

Effective Java

Юрий Литвинов
yurii.litvinov@gmail.com

20.02.2019г

О контрольной

- ▶ Ожидалось решение, использующее `Object` для хранения данных, разбор случаев и кучу `unchecked cast`-ов
 - ▶ Естественно, заглушенных с помощью *`//noinspection unchecked`*
 - ▶ Было одно решение, где использовалась иерархия классов-обёрток с одним полем, получилось лучше, чем ожидалось
- ▶ Надо было реализовать `size`, `get`, `set`, `add` и `remove` по индексу
 - ▶ Остальное реализовано в `AbstractList` достаточно эффективно
- ▶ `addAll` в конструкторе, принимающем коллекцию — нехорошо
- ▶ `System.arraycopy`
- ▶ Магические константы
- ▶ Беда с сырыми типами и `unchecked cast`-ами не по делу

О домашке

Заклинание, заставляющее красиво отобразить состояние тестов в CI:

```
test {  
    useJUnitPlatform()  
    testLogging {  
        events 'PASSED', 'FAILED', 'SKIPPED'  
    }  
}
```

Книжка, по которой рассказ

Joshua Bloch, Effective Java, 3rd Edition. Addison-Wesley Professional, 2017. 412PP.



49. Проверяем достоверность параметров

- ▶ Часто на аргументы методов накладываются ограничения
 - ▶ Нужно их документировать
 - ▶ `@throws`
 - ▶ А метод начинать с проверки и быстро завершать, иначе
 - ▶ Половина метода может выполняться, оставив некорректное состояние
 - ▶ Метод тихо вернёт некорректный результат
- ▶ Особенно если аргументы откладываются для обработки в дальнейшем
 - ▶ Частый случай — конструкторы
- ▶ `IllegalArgumentException`, `IndexOutOfBoundsException`, `NullPointerException`...
 - ▶ Или преобразование исключения в понятное пользователю
- ▶ `java.util.Objects.checkFromIndexSize`, `checkFromToIndex`, `checkIndex`

public-методы

```

/*
 * Возвращает объект BigInteger, значением которого является модуль
 * данного числа по основанию m. Этот метод отличается от метода
 * remainder тем, что всегда возвращает неотрицательное значение
 * BigInteger.
 *
 * @param m модуль, должен быть положительным числом
 * @return this mod m
 * @throws ArithmeticException, if m <= 0
 */
public BigInteger mod(BigInteger m) {
    if (m.signum() <= 0) {
        throw new ArithmeticException("Modulus not positive");
    }
    ... // Вычисления
}

```

private-методы

// Закрытая вспомогательная функция для рекурсивной сортировки

```
private static void sort(long a[], int offset, int length) {
```

```
    assert a != null;
```

```
    assert offset >= 0 && offset <= a.length;
```

```
    assert length >= 0 && length <= a.length - offset;
```

```
    // Вычисления
```

```
}
```

50. Резервные копии могут быть полезны

- ▶ JVM даёт уверенность в безопасности работы с памятью
 - ▶ В отличие от C++, инварианты класса будут выполняться, что бы ни происходило в остальной системе
- ▶ Однако всё равно надо оберегать свои инварианты!
 - ▶ Программировать надо так, будто все клиенты вашего класса делают (практически) всё возможное, чтобы нарушить его инварианты

// Неправильный класс «неизменяемого» периода времени

```
public final class Period {
```

```
    private final Date start;
```

```
    private final Date end;
```

```
    /**
```

```
        @param start - начало периода.
```

```
        @param end - конец периода; не должен предшествовать началу.
```

```
        @throws IllegalArgumentException, если start позже, чем end.
```

```
        @throws NullPointerException, если start или end равны null.
```

```
    */
```

```
    public Period(Date start, Date end) {
```

```
        if (start.compareTo(end) > 0)
```

```
            throw new IllegalArgumentException(start + " after " + end);
```

```
        this.start = start;
```

```
        this.end = end;
```

```
    }
```

```
    public Date start() { return start; }
```

```
    public Date end() { return end; }
```

```
    // Остальное опущено
```

```
}
```

// Атака на содержимое экземпляра Period

```
Date start = new Date();
```

```
Date end = new Date();
```

```
Period p = new Period(start, end);
```

```
end.setYear(78); // Изменяет содержимое объекта p!
```

Defensive copies

*// Исправленный конструктор: для представленных параметров
// создает резервные копии*

```
public Period(Date start, Date end) {  
    this.start = new Date(start.getTime());  
    this.end = new Date(end.getTime());  
    if (this.start.compareTo(this.end) > 0) {  
        throw new IllegalArgumentException(start + " after " + end);  
    }  
}
```

- ▶ Проверяем уже созданные копии, а не исходные параметры (ибо многопоточность)
- ▶ Не используем clone(), потому что его могут переопределить и подсунуть нам подкласс

Атака №2

```
Date start = new Date();  
Date end = new Date();  
Period p = new Period(start, end);  
p.end().setYear(78); // Изменяет внутренние данные p!
```

Атака №2

```
Date start = new Date();  
Date end = new Date();  
Period p = new Period(start, end);  
p.end().setYear(78); // Изменяет внутренние данные p!
```

Решение:

```
public Date start() {  
    return (Date) start.clone();  
}
```

```
public Date end() {  
    return (Date) end.clone();  
}
```

Мораль

- ▶ Для каждого аргумента сеттера или конструктора, который сохраняем
 - ▶ А является ли передаваемый объект изменяемым?
 - ▶ Будет ли наш объект устойчив к его изменениям?
 - ▶ Например, ключ в Map
 - ▶ Массивы ненулевой длины всегда изменяемы
- ▶ Или можно хранить что-то другое
 - ▶ Например, `Instant`, `LocalDateTime`, `ZonedDateTime` или даже `Date.getTime()` вместо `Date`
- ▶ Не забыть отразить всё это в документации к классу!
- ▶ А также помнить про производительность

51. Советы по проектированию сигнатур методов

- ▶ Тщательно выбирайте названия методов
 - ▶ С учётом всех соглашений и стайлгайдов
- ▶ Методов не должно быть слишком много
 - ▶ Одна операция — один метод
 - ▶ Не надо гнаться за удобством сразу, пусть время покажет
- ▶ Длинный перечень параметров — плохо
 - ▶ 4 — уже много (особенно если они похожих типов)
 - ▶ Что делать?
 - ▶ Разбить метод на несколько ортогональных (например, метод `subList` в `List`, возвращающий `view` на часть списка)
 - ▶ Создание вспомогательных классов, выделение параметров в сущность
 - ▶ Шаблон `Builder`

Ещё советы

- ▶ Интерфейсы вместо конкретных классов параметров
 - ▶ Более обобщённый код
- ▶ Двухэлементные перечисления вместо Boolean
 - ▶ `Thermometer.newInstance(true);`
 - ▶ `public enum TemperatureScale { FAHRENHEIT, CELSIUS }`
`Thermometer.newInstance(TemperatureScale.CELSIUS);`
 - ▶ Потом можно и Кельвины какие-нибудь добавить при желании
 - ▶ В enum можно добавить полезных методов

52. Перегружаем методы осторожно

// Что выведет данная программа?

```
public class CollectionClassifier {
    public static String classify(Set<?> s) { return "Set"; }
    public static String classify(List<?> l) { return "List"; }
    public static String classify(Collection<?> c) {
        return "Unknown Collection"; }

    public static void main(String[] args) {
        Collection<?>[] collections = {
            new HashSet<String>(),
            new ArrayList<BigInteger>(),
            new HashMap<String, String>().values()
        };
        for (Collection<?> c : collections)
            System.out.println(classify(c));
    }
}
```

52. Перегружаем методы осторожно

// Что выведет данная программа?

```
public class CollectionClassifier {
    public static String classify(Set<?> s) { return "Set"; }
    public static String classify(List<?> l) { return "List"; }
    public static String classify(Collection<?> c) {
        return "Unknown Collection"; }

    public static void main(String[] args) {
        Collection<?>[] collections = {
            new HashSet<String>(),
            new ArrayList<BigInteger>(),
            new HashMap<String, String>().values()
        };
        for (Collection<?> c : collections)
            System.out.println(classify(c));
    }
}
```

Выбор варианта
перегрузки
осуществляется
на стадии
компиляции!

А вот с переопределением всё иначе

```
class Wine {  
    String name() { return "wine"; }  
}  
class SparklingWine extends Wine {  
    @Override String name() { return "sparkling wine"; }  
}  
class Champagne extends SparklingWine {  
    @Override String name() { return "champagne"; }  
}  
public class Overriding {  
    public static void main(String[] args) {  
        Wine[] wines = { new Wine(), new SparklingWine(), new Champagne() };  
        for (Wine wine : wines) System.out.println(wine.name());  
    }  
}
```

Мораль

- ▶ Не следует писать код, непонятный среднему программисту
 - ▶ Особенно в рамках API
 - ▶ Стоит избегать запутанных вариантов перегрузки
 - ▶ Всегда можно назвать методы по-разному
- ▶ Одинаковое количество параметров перегруженных методов — плохо
 - ▶ Надо будет следить за приводимостью типов
 - ▶ Методы с `varargs` лучше не перегружать вообще
- ▶ А как же конструкторы?
 - ▶ Статические фабричные методы
- ▶ Особая боль — автоупаковщики

Ещё один пример

```
public class SetList {  
    public static void main(String[] args) {  
        Set<Integer> set = new TreeSet<Integer>();  
        List<Integer> list = new ArrayList<Integer>();  
        for (int i = -3; i < 3; i++) {  
            set.add(i);  
            list.add(i);  
        }  
        for (int i = 0; i < 3; i++) {  
            set.remove(i);  
            list.remove(i);  
        }  
        System.out.println(set + " " + list);  
    }  
}
```

Ещё один пример

```

public class SetList {
    public static void main(String[] args) {
        Set<Integer> set = new TreeSet<Integer>();
        List<Integer> list = new ArrayList<Integer>();
        for (int i = -3; i < 3; i++) {
            set.add(i);
            list.add(i);
        }
        for (int i = 0; i < 3; i++) {
            set.remove(i);
            list.remove(i);
        }
        System.out.println(set + " " + list);
    }
}

```

Вывод:

[-3, -2, -1] [-2, 0, 2]

Перегрузка – зло.

53. varargs — опасная штука

// Простое использование varargs

```
static int sum(int... args) {  
    int sum = 0;  
    for (int arg : args)  
        sum += arg;  
    return sum;  
}
```

53. varargs — опасная штука

// Простое использование varargs

```
static int sum(int... args) {  
    int sum = 0;  
    for (int arg : args)  
        sum += arg;  
    return sum;  
}
```

// Неверное использование varargs для передачи одного или более аргументов!

```
static int min(int... args) {  
    if (args.length == 0)  
        throw new IllegalArgumentException("Too few arguments");  
    int min = args[0];  
    for (int i = 1; i < args.length; i++)  
        if (args[i] < min)  
            min = args[i];  
    return min;  
}
```


Более правильный вариант

```
static int min(int firstArg, int... remainingArgs) {  
    int min = firstArg;  
    for (int arg : remainingArgs)  
        if (arg < min)  
            min = arg;  
    return min;  
}
```

varargs методы

- ▶ Метод с final массивом можно заменить на varargs метод
 - ▶ Но это не значит, что так надо делать

- ▶ Идиома печати массива (java 1.5-):

```
System.out.println(Arrays.asList(myArray));
```

- ▶ Но с примитивными типами всё равно получим не то:

```
public static void main(String[] args) {  
    int[] digits = { 3, 1, 4, 1, 5, 9, 2, 6, 5, 4 };  
    System.out.println(Arrays.asList(digits));  
}
```

// Правильный способ напечатать массив

```
System.out.println(Arrays.toString(myArray));
```

Производительность

- ▶ Каждый вызов приводит к размещению и инициализации массива
- ▶ Если критична производительность — делаем несколько перегрузок:

```
public void foo() {}  
public void foo(int a1) {}  
public void foo(int a1, int a2) {}  
public void foo(int a1, int a2, int a3) {}  
public void foo(int a1, int a2, int a3, int... rest) {}
```

54. Возвращаем массив нулевой длины вместо null

```
private List cheesesInStock = ...;
public List<Cheese> getCheeses() {
    return cheesesInStock.isEmpty()
        ? null
        : new ArrayList<>(cheesesInStock);
}
```

- ▶ Можно возвращать закэшированный пустой список
- ▶ И вообще, предварительная оптимизация это зло

```
List<Cheese> cheeses = shop.getCheeses();
if (cheeses != null && cheeses.contains(Cheese.STILTON)) {
    System.out.println("Jolly good, just the thing.");
}
```

Выгрузка элементов из коллекции в массив

// Правильный способ вывести массив из коллекции

```
private final List cheesesInStock = ...;
private static final Cheese[] EMPTY_CHEESE_ARRAY = new Cheese[0];
public Cheese[] getCheeses() {
    return (Cheese[]) cheesesInStock.toArray(EMPTY_CHEESE_ARRAY);
}
```

// Правильный способ возврата копии коллекции.

```
public List<Cheese> getCheeseList() {
    if (cheesesInStock.isEmpty())
        return Collections.emptyList(); // Always returns same list
    else
        return new ArrayList<Cheese>(cheesesInStock);
}
```

56. Для всех открытых элементов API пишите доки!

- ▶ А ещё стоит документировать и другие элементы для тех, кто будет это всё сопровождать
- ▶ Документация для метода должна лаконично описывать контракт между этим методом и его клиентами
 - ▶ Что делает, а не как
 - ▶ Исключения — методы в классах, предназначенных для наследования
 - ▶ Предусловия и постусловия
 - ▶ `@throws` для описания предусловий
 - ▶ Или в `@param`
 - ▶ Побочные эффекты
 - ▶ Потокобезопасность

```

/**
 * Возвращает элемент, который занимает заданную позицию в данном списке.
 *
 * <p>Этот метод <i>Не</i> дает гарантии, что будет выполняться в постоянное
 * время. В некоторых реализациях он будет выполняться во время,
 * пропорциональное положению его элементов.
 *
 * @param index индекс элемента, который нужно вернуть;
 * индекс должен быть меньше размера списка и неотрицательным
 * @return элемент, занимающий в списке указанную позицию
 * @throws IndexOutOfBoundsException, если индекс лежит вне диапазона
 * ({@code index < 0 || index >= this.size()})
 */
E get(int index);

```

Подробности

- ▶ Методы и конструкторы — глагольная конструкция
 - ▶ “Создаёт пустой список”, “Возвращает количество элементов”
- ▶ Классы, интерфейсы, поля — именная конструкция
 - ▶ “Значение, наиболее близкое к пи”, “Задача для однократного исполнения”
- ▶ Для генериков не забываем комментировать параметры типа
- ▶ Для перечислений документируем константы, тип и все открытые методы
 - ▶ Можно в одну строку, если комментарий краткий
- ▶ Не забываем документировать аннотации для аннотируемых типов
- ▶ Для пакетов создаём package-info.java
- ▶ Наследование комментариев

58. Проверяемые vs непроверяемые исключения

- ▶ Проверяемые исключения
 - ▶ Явное предупреждение клиентскому коду
 - ▶ Вызывающая сторона в состоянии обработать и восстановиться
 - ▶ Вспомогательные методы и данные в объекте исключения
- ▶ Непроверяемые исключения
 - ▶ `RuntimeException` — индикация программных ошибок
 - ▶ `Errors` – невозможность дальнейшего выполнения
- ▶ `Throwable`
 - ▶ Не рекомендуется наследовать явно

59. Избегайте ненужного использования проверяемых исключений

- ▶ Невозможно предотвратить условие для исключительной ситуации
- ▶ Программист может как-то полезно её обработать

```
} catch (TheCheckedException e) {  
    throw new AssertionError(); // Этого случиться не может!  
}
```

```
} catch (TheCheckedException e) {  
    // nothing to do  
}
```

Вариант решения: разделение на два метода

- ▶ Прямо как в Iterator
- ▶ Beware!
 - ▶ Многопоточность
 - ▶ Изменение состояния извне

```
// Вызов с проверяемым исключением  
try {  
    obj.action(args);  
} catch (TheCheckedException e) {  
    // Обработать исключительную ситуацию  
}
```

```
// Вызов с использованием метода проверки  
// состояния и непроверяемого исключения  
if (obj.actionPermitted(args)) {  
    obj.action(args);  
} else {  
    // Обработать исключительную ситуацию  
}
```

61. Выбрасывайте исключения, соответствующие абстракции

- ▶ Трансляция исключений
 - ▶ Перехватывайте исключения, бросайте дальше что-то более адекватное
 - ▶ Exception chaining

```
/**
 * Возвращает элемент, находящийся в указанной позиции в этом списке.
 * @throws IndexOutOfBoundsException, если индекс находится
 * за пределами диапазона (index < 0 || index >= size()).
 */
public E get(int index) {
    ListIterator<E> i = listIterator(index);
    try {
        return i.next();
    } catch (NoSuchElementException e) {
        throw new IndexOutOfBoundsException("Index: " + index);
    }
}
```

62. Документируйте все выбрасываемые исключения

- ▶ Объявляйте все проверяемые исключения при помощи **throws** и **@throws**
 - ▶ Не экономьте при помощи суперкласса (**throws** Throwable)
 - ▶ Однако бывают исключения в публичных интерфейсах
- ▶ Документируйте все непроверяемые исключения при помощи **@throws**
 - ▶ В сигнатуре метода не нужно
 - ▶ Не всегда получается это сделать в полной мере
- ▶ Документирование на уровне класса в целом
 - ▶ “Все методы выбрасывают NPE, если им передают аргументом null”

65. Не игнорируйте исключения

- ▶ Пустой блок **catch** лишает смысла механизм исключений
- ▶ Как минимум – комментарий, почему исключение игнорируется
- ▶ Логирование исключений
- ▶ Fail fast