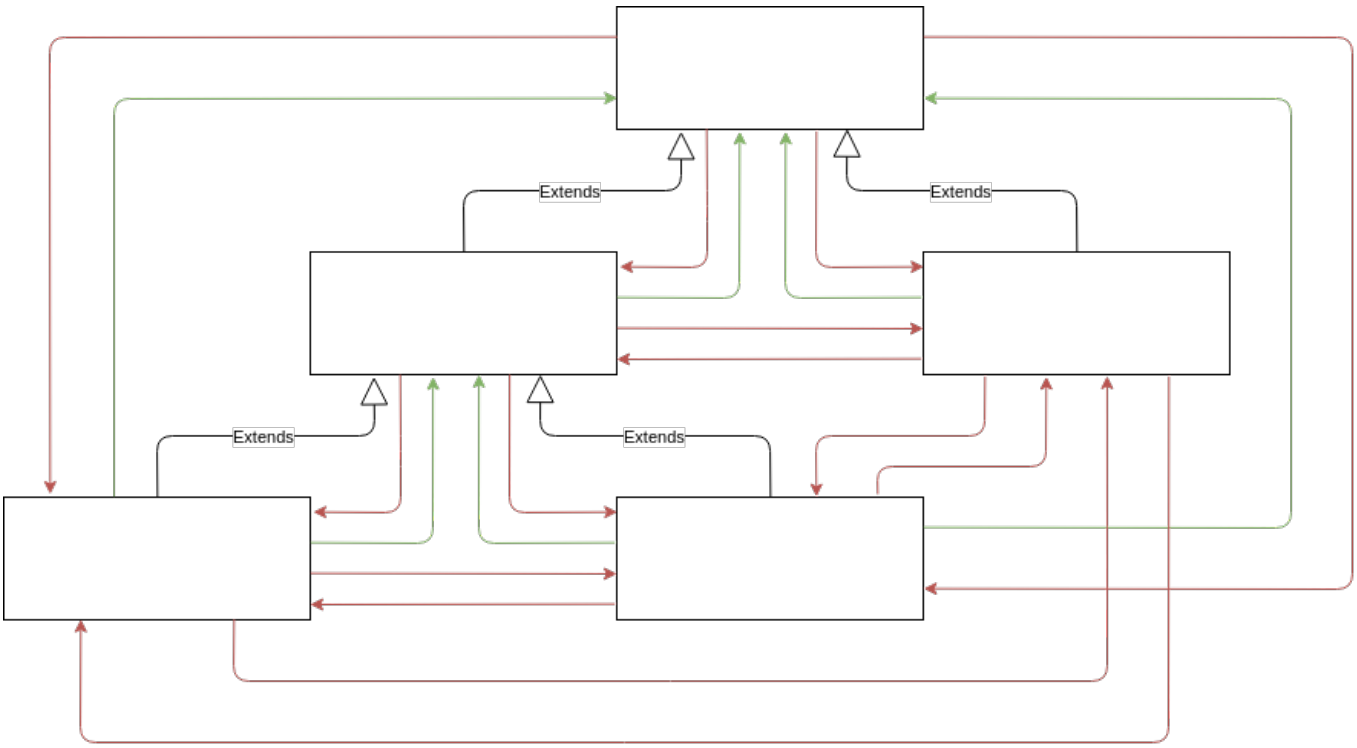
Иерархия наследования — это дерево, вверху которого расположен *корень*, самый *базовый тип* (*менее конкретный тип*), ниже которого располагаются его *подтипы* (*более конкретные типы*). В случаях преобразования подтипа к базовому типу говорят, что выполняется *восходящее преобразование* (*upcasting*). 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.



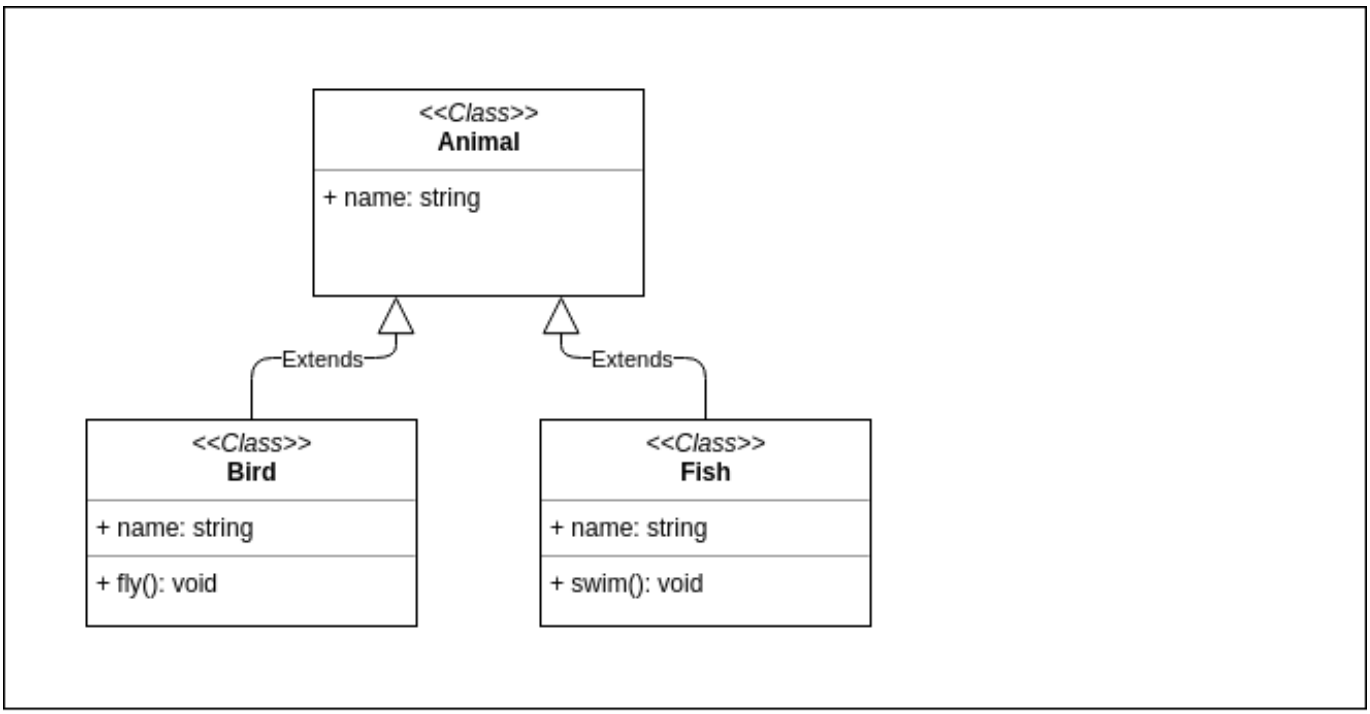
Этого вполне достаточно для того, чтобы приступить к разбору видов вариантности.

[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.



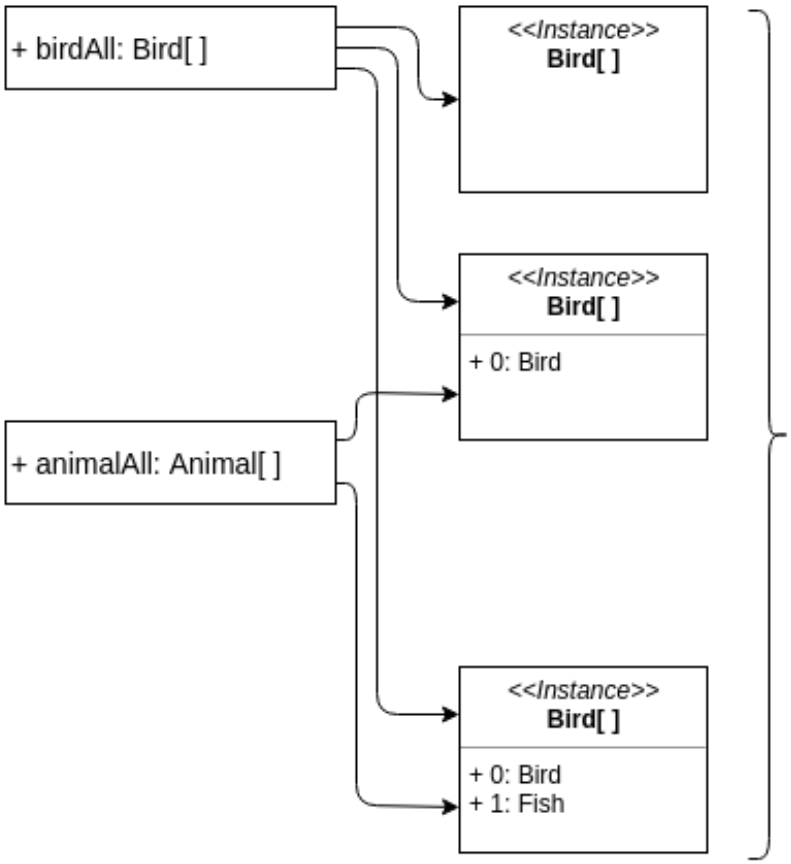
0 Создаем массив с типом Bird[ ] и присваиваем его ссылке с типом Bird[ ]

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

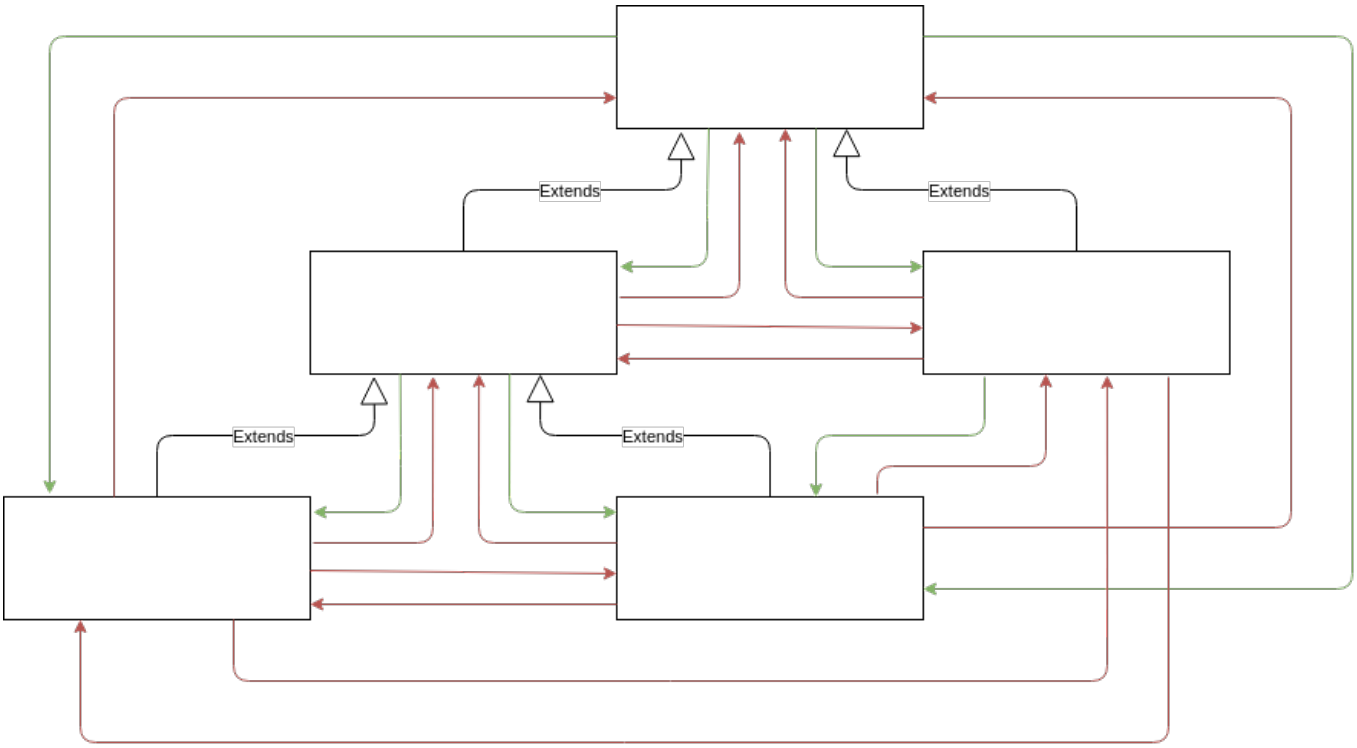
Теперь в массиве с типом Bird[ ] присутствует элемент принадлежащий к типу Fish



Ковариантность рекомендуется применять в местах, допускающих чтение.

[10.3] Контравариантность

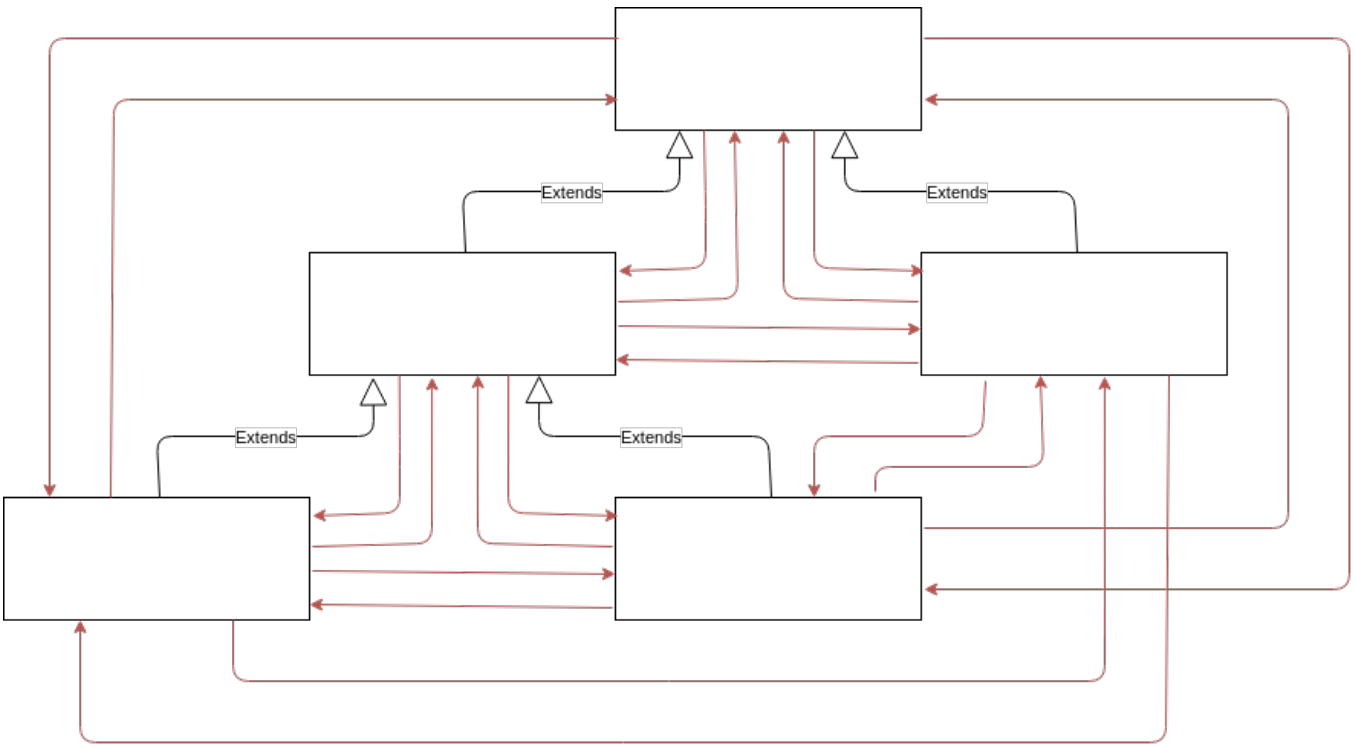
Контравариантность — это противоположный ковариантности механизм, позволяющий использовать менее конкретный тип там, где изначально предполагалось использовать более конкретный тип. Другими словами, совместимыми считаются типы имеющие отношения `A < B` и `A = B`.



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

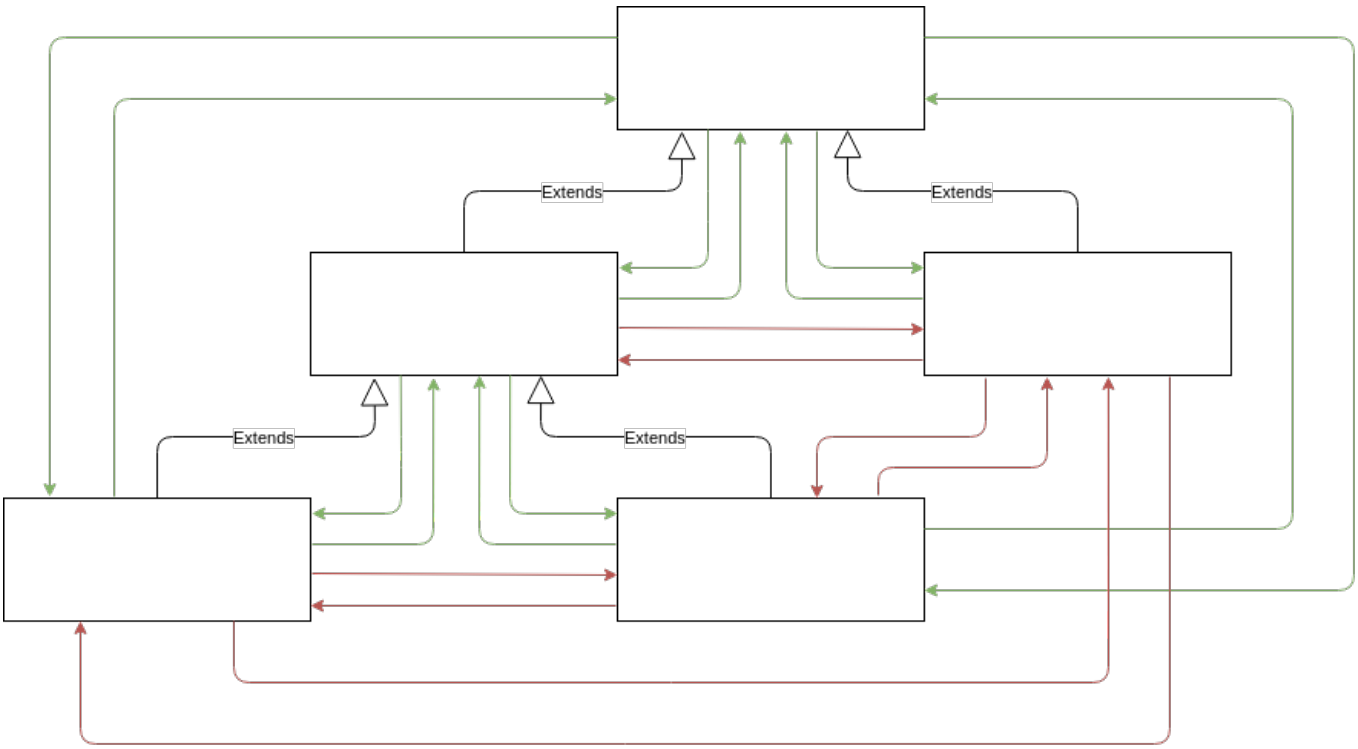
[10.4] Инвариантность

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.