



weekens 5 января 2014 в 08:47

Использование generic wildcards для повышения удобства Java API

Java, Программирование

Доброго времени суток!

Этот пост для тех, кто работает над очередным API на языке Java, либо пытается усовершенствовать уже существующий. Здесь будет дан простой совет, как с помощью конструкций `? extends T` и `? super T` можно значительно повысить удобство вашего интерфейса.

[Перейти сразу к сути](#)

Исходный API

Предположим, у вас есть интерфейс некоего хранилища объектов, параметризованный, допустим, двумя типами: тип ключа (K) и тип значения (V). Интерфейс определяет набор методов для работы с данными в хранилище:

```
public interface MyObjectStore<K, V> {
    /**
     * Кладёт значение в хранилище по заданному ключу.
     *
     * @param key Ключ.
     * @param value Значение.
     */
    void put(K key, V value);

    /**
     * Читает значение из хранилища по заданному ключу.
     *
     * @param key Ключ.
     * @return Значение либо null.
     */
    @Nullable V get(K key);

    /**
     * Кладёт все пары ключ-значение в хранилище.
     *
     * @param entries Набор пар ключ-значение.
     */
    void putAll(Map<K, V> entries);

    /**
     * Читает все значения из хранилища по заданным
     * ключам.
     *
     * @param keys Набор ключей.
     * @return Пары ключ-значение.
     */
    Map<K, V> getAll(Collection<K> keys);

    /**
     * Читает из хранилища все значения, удовлетворяющие
     * заданному условию (предикату).
     *
     * @param p Предикат для проверки значений.
     * @return Значения, удовлетворяющие предикату.
     */
    Collection<V> getAll(Predicate<V> p);

    ... // и так далее
}
```

► [Определение Predicate](#)

Интерфейс выглядит вполне адекватно и логично, пользователь без проблем может написать простой код для работы с хранилищем

```
MyObjectStore<Long, Car> carsStore = ...;

carsStore.put(20334L, new Car("BMW", "X5", 2013));

Car c = carsStore.get(222L);

...
```

Однако, в чуть менее тривиальных случаях клиент вашего API столкнётся с неприятными ограничениями.

Использование ? super T

Возьмём последний метод, который читает значения, удовлетворяющие предикату. Что с ним может быть не так? Берём, да и пишем

```
Collection<Car> cars = carsStore.getAll(new Predicate<Car>() {
    @Override public boolean apply(Car exp) {
        ... // Здесь наша логика по выбору автомобиля.
    }
});
```

Но дело в том, что у нашего клиента уже есть предикат для выбора автомобилей. Только он параметризован не классом Car, а классом Vehicle, от которого Car унаследован. Он может попытаться записать Predicate вместо Predicate, но в ответ получит ошибку компиляции:

```
no suitable method found for getAll(Predicate<Vehicle>)
```

Компилятор говорит нам, что вызов метода невалиден, поскольку Vehicle – это не Car. Но ведь он является родительским типом Car, значит, всё, что можно сделать с Vehicle, можно сделать и с Car! Так что мы вполне могли бы использовать предикат по Vehicle для выбора значений типа Car. Просто мы не сказали компилятору об этом, и, тем самым, заставляем пользователя городить конструкции вроде:

```
final Predicate<Vehicle> vp = mgr.getVehiclePredicate();

Collection<Car> cars = carsStore.getAll(new Predicate<Car>() {
    @Override public boolean apply(Car exp) {
        return vp.apply(exp);
    }
});
```

А ведь всё решается так просто! Нам нужно лишь слегка изменить сигнатуру метода:

```
Collection<V> getAll(Predicate<? super V> p);
```

Запись Predicate<? super V> означает "предикат от V или любого супертипа V (вплоть до Object)". Данное изменение никак не ломает компиляцию существующего кода, зато устраняет абсолютно бессмысленные ограничения на параметр предиката. Клиент теперь может использовать свой предикат для Vehicle совершенно свободно:

```
MyObjectStore<Long, Car> carsStore = ...;

Predicate<Vehicle> vp = mgr.getVehiclePredicate();

Collection<Car> cars = carsStore.getAll(vp);
```

Мы обобщим данный приём чуть ниже, и запомнить его будет совсем просто.

Использование ? extends T

С передаваемыми коллекциями та же история, только в обратную сторону. Здесь, в большинстве случаев, имеет смысл использовать `extends T` для типа элементов коллекции. Посудите сами: имея ссылку на `MyObjectStore<Long, Vehicle>`, пользователь вполне вправе положить в хранилище набор объектов `Map<Long, Car>` (ведь `Car` – это подтип `Vehicle`), но текущая сигнатура метода не позволяет это сделать:

```
MyObjectStore<Long, Vehicle> carsStore = ...;

Map<Long, Car> cars = new HashMap<Long, Car>(2);

cars.put(1L, new Car("Audi", "A6", 2011));
cars.put(2L, new Car("Honda", "Civic", 2012));

carsStore.putAll(cars); // Ошибка компиляции.
```

Чтобы снять это бессмысленное ограничение, мы, как и в предыдущем примере, расширяем сигнатуру нашего интерфейсного метода, используя wildcard `? extends T` для типа элемента коллекции:

```
void putAll(Map<? extends K, ? extends V> entries);
```

Запись `Map<? extends K, ? extends V>` буквально означает "мапка с ключами типа `K` или любого из подтипов `K` и со значениями типа или любого из подтипов `V`".

Принцип PECS - Producer Extends Consumer Super

Настало время вывести общий принцип, благодаря которому мы всегда будем писать интерфейсы, абсолютно безопасные с точки зрения типов, но при этом не имеющие бессмысленных и создающих неудобства ограничений.

Этот принцип Joshua Bloch называет **PECS (Producer Extends Consumer Super)**, а авторы книги *Java Generics and Collections* (Maurice Naftalin, Philip Wadler) – **Get and Put Principle**. Но давайте остановимся на PECS, запомнить проще. Этот принцип гласит:

Если метод имеет аргументы с параметризованным типом (например, `Collection` или `Predicate`), то в случае, если аргумент – *производитель* (*producer*), нужно использовать `? extends T`, а если аргумент – *потребитель* (*consumer*), нужно использовать `? super T`.

Производитель и *потребитель*, кто это такие? Очень просто: если метод читает данные из аргумента, то этот аргумент – *производитель*, а если метод передаёт данные в аргумент, то аргумент является *потребителем*. Важно заметить, что определяя *производителя* или *потребителя*, мы рассматриваем только данные типа `T`.

В нашем примере `Predicate` – это *потребитель* (метод `getAll(Predicate)` передаёт в этот аргумент данные типа `T`), а `Map<K, V>` – *производитель* (метод `putAll(Map<K, V>)` читает данные типа `T` – в данном случае под `T` подразумевается `K` и `V` – из этого аргумента).

В случае, если аргумент является и *потребителем*, и *производителем* одновременно – например, если метод одновременно и читает из коллекции, и пишет в неё (плохой стиль, но всякое бывает) – тогда его нужно оставить как есть.

С возвращаемыми значениями тоже ничего делать не нужно – никакого удобства использование wildcard-ов в этом случае пользователю не принесёт, а лишь вынудит его использовать wildcard-ы в собственном коде.

Вооружившись PECS-принципом, мы можем теперь пройтись по всем методам нашего `MyObjectStore` интерфейса и сделать улучшения там, где это требуется. Методы `put(K, V)` и `get(K)` улучшений не требуют (т.к. они не имеют аргументов с параметризованным типом); методы `putAll(Map<? extends K, ? extends V>)` и `getAll(Predicate<? super V>)` мы уже и так улучшили, дальше некуда; вот метод

```
getAll(Collection) имеет аргумент-производитель с параметризованным типом, который мы можем расширить. Вместо
```

```
Map<K, V> getAll(Collection<K> keys);
```

делаем

```
Map<K, V> getAll(Collection<? extends K> keys);
```

и радуемся новому, более удобному API! (Заметьте, возвращаемое значение мы не трогаем!)

Другие примеры потребителя и производителя

Производителями могут быть не только коллекции. Самый очевидный пример *производителя* – это фабрика:

```
interface Factory<T> {  
    /**  
     * Создаёт новый экземпляр объекта заданного типа.  
     *  
     * @param args Аргументы.  
     * @return Новый объект.  
     */  
    T create(Object... args);  
}
```

Хорошим примером аргумента, являющегося и *производителем*, и *потребителем*, будет аргумент вот такого типа:

```
interface Cloner<T> {  
    /**  
     * Клонировет объект.  
     *  
     * @param obj Исходный объект.  
     * @return Копия.  
     */  
    T clone(T obj);  
}
```

Коллекция может быть *потребителем* в случае, если это output-коллекция, в которую метод складывает результат своей работы (х такой стиль в Java редко используется и считается плохим тоном).

Заключение

В этой статье мы познакомились с принципом PECS (Producer Extends Consumer Super) и научились его применять при разработке API на Java. Как показывает практика, даже в самых продвинутых программистских конторах об этом принципе некоторые разработчики не знают, и в результате проектируют не совсем удобное API. Но, к счастью, исправляются подобные ошибки очень легко, а запомнив мнемонику PECS однажды, вы уже просто не сможете не пользоваться ей в дальнейшем.

Литература

1. Joshua Bloch – Effective Java (2nd Edition)
2. Maurice Naftalin, Philip Wadler – Java Generics and Collections

Метки: [java](#), [generics](#), [интерфейсы](#), [api](#)

↑ +33 ↓ 231 54,9k 5



17,0

Карма

0,0

Рейтинг

6

Подписчики

Виктор Исаев @weekens

Пользователь

Поделиться публикацией



ПОХОЖИЕ ПУБЛИКАЦИИ

31 мая 2017 в 21:34

Странности Generic типов Java

↑ +21 👁 18,7k 📖 69 💬 24

17 августа 2016 в 16:02

Java Stream API: что делает хорошо, а что не очень

↑ +56 👁 19,8k 📖 77 💬 8

19 мая 2016 в 13:18

Generics в Kotlin vs. Generics в JAVA: сходства, различия, особенности

↑ +16 👁 17,8k 📖 87 💬 7

ЗАКАЗЫ ДЛЯ ФРИЛАНСЕРОВ**Фриланс**[Доработать интернет магазин доставки еды Edashop](#)

50000 руб./за пр

15.02.2018 • 0 откликов

[Требуется дизайн для сайта о исторической-реконструкции](#)

Цена догово

15.02.2018 • 4 отклика

[Интеграция ESD на сайте Shop Script 7](#)

15000 руб./за пр

15.02.2018 • 0 откликов

[Все заказы](#)[Зарегистрироваться](#)[Разместить з](#)

Карьера Android-разработчика начинается здесь

Образовательная программа,
разработанная профессионалами

e-legion [Учиться онлайн](#)

Реклама

Комментарии 5 **burdakovd** 05.01.14 в 16:23 # 📖

Когда начал писать на Java — и узнал что есть конструкции ? extends T и ? super T — сразу попытался представить ситуации, где они могут быть полезны — и как-то естественно пришёл к таким интерфейсам.

Я бы сформулировал вместо PECS другой принцип (сокрытия реализации.) — «знай только необходимый минимум о типах передаваемых параметров» — отсюда вытечет как использование extends так и super.

Colwin 13.01.14 в 11:45 # 📖 🔄

Не для всех это очевидно, к сожалению. :-)

У разных людей по-разному развита склонность к абстрактному мышлению.

И если человек специально не подумает об этом, то может и не прийти к аналогичным выводам.

grossws 13.01.14 в 13:53 # 📖 🔄

Ага, вот пример, на который я нарвался в apache commons-configuration: issues.apache.org/jira/browse/CONFIGURATION-561.

Товарищи удачно мигрировали на Java 1.5+ (до этого была совместимость 1.4+, где generics ещё не было).

▶ [проблемный код](#)

Сейчас это, конечно, исправлено.

 Skyggedans 05.01.14 в 18:45  

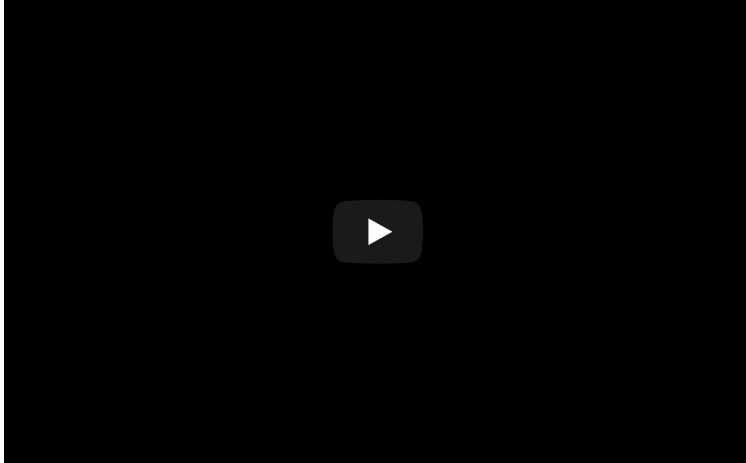


Было бы неплохо в контексте статьи упомянуть такой термин, как "**вариантность**". Например, что параметр типа в конструкции `<T>` является **инвариантным**, в `<? extends T>` — **ковариантным**, а в `<? super T>` — **контравариантным**. А то много разработчиков слышали эти термины, но что именно они описывают — без понятия, хотя каждый знаком с их сутью. К тому же, во всяких блогах по ООП они, почему-то, в подавляющем большинстве объясняются на примерах из C#.

 23derevo 06.01.14 в 02:53  



Вот тут Миша Ершов хорошо рассказал про совместимые API, в том числе, и про дженерификацию:



Только [полноправные пользователи](#) могут оставлять комментарии. [Войдите](#), пожалуйста.

ИНТЕРЕСНЫЕ ПУБЛИКАЦИИ





Яндекс.Алгоритм 2018: оптимизационный трек и ML-задача от разработчиков Алисы

 +22  951  8  0

Умный дом на ESP. Часть I GT

 +5  1,6k  21  31

ТОП-10. Разбор лучших докладов в свободном доступе. Heisenbug 2017 Moscow

 +12  1k  18  0

Avito Data Science Meetup: Personalization

 +12  718  2  0

ЦНИИмаш запатентовал лазерную систему опознавания «свой-чужой» для спутников GT

 +7  1,9k  2  6

Аккаунт	Разделы	Информация	Услуги	Приложения
Войти	Публикации	О сайте	Реклама	 
Регистрация	Хабы	Правила	Тарифы	
	Компании	Помощь	Контент	
	Пользователи	Соглашение	Семинары	
	Песочница	Конфиденциальность		

