Вариантность типов в Java и Kotlin

Иван Пономарёв

Иван Пономарёв

- Staff Engineer @ Synthesized.io
- Teaching Java @ МФТИ and Mainor

https://inponomarev.ru/corejava

Вариантность

- инвариантность
- ковариантность
- контравариантность

Звучит пугающе

Нам правда надо это знать?

Covariance and contravariance [edit]

There are many constructions in mathematics that would be functors but for the fact that they "turn morphisms around" and "reverse composition". We then define a **contravariant functor** *F* from *C* to *D* as a mapping that

- associates each object X in *C* with an object $F(X)$ in *D*,
- associates each morphism $f\colon X \to Y$ in *C* with a morphism $F(f)\colon F(Y) \to F(X)$ in *D* such that the following two conditions hold:
  - $F(\mathrm{id}_X) = \mathrm{id}_{F(X)}$ for every object X in *C*,
  - $F(g \circ f) = F(f) \circ F(g)$ for all morphisms $f\colon X \to Y$ and $g\colon Y \to Z$ in *C*.

Note that contravariant functors reverse the direction of composition.

Ordinary functors are also called **covariant functors** in order to distinguish them from contravariant ones.

Вариантность типов

- Даже опытные разработчики Java/Kotlin иногда не понимают эту тему.
- Это приводит к плохому дизайну внутренних API: дженерики либо вовсе не используются, либо используются неправильно.

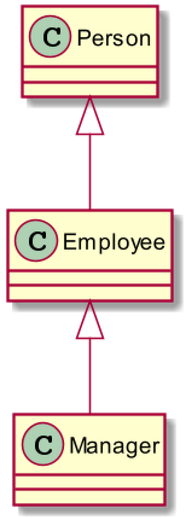
Причина

- Понимание этих концепций приводит к созданию более качественных API.
- API — это фундамент здания, реализация — это само здание.

Рассмотрим в исторической перспективе

- От типизированных массивов в Java через дженерики в Java до дженериков в Котлине
- А в конце спринг и интероп между джавой и котлином

Этот пример иерархии будет использован всюду



Возможные варианты

🚫 **не скомпилируется**

⚠️ **скомпилируется**, но будет **исключение** во время выполнения

✅ **скомпилируется** и выполнится **нормально**

☣️ **скопилируется** и выполнится, но случится **heap pollution**
(ситуация в которой переменная некоторого типа ссылается на объект несовместимого типа)

Вернёмся к корням: массивы в Java

Если джава-массив `a` не пуст, что нам вернёт `a[0]`?

type of `a`	**type of `a[0]`**
`Person[]`	
`Employee[]`	
`Manager[]`	
`Object[]`	

Вернёмся к корням: массивы в Java

Если джава-массив `a` не пуст, что нам вернёт `a[0]`?

type of `a`	**type of `a[0]`**
`Person[]`	`Person` (or `null`)
`Employee[]`	`Employee` (or `null`)
`Manager[]`	`Manager` (or `null`)
`Object[]`	`Object` (or `null`)

Если Java массив не пуст, каков результат задания значения его элементу?

🚫 ⚠️ ✅ ☣️

	Person	Employee	Manager	null
Object[]				❓
Person[]				❓
Employee[]				❓
Manager[]				❓

Если Java массив не пуст, каков результат задания значения его элементу?

	Person	Employee	Manager	null
Object[]				✅
Person[]				✅
Employee[]				✅
Manager[]				✅

Если Java массив не пуст, каков результат задания значения его элементу?

🚫 ⚠️ ✅ ☣️

	Person	Employee	Manager	null
Object[]				✅
Person[]				✅
Employee[]	❓			✅
Manager[]	❓	❓		✅

Если Java массив не пуст, каков результат задания значения его элементу?

🚫 ⚠️ ✅ ☣️

	`Person`	`Employee`	`Manager`	`null`
`Object[]`				✅
`Person[]`				✅
`Employee[]`	🚫			✅
`Manager[]`	🚫	🚫		✅

Если Java массив не пуст, каков результат задания значения его элементу?

🚫 ⚠️ ✅ ☣️

	`Person`	`Employee`	`Manager`	`null`
`Object[]`	❓	❓	❓	✅
`Person[]`	❓	❓	❓	✅
`Employee[]`	🚫	❓	❓	✅
`Manager[]`	🚫	🚫	❓	✅

Если Java массив не пуст, каков результат задания значения его элементу?

ЧТОА?

	`Person`	`Employee`	`Manager`	`null`
`Object[]`	✅⚠️	✅⚠️	✅⚠️	✅
`Person[]`	✅⚠️	✅⚠️	✅⚠️	✅
`Employee[]`	🚫	✅⚠️	✅⚠️	✅
`Manager[]`	🚫	🚫	✅⚠️	✅

Можем ли мы присвоить массив некоторых типов массиву других типов?

🚫 ⚠️ ✅ ☣️

To → From ↓	`Object[]`	`Person[]`	`Employee[]`	`Manager[]`
`Object[]`	✅	❓	❓	❓
`Person[]`		✅	❓	❓
`Employee[]`			✅	❓
`Manager[]`				✅

Можем ли мы присвоить массив некоторых типов массиву других типов?

🚫 ⚠️ ✅ ☣️

To → From ↓	`Object[]`	`Person[]`	`Employee[]`	`Manager[]`
`Object[]`	✅	🚫	🚫	🚫
`Person[]`		✅	🚫	🚫
`Employee[]`			✅	🚫
`Manager[]`				✅

Можем ли мы присвоить массив некоторых типов массиву других типов?

🚫 ⚠️ ✅ ☣️

To → From ↓	`Object[]`	`Person[]`	`Employee[]`	`Manager[]`
`Object[]`	✅	🚫	🚫	🚫
`Person[]`	❓	✅	🚫	🚫
`Employee[]`	❓	❓	✅	🚫
`Manager[]`	❓	❓	❓	✅

Можем ли мы присвоить массив некоторых типов массиву других типов?

🚫 ⚠️ ✅ ☣️

To → From ↓	`Object[]`	`Person[]`	`Employee[]`	`Manager[]`
`Object[]`	✅	🚫	🚫	🚫
`Person[]`	✅	✅	🚫	🚫
`Employee[]`	✅	✅	✅	🚫
`Manager[]`	✅	✅	✅	✅

Как насчёт такого?

```
Manager[] managers = new Manager[10];
Person[] persons = managers; //должно скомпилироваться и выполниться
persons[0] = new Person();   //строка 1 ??
Manager m = managers[0];     //строка 2 ?!
```

Как насчёт такого?

⚠️⚠️⚠️⚠️

```
Manager[] managers = new Manager[10];
Person[] persons = managers; //должно скомпилироваться и выполниться
persons[0] = new Person();   //строка 1 ??
Manager m = managers[0];     //строка 2 ?!
```

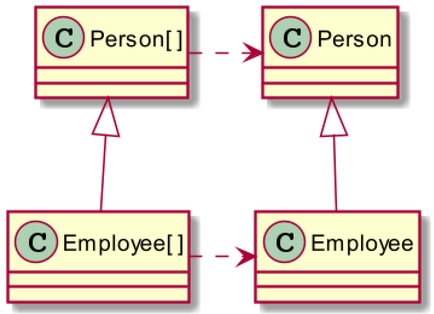
⚠️ `ArrayStoreException` at line 1.

Промежуточные итоги насчёт Java-массивов

- Массивы в джаве *reified* (*овеществлены*) — т. е. они хранят в себе информацию о типах своих элементов во время выполнения, и также они проверяют типы во время выполнения.

Промежуточные итоги насчёт Java-массивов

- Массивы в джае *ковариантны* — что означает следующее:



- Ковариантность безопасна когда мы читаем значения, но может привести к проблемам когда мы пишем значения. Для джава-массивов проблема частично решается проверкой типов в рантайме из-за овеществления (reification) массивов.

В Java и Kotlin, какой тип возвращает `list.get(0)`?

(допустим, что список не пуст)

`List<Person>`	`Person`
`List<Employee>`	`Employee`
`List<Manager>`	`Manager`
`List<?>`	❓
`List<*>`	❓

В Java и Kotlin, какой тип возвращает `list.get(0)`?

(допустим, что список не пуст)

`List<Person>`	`Person`
`List<Employee>`	`Employee`
`List<Manager>`	`Manager`
`List<?>`	`Object`
`List<*>`	`Any?`

Каков результат выполнения `list.add(…)` в Джаве и Котлине?

	`Person`	`Employee`	`Manager`	`null`
`List<Person>`				
`List<Employee>`				
`List<Manager>`				
`List<?>` / `List<*>`				

Каков результат выполнения `list.add(...)` в Джаве и Котлине?

Ха-ха, метода `add` нет в котлиновском `List`!

	`Person`	`Employee`	`Manager`	`null`
`List<Person>`	🙃	🙃	🙃	🙃
`List<Employee>`	🙃	🙃	🙃	🙃
`List<Manager>`	🙃	🙃	🙃	🙃
`List<?>` / `List<*>`	🙃	🙃	🙃	🙃

Каков результат выполнения `list.add(...)` в Джаве и Котлине?

🚫 ⚠️ ✅ ☣️

	`Person`	`Employee`	`Manager`	`null`
`List<Person>` / `MutableList<Person?>`	❓	❓	❓	
`List<Employee>` / `MutableList<Employee?>`		❓	❓	
`List<Manager>` / `MutableList<Manager?>`			❓	
`List<?>`				
`MutableList<*>`				

Каков результат выполнения `list.add(…)` в Джаве и Котлине?

🚫 ⚠️ ✅ ☣️

	`Person`	`Employee`	`Manager`	`null`
`List<Person>` / `MutableList<Person?>`	✅	✅	✅	
`List<Employee>` / `MutableList<Employee?>`		✅	✅	
`List<Manager>` / `MutableList<Manager?>`			✅	
`List<?>`				
`MutableList<*>`				

Каков результат выполнения `list.add(…)` в Джаве и Котлине?

🚫 ⚠️ ✅ ☣️

	Person	Employee	Manager	null
`List<Person>` / `MutableList<Person?>`	✅	✅	✅	
`List<Employee>` / `MutableList<Employee?>`	❓	✅	✅	
`List<Manager>` / `MutableList<Manager?>`	❓	❓	✅	
`List<?>`				
`MutableList<*>`				

Каков результат выполнения `list.add(…)` в Джаве и Котлине?

🚫 ⚠️ ✅ ☣️

	`Person`	`Employee`	`Manager`	`null`
`List<Person>` / `MutableList<Person?>`	✅	✅	✅	
`List<Employee>` / `MutableList<Employee?>`	🚫	✅	✅	
`List<Manager>` / `MutableList<Manager?>`	🚫	🚫	✅	
`List<?>`				
`MutableList<*>`				

Каков результат выполнения `list.add(…)` в Джаве и Котлине?

🚫 ⚠️ ✅ ☣️

	`Person`	`Employee`	`Manager`	`null`
`List<Person>` / `MutableList<Person?>`	✅	✅	✅	
`List<Employee>` / `MutableList<Employee?>`	🚫	✅	✅	
`List<Manager>` / `MutableList<Manager?>`	🚫	🚫	✅	
`List<?>`	❓	❓	❓	
`MutableList<*>`	❓	❓	❓	

Каков результат выполнения `list.add(…)` в Джаве и Котлине?

🚫 ⚠️ ✅ ☣️

	`Person`	`Employee`	`Manager`	`null`
`List<Person>` / `MutableList<Person?>`	✅	✅	✅	
`List<Employee>` / `MutableList<Employee?>`	🚫	✅	✅	
`List<Manager>` / `MutableList<Manager?>`	🚫	🚫	✅	
`List<?>`	🚫	🚫	🚫	
`MutableList<*>`	🚫	🚫	🚫	

Каков результат выполнения `list.add(…)` в Джаве и Котлине?

	Person	Employee	Manager	null
`List<Person>` / `MutableList<Person?>`	✅	✅	✅	❓
`List<Employee>` / `MutableList<Employee?>`	🚫	✅	✅	❓
`List<Manager>` / `MutableList<Manager?>`	🚫	🚫	✅	❓
`List<?>`	🚫	🚫	🚫	❓
`MutableList<*>`	🚫	🚫	🚫	❓

Каков результат выполнения `list.add(…)` в Джаве и Котлине?

🚫 ⚠️ ✅ ☣️

	`Person`	`Employee`	`Manager`	`null`
`List<Person>` / `MutableList<Person?>`	✅	✅	✅	✅
`List<Employee>` / `MutableList<Employee?>`	🚫	✅	✅	✅
`List<Manager>` / `MutableList<Manager?>`	🚫	🚫	✅	✅
`List<?>`	🚫	🚫	🚫	✅
`MutableList<*>`	🚫	🚫	🚫	🚫

Можем ли мы присвоить эти списки друг другу?

🚫 ⚠️ ✅ ☣️

To → From ↓	`List / MutableList <Person>`	`List / MutableList <Employee>`	`List / MutableList <Manager>`	`List<?>/ MutableList<*>`
`List/MutableList <Person>`	✅	🚫	🚫	✅
`List/MutableList <Employee>`	❓	✅	🚫	✅
`List/MutableList <Manager>`	❓	❓	✅	✅
`List<?>/ MutableList<*>`	❓	❓	❓	✅

Можем ли мы присвоить эти списки друг другу?

🚫 ⚠️ ✅ ☣️

To → From ↓	`List / MutableList <Person>`	`List / MutableList <Employee>`	`List / MutableList <Manager>`	`List<?>/ MutableList<*>`
`List/MutableList <Person>`	✅	🚫	🚫	✅
`List/MutableList <Employee>`	🚫	✅	🚫	✅
`List/MutableList <Manager>`	🚫	🚫	✅	✅
`List<?> / MutableList<*>`	🚫	🚫	🚫	✅

Промежуточные итоги

- В джаве дженерики *инвариантны*, т. е. их можно присвоить только переменным точно такого же типа.

Иначе рискуем получить:

```
List<Manager> managers = new ArrayList<>();
List<Person> persons = managers; //не скомпилируется
persons.add(new Person());  //в рантайме не проверишь
```

Промежуточные итоги

- В котлине, `MutableList` всё ещё инвариантен.
- Так же и `Array<T>`, являющийся обёрткой вокруг нативных джава-массивов. Таким образом, нельзя присвоить `Array<String>` в `Array<Any>`.

А что насчёт иммутабельных списков в Котлине?

🚫 ⚠️ ✅ ☣️

To → From ↓	`List<Person>`	`List<Employee>`	`List<Manager>`	`List<*>`
`List<Person>`	✅	🚫	🚫	✅
`List<Employee>`	❓	✅	🚫	✅
`List<Manager>`	❓	❓	✅	✅
`List<*>`	🚫	🚫	🚫	✅

А что насчёт иммутабельных списков в Котлине?

🚫 ⚠️ ✅ ☣️

To → From ↓	`List<Person>`	`List<Employee>`	`List<Manager>`	`List<*>`
`List<Person>`	✅	🚫	🚫	✅
`List<Employee>`	✅	✅	🚫	✅
`List<Manager>`	✅	✅	✅	✅
`List<*>`	🚫	🚫	🚫	✅

И мы уже знаем, почему это безопасно!

В отличие от Джавы, мы можем объявить вариантность по месту декларации

Java	**Kotlin**
`public interface List<E> extends Collection<E> {...}`	`public interface List<out E> : Collection<E> {...}`

- `List<E>` в котлине это **ковариантный** тип.
- Но это безопасно, т. к. мы только читаем записи из котлиновского `List`-а, и никогда не пишем их!

Почему проверка во времени выполнения не подходит для мутабельных списков?

- Во-первых, мы хотим избежать исключений во времени выполнения (оставим их динамическим языкам)
- Дженерики в Джаве и Котлине используют type erasure.
  Это значит, что во время выполнения `List<T>` не знает значения `T` и не может выполнить проверку.

JVM и дженерики

- Появились в Java 5 (2004 год)
- Существовала проблема обратной совместимости
- Были реализованы как фича языка, а не фича платформы
- Привет, Type Erasure

Сырые типы

Generic Type (source)

```
class Pair<T> {
  private T first;
  private T second;
  Pair(T first,
       T second)
   {this.first = first;
    this.second = second;}
  T getFirst()
   {return first; }
  T getSecond()
   {return second; }
  void setFirst(T newValue)
   {first = newValue;}
  void setSecond(T newValue)
   {second = newValue;}
}
```

Raw Type (compiled)

```
class Pair {
  private Object first;
  private Object second;
  Pair(Object first,
       Object second)
   {this.first = first;
    this.second = second;}
  Object getFirst()
   {return first; }
  Object getSecond()
   {return second; }
  void setFirst(Object newValue)
   {first = newValue;}
  void setSecond(Object newValue)
   {second = newValue;}
}
```

Граничные типы вместо Object

Generic Type (source)

```
class Pair<T extends Employee>{
  private T first;
  private T second;
  Pair(T first,
       T second)
   {this.first = first;
    this.second = second;}
  T getFirst()
   {return first; }
  T getSecond()
   {return second; }
  void setFirst(T newValue)
   {first = newValue;}
  void setSecond(T newValue)
   {second = newValue;}
}
```

Raw Type (compiled)

```
class Pair {
  private Employee first;
  private Employee second;
  Pair(Employee first,
       Employee second)
   {this.first = first;
    this.second = second;}
  Employee getFirst()
   {return first; }
  Employee getSecond()
   {return second; }
  void setFirst(Employee newValue)
   {first = newValue;}
  void setSecond(Employee newValue)
   {second = newValue;}
}
```

Вызовы методов

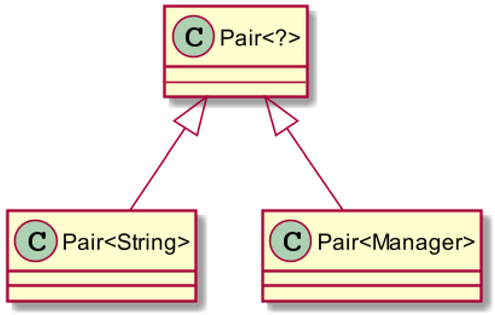
Source code	Compiled
`Pair<Manager> buddies =` `  new Pair<>();`	`Pair buddies =` `  new Pair();`
`/*контроль типов` `во время компиляции*/`	`/*контроль типов не нужен -- всё проверили` `во время компиляции!*/`
`buddies.setFirst(cfo);` `buddies.setSecond(cto);`	`buddies.setFirst(cfo);` `buddies.setSecond(cto);`

`/*приведение типов не нужно*/`
`Manager buddy =`
`  buddies.getFirst();`
	`/*приведение типов вставлено компилятором*/`
	`Manager buddy =`
	`  (Manager) buddies.getFirst();`

Итог: как это работает

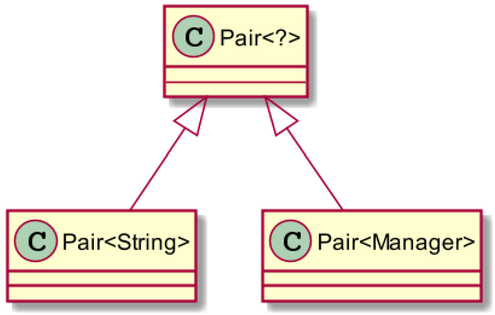
- Параметризованных классов в JVM нет, только обычные классы и методы.
- Типовые параметры заменяются на Object или на свою границу.
- Для сохранения полиморфизма добавляются связывающие методы (bridge methods).
- Сведение типов добавляется по мере необходимости

Основное ограничение Type Erasure



Java	Kotlin
`if (a instanceof Pair<String>) ...`	`if (a is Pair<String>) ...`

Основное ограничение Type Erasure

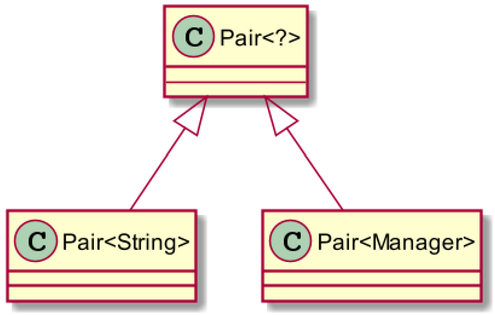


Java	Kotlin
`if (a instanceof Pair<String>) ...`	`if (a is Pair<String>) ...`

 Мы не можем выяснить типовый параметр во время выполнения!

Основное ограничение Type Erasure

✅✅✅✅



Java	Kotlin
`if (a instanceof Pair<?>) ...`	`if (a is Pair<*>) ...`

✅ Хотя нам бы хотелось знать больше...

Массивы и дженерики — лютые враги

```
List<String>[] a = new ArrayList<String>[10];
```

Массивы и дженерики — лютые враги

```
List<String>[] a = new ArrayList<String>[10];
```

 "Generic Array Creation", ведь такой массив не может содержать полную информацию о типе своих элементов!

(Создав такой массив, вместо `List<String>` в рантайме можно будет подпихнуть `List<Manager>` и устроить ).

Взгляните на вот это

Java

```
Pair<Integer> intPair =
        new Pair<>(42, 0);
Pair<?> pair = intPair;
Pair<String> stringPair =
         (Pair<String>) pair;
stringPair.b = "foo";
System.out.println(
         intPair.a * intPair.b);
```

Kotlin

```
var intPair = Pair<Int>(42, 0)
var pair: Pair<*> = intPair
var stringPair: Pair<String> =
            pair as Pair<String>
stringPair.b = "foo"
println(intPair.a * intPair.b)
```

Взгляните на вот это

Java

```
Pair<Integer> intPair =
        new Pair<>(42, 0);
Pair<?> pair = intPair;
Pair<String> stringPair =
         (Pair<String>) pair;
stringPair.b = "foo";
System.out.println(
         intPair.a * intPair.b);
```

Kotlin

```
var intPair = Pair<Int>(42, 0)
var pair: Pair<*> = intPair
var stringPair: Pair<String> =
            pair as Pair<String>
stringPair.b = "foo"
println(intPair.a * intPair.b)
```

 `intPair` превратился в монстра, это больше не пара целочисленных значений!

Как быть, если хочется такого?


```
MyList<Manager> managers = ...
MyList<Employee> employees = ...

//Допустимые варианты, хотим чтоб компилировалось!
employees.addAllFrom(managers);
managers.addAllTo(employees);

//Недопустимые варианты, не хотим чтоб компилировалось!
managers.addAllFrom(employees);
employees.addAllTo(managers);
```

Наивный подход на джаве…

```
class MyList<E> implements Iterable<E> {
    void add(E item) { ... }
    void addAllFrom(MyList<E> list) {
        for (E item : list) this.add(item);
    }
    void addAllTo(MyList<E> list) {
        for (E item : this) list.add(item);
    }
  ...}
MyList<Manager> managers = ...;  MyList<Employee> employees = ...;

employees.addAllFrom(managers); managers.addAllTo(employees);
```

Наивный подход на джаве…

🚫🚫🚫🚫

```
class MyList<E> implements Iterable<E> {
    void add(E item) { ... }
    void addAllFrom(MyList<E> list) {
        for (E item : list) this.add(item);
    }
    void addAllTo(MyList<E> list) {
        for (E item : this) list.add(item);
    }
  ...}
MyList<Manager> managers = ...;  MyList<Employee> employees = ...;

employees.addAllFrom(managers); managers.addAllTo(employees);
```

🚫 Не скомпилируется из-за инвариантности типового параметра

Такой же наивный подход на котлине…

🚫⚠️✅☣️

```
class MyList<E> : Iterable<E> {
  fun add(item: E) {...}
  fun addAllFrom(list: MyList<E>) {
    for (item in list) add(item)
  }
  fun addAllTo(list: MyList<E>) {
    for (item in this) list.add(item)
  }
...}

val managers: MyList<Manager> = ...; val employees: MyList<Employee> = ...

employees.addAllFrom(managers); managers.addAllTo(employees)
```

Такой же наивный подход на котлине…

🚫🚫🚫🚫

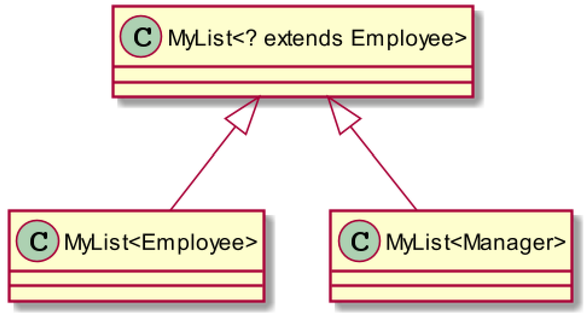
```
class MyList<E> : Iterable<E> {
  fun add(item: E) {...}
  fun addAllFrom(list: MyList<E>) {
    for (item in list) add(item)
  }
  fun addAllTo(list: MyList<E>) {
    for (item in this) list.add(item)
  }
...}

val managers: MyList<Manager> = ...; val employees: MyList<Employee> = ...

employees.addAllFrom(managers); managers.addAllTo(employees)
```

🚫 Не скомпилируется из-за инвариантности типового параметра

Java Covariant Wildcard Types

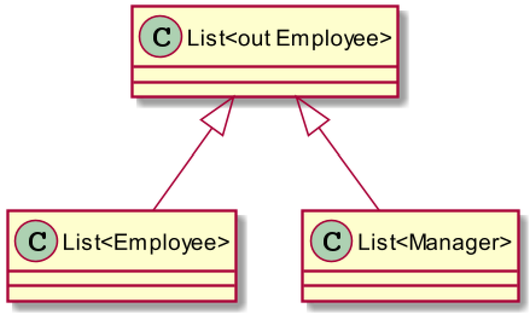



```
class MyList<E> implements Iterable<E> {
    void addAllFrom (List<? extends E> list){
       for (E item: list) add(item); }
}
MyList<Manager> managers = ...; MyList<Employee> employees = ...
employees.addAllFrom(managers);
```

✅ Если `E` — это `Employee`,

то `MyList<Manager>` "пролезет" в `MyList<? extends E>`!!

Kotlin Covariant Use-Site Type Projections




```
class MyList<E> : Iterable<E> {
    fun addAllFrom(list: MyList<out E>) {
        for (item in list) add(item) }
}
val managers: MyList<Manager> = ... ; val employees: MyList<Employee> = ...
employees.addAllFrom(managers)
```

✅ Если `E` это `Employee`, `MyList<Manager>` "пролезет" в `MyList<out E>`!!

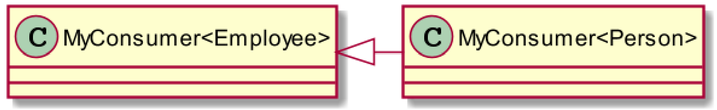
Declaration-site covariance в котлине для возвращаемых значений

Вот то, чего в джаве нет:

```
class MyImmutablePair<out E>(val a: E, val b: E)
```

- В таком классе мы можем декларировать только методы, которые возвращают что-либо с типом `E`, но не доступные методы, у которых есть аргументы с типом `E`.
- Параметры конструктора и приватные методы с типом `E` — это OK!

Declaration-site covariance в котлине для возвращаемых значений



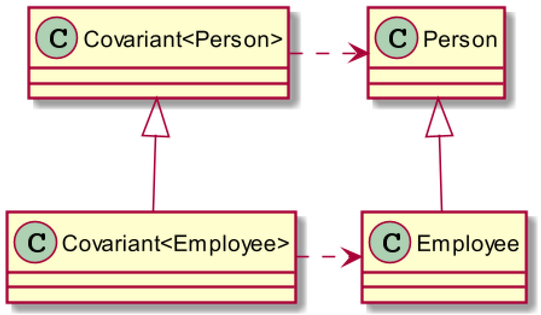

```
class MyList<E> : Iterable<E> {
  //Больше не заморачиваемся с вариантностью по месту использования!
  fun addAllFrom(pair: MyImmutablePair<E>){
    add(pair.a); add(pair.b) }
  ...
}

val twoManagers: MyImmutablePair<Manager> = ...
employees.addAllFrom(twoManagers)
```

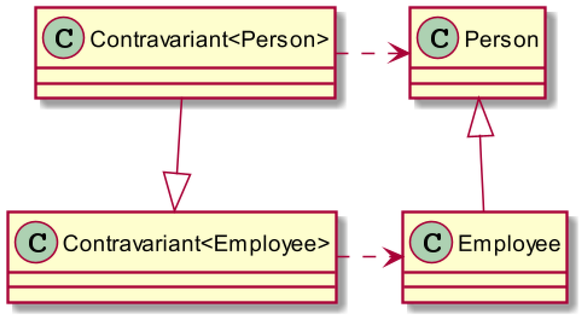
✅ `MyImmutablePair<Manager>` может быть присвоен к переменной `MyImmutablePair<Employee>`

До сих пор мы говорили только про ковариантность. А что насчёт контравариантности?

`Covariant<out E>`

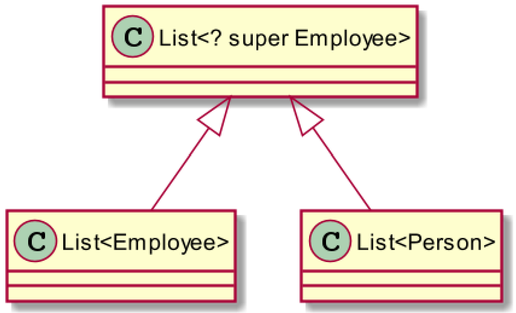


`Contravariant<in E>`



`Predicate<Person>` может заменить собой `Predicate<Employee>` и `Predicate<Manager>`, таким образом он может быть рассмотрен как их субтип.

Java wildcard contravariant types

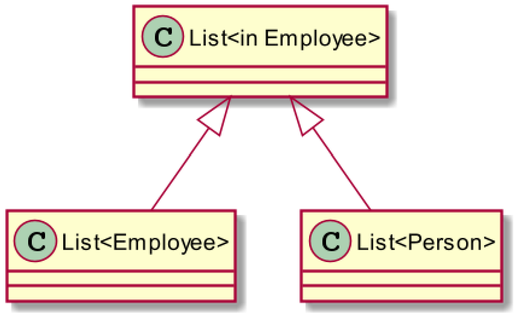



```
class MyList<E> implements Iterable<E> {
    void addAllTo (List<? super E> list) {
       for (E item: this) list.add(item); }
}
MyList<Employee> employees = ...; MyList<Person> people = ...;

employees.addAllTo(people);
```

✅ If `E` is `Employee`, `MyList<Person>` will do as `MyList<? super E>`!

Kotlin Contravariant Use-Site Type Projections




```
class MyList<E> : Iterable<E> {
    fun addAllTo(list: MyList<in E>) {
        for (item in this) list.add(item) }
}
val employees: MyList<Employee> = ... ; val people: MyList<Person> = ...

employees.addAllTo(people)
```

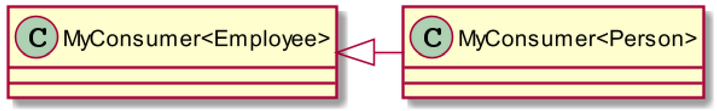
✅ Если `E` это `Employee`, `MyList<Person>` подойдёт вместо `MyList<in E>`!

Declaration-site контравариантность в котлине для значений только для записи

```
class MyConsumer<in E> {
    fun consume(p: E){
        ...
    }
}
```

- Теперь мы можем определять методы, которые имеют аргументы с типом `E`, но мы не можем открывать на чтение что угодно с типом `E`.
- Мы можем иметь приватные поля с типом `E`, и даже приватные методы, возвращающие `E`.

Declaration-site контравариантность в котлине для значений только для записи




```
class MyList<E> : Iterable<E> {
  //Больше не заморачиваемся с вариантностью по месту использования!
  fun addAllTo(consumer: MyConsumer<E>){
        for (item in this) consumer.consume(item)
  }
  ...
}
val employees: MyList<Employee> = ...
val personConsumer: MyConsumer<Person> = ...
employees.addAllTo(personConsumer)
```

✅ `MyConsumer<Person>` может быть присвоен переменной с типом `MyConsumer<Employee>`

Что мы можем делать с объектом типа `in` в Котлине?

```
fun <E> doSomething (list: MyList<in E>) {
    list.add(null)
}
```

Что мы можем делать с объектом типа `in` в Котлине?

```
fun <E> doSomething (list: MyList<in E>) {
    list.add(null)
}
```

Что мы можем делать с объектом типа `in` в Котлине?

```
fun <E> doSomething (list: MyList<in E?>) {
    list.add(null)
}
```

Что мы можем делать с объектом типа `in` в Котлине?

✅✅

```
fun <E> doSomething (list: MyList<in E?>) {
    list.add(null)
}
```

✅

Что мы можем делать с объектом типа `in` в Котлине?

🚫✅

```
fun <E> doSomething (list: MyList<in E>) {
    val first: Any = list.first()
}
```

Что мы можем делать с объектом типа `in` в Котлине?

```
fun <E> doSomething (list: MyList<in E>) {
    val first: Any = list.first()
}
```

Что мы можем делать с объектом типа `in` в Котлине?

🚫✅

```
fun <E> doSomething (list: MyList<in E>) {
    val first: Any? = list.first()
}
```

Что мы можем делать с объектом типа `in` в Котлине?

✅✅

```
fun <E> doSomething (list: MyList<in E>) {
    val first: Any? = list.first()
}
```

✅

Мнемоническое правило для Джавы

PECS

Producer — Extends, Consumer — Super

```
public static <T> T max(Collection<? extends T> coll,
                        Comparator<? super T> comp)

Collections.max(List<Integer>, Comparator<Number>)
Collections.max(List<Manager>, Comparator<Person>)
```

Producer — Out, Consumer — In

Простое правило:

- если мы собираемся только читать значения типа `T`, лучше объявить его как `out` по месту декларации
- если мы собираемся только писать значения `T`, лучше объявить его как `in` по месту декларации

Declaration-site variance — не просто "сахар"!

- Классы Kafka Streams `KStream<K, V>`, `KTable<K, V>` семантически ковариантны: поток `Employee` можно безопасно рассматривать как поток `Person`!!
- Это нельзя поправить просто добавлением `? extends` повсюду…

Проблема из реальной жизни при дизайне KStreams API

```
/* БИБЛИОТЕЧНЫЙ КОД */
class KStream<E> { ... }
class Processor<E> {
  void withFunction(Function<? super KStream<E>,
                             ? extends KStream<E>> chain) {...}
}
/* ПОЛЬЗОВАТЕЛЬСКИЙ КОД */
KStream<Employee> transformA(KStream<Employee> s) {...}
KStream<Manager> transformB(KStream<Person> s) {...}

/* Мы хотим использовать ссылки на методы! */
Processor<Employee> processor = ...
processor.withFunction(this::transformA);
processor.withFunction(this::transformB);
```

⛔ Хотя `this::transformA` сработает, `this::transformB` не скомпилируется с ошибкой "KStream<Employee> is not convertible to KStream<Person>"

Попробуем починить?…

```
/* БИБЛИОТЕЧНЫЙ КОД */
class KStream<E> { ... }
class Processor<E> {
  void withFunction(Function<? super KStream<? extends E>,
                             ? extends KStream<? extends E>> chain) {...}
}
/* ПОЛЬЗОВАТЕЛЬСКИЙ КОД */
KStream<Employee> transformA(KStream<Employee> s) { ... }
KStream<Manager> transformB(KStream<Person> s) { ... }

Processor<Employee> processor = new Processor<>();
processor.withFunction(this::transformA);
processor.withFunction(this::transformB);
```

 Обе строчки не скомпилируются с чем-то вроде "KStream<capture of ? super Employee> is not convertible to KStream<Employee>"

Пробуем починить

Тем временем в Котлине…

```
/* БИБЛИОТЕЧНЫЙ КОД */
class KStream<out E>
class Processor<E> {
  fun withFunction(chain: (KStream<E>) -> KStream<E>) {}
}
/* ПОЛЬЗОВАТЕЛЬСКИЙ КОД */
fun transformA(s: KStream<Employee>): KStream<Employee> { ... }
fun transformB(s: KStream<Person>): KStream<Manager> { ... }

val processor: Processor<Employee> = Processor()
processor.withFunction(this::transformA)
processor.withFunction(this::transformB)
```

✅ Всё скомпилируется и будет работать как надо!

Выводы

- Использовать готовые дженерики просто
- Чтобы создавать свои дженерики, надо кое-что понимать

Выводы

- Котлин предлагает улучшения дженериков по сравнению с Джавой, делая использование готовых дженериков ещё более простым
- Но чтобы создавать свои дженерики в Котлине, ещё важнее помнить про главные принципы

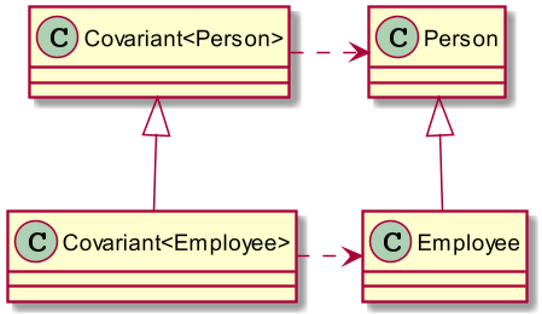
Covariance vs. Contravariance

Covariance

`? extends`

`out`

read-only

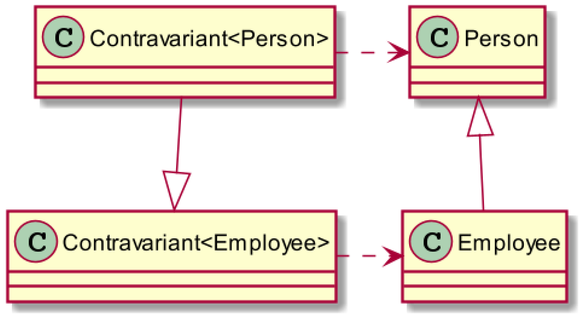


Contravariance

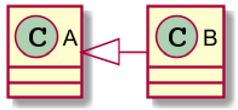
`? super`

`in`

write-only



Covariance vs. Invariance vs. Contravariance



Covariance			Invariance			Contravariance		
To →	C<A>	C<B>	To →	C<A>	C<B>	To →	C<A>	C<B>
From ↓			From ↓			From ↓		
C<A>	✅	🚫	C<A>	✅	🚫	C<A>	✅	✅
C<B>	✅	✅	C<B>	🚫	✅	C<B>	🚫	✅

Спасибо за внимание!

Producer Extends, Consumer Super

Producer Out, Consumer In

ivan@synthesized.io

@inponomarev