

12. «Наводя мосты между дисциплинами»

Информатика, ИТИС, 1 курс 2 семестр

М.М.Абрамский

2017

КУСОЧЕК ПРОШЛОЙ ПАРЫ

Не буду загадывать загадки

- Всем очевидно, что оба типа плохих парней – наследники общего подкласса

```
abstract public class EvilGuy {  
    abstract public void attack(Player p);  
}
```

```
public class EvilElf extends EvilGuy {  
    public void attack(Player p) { ... }  
}
```

```
public class EvilDwarf extends EvilGuy {  
    public void attack(Player p) { ... }  
}
```

Сражение в коде

```
public void gameProcess() {  
  
    //...  
    while (battleIsGoing) {  
        badGuys.add(new EvilElf());  
        badGuys.add(new EvilDwarf());  
    }  
    //...  
}
```

Сражение в коде

```
public void gameProcess () {  
  
    //...  
    while (battleIsGoing) {  
        badGuys.add(new EvilElf());  
        badGuys.add(new EvilDwarf());  
    }  
    //...  
}
```

Если будет новый злодей – придется лезть в код Game и менять его

Чем он провинился? Его и так все время меняют!

Враги все равно для него все одинаковые.

Что нарушается?

Factory Method (фабричный метод)

Метод, который создает экземпляры класса,
при этом реализация определяется в
подклассах

А не наоборот!

Решение. Новые классы

```
public interface EvilGuyCreator {  
    EvilGuy getNew();  
}
```

```
public class EvilElfCreator implements EvilGuyCreator {  
    public EvilGuy getNew() {  
        return new EvilElf();  
    }  
}
```

```
public class EvilDwarfCreator implements EvilGuyCreator {  
    public EvilGuy getNew() {  
        return new EvilDwarf();  
    }  
}
```

Решение

```
public void gameProcess() {  
  
    //...  
    EvilGuyCreator [] creators = {new EvilElfCreator(),  
                                   new EvilDwarfCreator()};  
    while (battleIsGoing) {  
        for (EvilGuyCreator creator: creators)  
            badGuys.add(creator.getNew());  
    }  
    //...  
}
```

Если будет новый враг – нужно лишь добавить новый Creator.
(логично - сам виноват, наплодил наследников, еще и плохих парней)

Решение #2. Иногда делают так

```
abstract public class EvilGuy {  
    abstract public void attack(Player p);  
    public static EvilGuy getNext() {  
        String who = generate();  
        if (who.equals("elf")) {  
            return new EvilElf();  
        } else if (who.equals("dwarf")) {  
            return new EvilDwarf();  
        } else {  
            return null;  
        }  
    }  
    public static String generate() {  
        // генерируем  
    }  
}
```

Решение #2

```
public void gameProcess() {  
  
    //...  
    while (battleIsGoing) {  
        badGuys.add(EvilGuy.getNext());  
    }  
}
```

Если будет новый враг – идем менять EvilGuy
(логично - сам виноват, наплодил наследников, еще и плохих парней)

**ВЕРНЕМСЯ ОТ КОНЦЕПЦИЙ И
ПАРАДИГМ К РЕАЛЬНЫМ ДАННЫМ**

Начало начал

- У нас есть программа/алгоритм/код
- Она получает на вход одни данные и должна выдать на вход другие.
- Не всегда получается получать входные данные «на лету»
 - Поэтому нам нужны переменные – для хранения промежуточных данных в течение работы программы.

В чем храним

- Примитивные типы
- Массивы
 - Строки тут же
- Объекты
 - Наборы разнотипных данных, строки

- Вход – чаще всего однотипные данные неизвестного заранее размера
- Какой тип данных может их обрабатывать?

Только массив

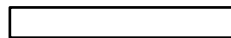
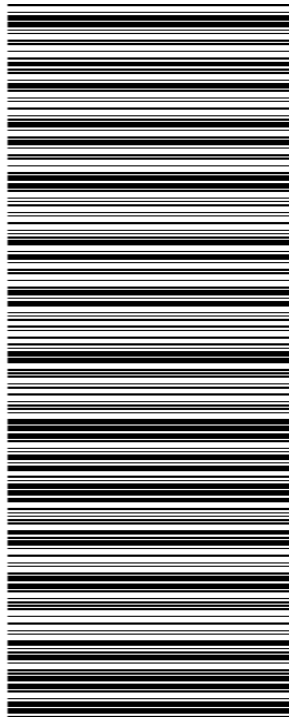
- Но все ли хорошо с ним?

Идеальная оперативная память



Реальная оперативная память

Имеет место
фрагментированность
оперативной памяти.



*Массив такого размера
уже не уместится*

Фиксированный размер

- Будучи объявленным один раз, массив не может менять свой размер:
 - `int [] arr = new int[100];`
 - Что заставляет разделять понятия «вместимости» и «реального размера» массива
 - Но и это не помогает

Нет, фактически можно увеличить размер массива

```
int [] arr = new int[100];
for (int i = 0; i < arr.length; i++) {
    arr[i] = (new Random()).nextInt();
}
// хотим еще один такой же элемент
int [] arr2 = new int[arr.length + 1];
for (int i = 0; i < arr.length; i++) {
    arr2[i] = arr[i];
}
arr2[arr.length] = (new Random()).nextInt();
}
arr = arr2;
```

Ничего так элемент добавили...

Вставка и удаление элемента

- Без комментариев.
 - Попробуйте в фотоальбом с напечатанными фотографиями вставить фотографию в начало, сохранив порядок.

Отсутствие инкапсуляции

- **Массив** – ссылочный тип, но он не объектно-ориентирован.
 - Нет методов, все операции с массивами приходится заново реализовывать.
 - При этом нужно писать одни и те же методы для разных типов, т.к., например, массивы `int []` и `char []` – разные типы.

Итак

В массиве есть свои плюсы, но минусы заставляют нас искать альтернативы

- Как на уровне интерфейса
 - Нет нужных методов, приходится постоянно писать одно и то же, интерфейс не удовлетворяет задаче
- Так и на уровне реализации
 - Фрагментация, блочное хранение массива

ЧТО ВООБЩЕ ТАКОЕ НАБОР ЭЛЕМЕНТОВ?

Приведите примеры из жизни

**ВСЕ ЛИ ЭТИ НАЗВАННЫЕ «НАБОРЫ»
ИМЕЮТ ОДИНАКОВЫЙ
ИНТЕРФЕЙС?**

Не-а!

Мешок яблок, пакет с покупками, библиотека, парковка, список студентов, фотографии в фотоальбоме, предметы на вашем столе, иконки на вашем рабочем столе, хранилище договоров, группа нападающих врагов, патроны в обойме, очередь в KFC.

Внимательно подумайте, что есть общего, в чем есть разница?

На что обратить внимание?

- Разрешены ли дубликаты или нет?
- Есть понятие позиции / нет?
- Сравнимы ли элементы друг с другом или нет?
 - Не требует понятия позиции
- Упорядоченное / неупорядоченное? Имеет ли это вообще смысл?
 - Требует сравнимости и позиции
- Разрешено ли добавление на произвольное место? Или не конкретизируется, куда добавлять?
 - Требует понятие позиции
- ...

Мешок

- **Дубликаты:** зависит от задачи
 - мешок подарков / мешок яблок
- **Нет понятия позиции.**
- **Сравнимость:** зависит от задачи
 - На будущее: одни и те же объекты можно сравнивать разными способами
- **Неупорядоченное**
- **Не конкретизируется, куда добавлять?**

Фотографии в фотоальбоме

- **Дубликаты:** нет
- **Есть** понятие позиции.
- **Сравнимость:** по годам
- **Упорядоченное.**
- **Можно** добавлять на произвольное место

Очередь в KFC

- **Дубликаты:** нет (если только вы не агент Смит)
- **Есть** понятие позиции.
- **Сравнимость:** нет
- **Неупорядоченное**
 - Нет сравнимости
- **Нельзя** добавлять на произвольное место
 - Если только вы не бессовестный, невоспитанный хам.

Список студентов

- **Дубликаты:** нет
- **Есть** понятие позиции.
- **Сравнимость:** есть
 - Алфавитный порядок фамилий
- **Упорядоченность:** имеет смысл
- **Можно** добавлять на произвольное место
 - Но с другой стороны, это нарушает упорядоченность
 - Если упорядоченность требование – то при добавлении не указывается позиция, а элемент сам встанет куда надо.

- Если «набор объектов» – это класс, то на что влияют эти разные случаи?

На интерфейс

- Метод `add(x)` имеет смысл
 - И для мешка – просто добавление
 - И для альбома с фотографиями – добавление в конец
 - » Что может считаться просто добавлением
- Метод `add(i, x)` – вставка на позицию
 - Для альбома с фотографиями имеет смысл
 - Для мешка – бессмысленный (нет позиции)

Значит

- Можно сгруппировать набор методов, получив интерфейс «набора данных», имеющего смысл для конкретной задачи.
 - Интерфейс, не реализацию – она не важна. Ведь вставка фотографии в фотоальбом и вставка договора в хранилище договоров – одна и та же операция.

Этот интерфейс и называется

**АБСТРАКТНЫМ
ТИПОМ
ДАННЫХ
(АТД)**

АТД

Проходятся вами на АиСД.

Пройдем вместе с коллекциями.

Но можно их перечислить:

- *Список, стек, очередь, множество, коллекция, ассоциативный массив (отображение), очередь с приоритетом и др.*

А что с реализацией?

- За интерфейсом должно прятаться реальное хранилище объектов
- А пока единственное хранилище, которое мы знаем — ЭТО массив.
 - У которого свои проблемы
 - Фрагментация
 - Неуправляемость размером
 - $O(n)$ добавление и удаление.

Причина проблем

Прямая адресация: необходимость хранить все элементы массива единым блоком, чтобы вычислять адрес i -го элемента за $O(1)$

$$\text{Адрес}(i) = \text{Адрес}(0) + i * \text{size}$$

Отказ от блоков

- Нужно альтернативное средство связывания элементов друг с другом.

ЖЗН

- Вы приходите на медосмотр и видите расписание врачей и кабинетов

601 Невролог

610 Отоларинголог

623 Окулист

644 Терапевт

675 Хирург

Вам сказали начать с отоларинголога.

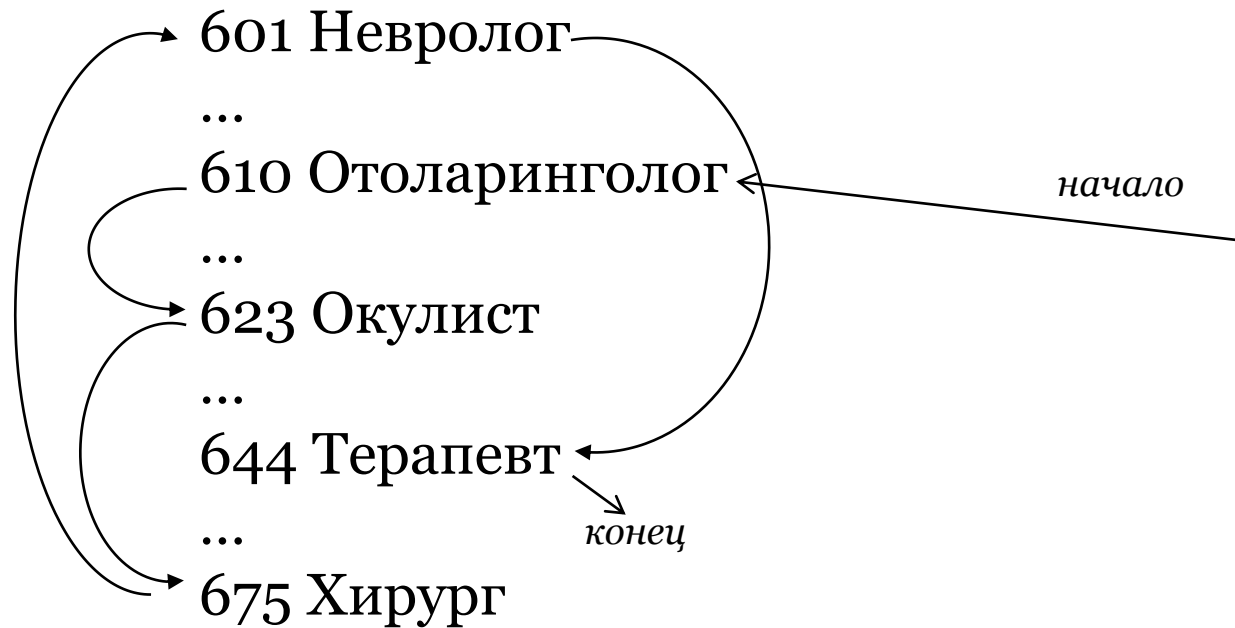
«По кабинетам»

Вы пришли в 610

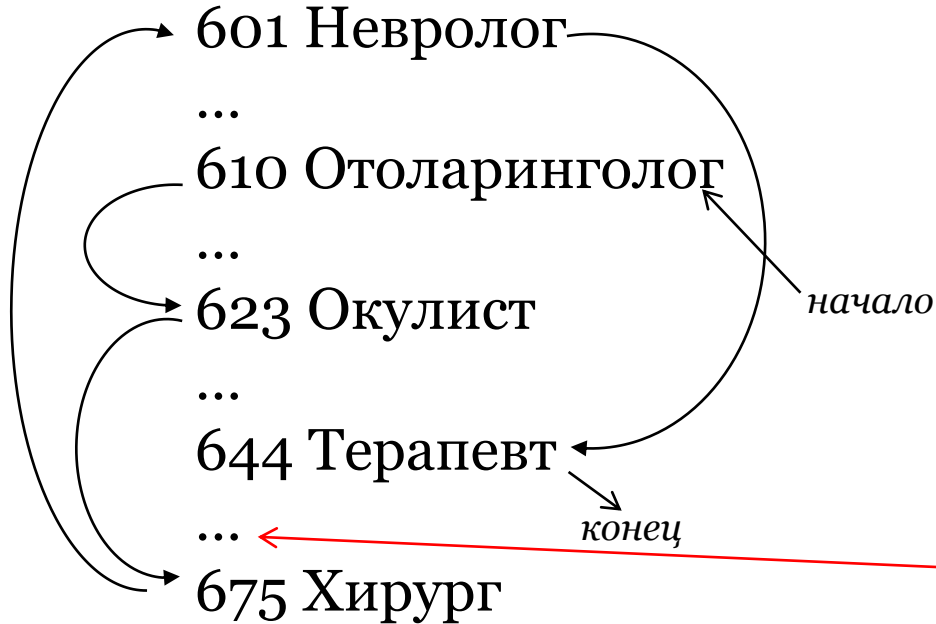
- Отоларинголог: Ок, теперь в 623
- Окулист: Ок, теперь в 675
- Хирург: Ок, теперь в 601
- Невролог: Ок, теперь к терапевту, в 644
- Терапевт: Медосмотр пройден

По факту

- Есть последовательность врачей, которая задается не их порядковым номером, а понятием «следующий» для каждого элемента.



В этом примере модель ОП



- Номера кабинетов – ячейки ОП
- В каждой «ячейке» хранятся
 - Значение (врач)
 - Ссылка/адрес/указатель на следующий элемент
 - На рисунке стрелка, а по факту – переменная-ссылка
- Т.к. связность обеспечивается с помощью понятия «следующий», фрагментация больше не проблема
- Если «следующий элемент – ссылка», то конец – это что?

Список

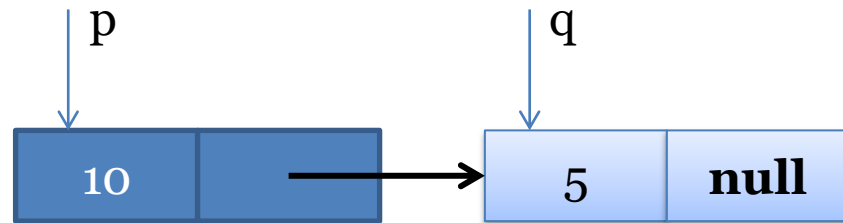
- Список элементов
- Каждый элемент — не одно, а два значения
 - Само значение (в примере — врач)
 - Ссылка на следующий элемент.
- Что в java умеет хранить переменные разных типов?

Elem (Node)

```
public class Elem {  
    int value; // пусть  
    Elem next;  
}
```

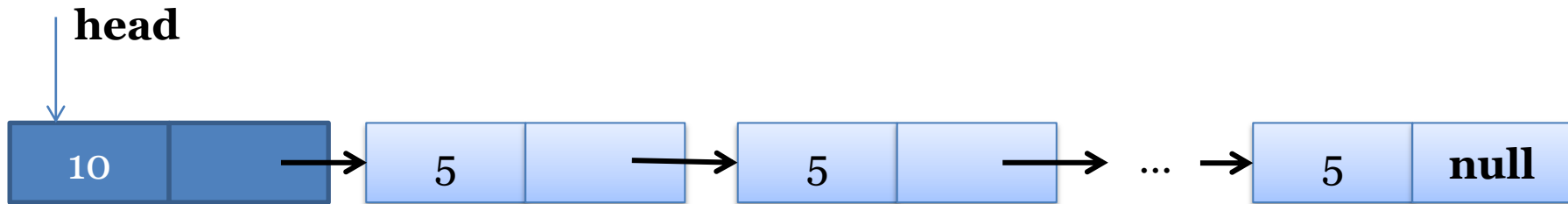
...

```
Elem q = new Elem();  
q.value = 5;  
Elem p = new Elem();  
p.value = 10;  
q.next = null;  
p.next = q;
```



*Следующий элемент –
того же типа, что этот
элемент (логично, чо)*

Односвязный линейный список



Голова – главное в списке.

Если ссылка на head потеряется – то все, списка нет.

Сложность доступа?

Сложность добавления?

Сложность удаления?

Класс

```
public class List {  
    private Elem head;  
  
    public void add(int value) {  
        Elem p = new Elem();  
        p.setValue(value);  
        p.setNext(head);  
        head = p;  
    }  
  
    public void add(T value, int position) { ... }  
  
}
```

Связность может быть разной

- Если ссылок на следующий элемент не одна, а две, то это уже — дерево.
- Но фактически, есть только два способа организовать набор данных — массив и связный список.
 - И их комбинации — список массивов, массив списков.

Структура данных

- Набор данных с конкретным способом их организации
 - Т.е. в отличие от АТД, важно, как именно реализовано.

Структуры данных

- Список, основанный на массиве
- Хэш-таблица
- Список, основанный на связном списке
- Кольцевой список
- Двухнаправленный список
- Бинарное дерево (и его вариации)
- Граф
- И др.

АТД и СД

- СД может реализовывать (и почти всегда реализует какой-либо) АТД
- Пример:
 - Множество можно реализовать с помощью хэш-таблицы, можно с помощью дерева
 - Стек можно реализовать через массив, можно через связный список
- Опять же, будем все это подробнее разбирать на коллекциях.

ΠΑΡΑ ΜΟΜΕΝΤΟΒ

Оболочки примитивных типов

- Для каждого примитива есть соответствующий класс-оболочка
 - Такое же число, только объект, а не примитив
 - Название то же, только с большой буквы, кроме Integer и Character (для int и char)
- Автоупаковка:

```
Integer i = new Integer(5);  
Integer i = 5;  
int j = i + 1;
```

Number

- 6 из 8 классов оболочек наследуются от Number
 - У них есть общие методы toIntValue(), toDoubleValue(), toFloatValue() и др., которые реализованы в каждом классе-оболочке.

- Под List понимаем связный список
выше
 - Т.к. официально пока не прошли коллекции

GENERIC

Исходная проблема

- Есть задачи, в которых параметром являются не только значения, но и тип.
 - Чем отличается сортировка массива `char` от массива строк?
 - Чем отличается задача печати списка студентов в файл и списка
 - И вообще, сортировка строк от сортировки файлов по имени?
- Большинство таких задач - задачи на объектах, которые являются контейнерами объектов других классов.

Не решается известными методами

Код все равно приходится писать разный, т.к. эти типы не связаны наследованием/полиморфизмом.

Если бы это были числа, то можно было бы написав сортировку на Number, применить ее к Integer и Double

- Кстати да, восходящее преобразование работает на массивах:
 - Integer [] можно поднять до Number[]
 - *Да, это не всегда возможно.*
- *Но что если алгоритм можно применить к любым типам*
 - *если это алгоритм на наборах данных, например.*

Решение - Generics

- Начиная с Java 5.
- Возможность объявлять классы и интерфейсы, указывая используемый в них тип «содержимого» в качестве параметра.

Обновленный Elem

```
class Elem<T> {  
    private T value;  
    private Elem next;  
    public Elem() {}  
    public Elem(T value, Elem next) {  
        this.value = value;  
        this.next = next;  
    }  
    public T getValue() { return value; }  
    public void setValue(T value) { this.value = value; }  
    public Elem getNext() { return next; }  
    public void setNext(Elem next) { this.next = next; }  
}
```

Обновленный List

```
public class CoolList<T> {  
    private Elem<T> head;  
  
    public void add(T value) {  
        Elem<T> p = new Elem<T>();  
        p.setValue(value);  
        p.setNext(head);  
        head = p;  
    }  
}
```

Использование

```
public static void main(String[] args) {  
  
    CoolList<String> lst = new CoolList<String>();  
    lst.add(123); // не работает, т.к. принимает String  
  
    CoolList<File> lst2 = new CoolList<File>();  
    CoolList<CoolList<Integer>> lst3 =  
        new CoolList<CoolList<Integer>>();  
  
    // но вообще говоря с java 7 можно опускать тип справа  
    // (diamond-operator)  
  
    CoolList<String> lst4 = new CoolList<>();  
}
```

Если забыть тип при объявлении

- То считается, что это Object

```
CoolList lst5 = new CoolList();  
lst5.add(123);  
lst5.add(new Object());
```

Горька правда про generics

- Их нет.

Горька правда про generics

- Их нет.
- Нет, они есть, но только на этапе КОМПИЛЯЦИИ.
 - При запуске Java все типы параметров превращаются в Object.
 - Зачем?

Обратная совместимость

Generics появились в Java 5. Нужно было, чтобы старые приложения поддерживались.

- И старые коллекции тоже.
- А значит, тип коллекций должен быть Object всегда.
 - Ну или хотя бы при запуске.
- Поэтому после компиляции, где проверяется корректность типов, типы **стираются** – и это означает, что в runtime все динамическое, что можно делать с объектами, нельзя делать с параметризованными типами.

Ограничения

- `new T()`, `new T[]` – нельзя
- Нельзя делать `instanceof` для параметра.
- Нельзя делать `static` поле типа `T`
- Нельзя перегрузить методы двумя классами с разными параметрами
 - `void print(CoolList<String> c)` и `void print(CoolList<Integer> c)` – нельзя
- В коллекциях должны быть объекты – нельзя примитивы
 - Поэтому и были придуманы оболочки типов.

«А чё можно-то?»

- Можно ограничить тип

```
public class CoolNumberList<T extends Number> {  
    Elem<T> head;  
    public void add(T value) {  
        ...  
    }  
    public double sum() {  
        double s = 0;  
        for (Elem<T> p = head; p != null; p = p.getNext()) {  
            s += p.getValue().doubleValue();  
        }  
        return s;  
    }  
}
```

Можем пользоваться
интерфейсом Number.

Кстати, для интерфейсов
также extends

Использование

```
public static void main(String[] args) {  
    CoolNumberList<Integer> intlist = new CoolNumberList<>();  
    CoolNumberList<Double> doublelist = new CoolNumberList<>();  
CoolNumberList<String> stringlist = new CoolNumberList<>();  
}
```


Параметрический полиморфизм

- Пользуясь иерархией наследование типа параметра, можно реализовывать параметрический полиморфизм.
 - Один и тот же класс контейнер – List
 - Типы параметров A и B – наследники одного общего класса (с разной реализацией некоторых методов)
 - Значит List<A> и List - могут иметь разное поведение.

Неизвестный тип

Вам нужно объявить метод, принимающий параметризованный класс, но параметр вам неизвестен.

```
public class Printer {  
    public void print(List<...> elements) {  
        //...  
    }  
}
```

- 
- List<T> нельзя, т.к. неизвестно, что такое T
 - List – явная заточенность под Object – а если есть специфика?
 - ?

<?> - Wildcard (неизвестный тип)

```
public class Printer {  
    public void print(List<?> elements) {  
        //...  
    }  
}
```


Поразмышляем

- Есть два типа, один унаследован от другого
 - Объект – Строка
 - Человек – Студент
 - Оружие - Меч
- Можно ли попробовать описать наборы объектов

Ковариантность

- Список из студентов – частный случай списка из людей
 - T ковариантен `List<T>`

Контравариантность

Возьмем пример из нашей RPG.

- Action<T> - действие с оружием T.
- Все действия, которые персонаж может сделать с экземплярами класса «Оружие», он может сделать и с классом «Меч» (но не наоборот).

Контравариантность

- Получается, что как будто интерфейс `Action<Оружие>` шире, чем `Action<Меч>`.
- Что означает, что как будто бы `Action<Оружие>` наследуется от `Action<Меч>`.
 - Меч наследуется от Оружия.
 - Но `Action<Оружие>` наследуется от `Action<Меч>`.
- `T` контрвариантен `Action<T>`

Обе реализованы в wildcard

Пример с <http://www.angelikalanger.com/GenericsFAQ/FAQSections/TypeArguments.html#FAQ103>

```
public class NumberCollections {  
    public static void copy(List<? super Number> dest,  
        List<? extends Number> src) {  
        for (int i = 0; i < src.size(); i++)  
            dest.set(i, src.get(i));  
    }  
}
```

Список чисел копируется из src в dest.

- **src** – должен содержать числа, поэтому extends Number – **ковариантность**.
- **dest** - принимает числа (но может быть и более общего типа – поэтому **контравариантность**).

Еще можно

- Создавать классы с несколькими параметрами типов:
 - Пример в будущем: `Map<K, V>`
- Создавать параметризованные методы
 - Для работы с параметризованными коллекциями без указания их типа. Оригинал примера копирования:

```
public static <T> void copy(List<? super T> dest, List<? extends T> src)
```

Ключевые слова

Diamond operator, generics, wildcard, Абстрактный тип данных, ковариантность, контравариантность, настраиваемый тип, неизвестный тип, обобщенный тип, параметрический полиморфизм, ромбовидный оператор, ссылочная связь, стирание типов, структура данных, фрагментация памяти

Почитать

АТД, СД, массивы

- <https://tproger.ru/translations/linked-list-for-beginners/>
- <https://habrahabr.ru/post/216725/>
- <https://www.slideshare.net/grebenshikov/2-2260765>

Generics

- <http://www.angelikalanger.com/GenericsFAQ/FAQSections/TypeArguments.html#FAQ103>
- <http://www.quizful.net/post/java-generics-tutorial>
- <http://developer.alexanderklimov.ru/android/java/generic.php>
- https://www.tutorialspoint.com/java/java_generics.htm