26 г. Днепропетровск. Покулитій александр грегорьевичСправочник по java. Статические строки

Строки в джава описываются классом String и являютя статическими, т.е. в существующей строке нельзя изменить символы и их колличество.    
  
Кроме стандартного создания оператором new, строки могут быть созданы напрямую из строковой литералы. При этом, в целях оптимизации, объекты созданные таким образом дополнительно сохраняются в отдельной области — строковый пул.

String s1 = "d" // строка будет сохранена в пуле

// строка не будет сохранена в пуле

// будет уничтожена сборщиком мусора

String s2 = new String("a");

Один из плюсов разделения строк на статические и динамические — повышение безопасности там, где строки используются в качестве аргументов (например, открытие баз данных, интернет соединений, механизм загрузки классов). 

**Операция +**

Для строк доступна операция +, позволяющая соединить несколько строк в одну. Если один из операндов не строка, то он автоматически преобразуется в строку. Для объектов в этих целях используется метод toString.   
При каждой операции внутренне используется объект динамической строки StringBuilder или StringBuffer. Поэтому для собирания строки из нескольких все равно оптимальней использовать сразу один StringBuilder/StringBuffer. 

**Выделение подстроки**

Есть замечание относительно метода substring — возвращаемая строка использует тот же байтовый массив, что и исходная. Например, вы загрузили строку А из файла в 1мб. Что-то там нашли и выделили в отдельную строку Б длиной в 3 символа. Строка Б в реальности тоже занимает те же 1мб.

String s ="very .... long string from file";

// совместно использует ту же память что и s

String sub1 = s.substring(2,4);

// этот объект использует отдельный массив на 4 символа

String sub2 = new String(s.substring(2,4));

**Основные методы:**

* **compareTo**(String anotherString) — лексиграфическое сравнение строк;
* **compareToIgnoreCase**(String str) — лексиграфическое сравнение строк без учета регистра символов;
* **regionMatches**(boolean ignoreCase, int toffset, String other, int ooffset, int len) — тест на идентичность участков строк, можно указать учет регистра символов;
* **regionMatches**(int toffset, String other, int ooffset, int len) — тест на идентичность участков строк;
* **concat**(String str) — возвращает соединение двух строк;
* **contains**(CharSequence s) — проверяет, входит ли указанная последовательность символов в строку;
* **endsWith**(String suffix) — проверяет завершается ли строка указанным суффиксом;
* **startsWith**(String prefix) — проверяет, начинается ли строка с указанного префикса;
* **startsWith**(String prefix, int toffset) — проверяет, начинается ли строка в указанной позиции с указанного префикса;
* **equals**(Object anObject) — проверяет идентична ли строка указанному объекту;
* **getBytes**() — возвращает байтовое представление строки;
* **getChars**(int srcBegin, int srcEnd, char[] dst, int dstBegin) — возвращает символьное представление участка строки;
* **hashCode**() — хеш код строки;
* **indexOf**(int ch) — поиск первого вхождения символа в строке;
* **indexOf**(int ch, int fromIndex) — поиск первого вхождения символа в строке с указанной позиции;
* **indexOf**(String str) — поиск первого вхождения указанной подстроки;
* **indexOf**(String str, int fromIndex) — поиск первого вхождения указанной подстроки с указанной позиции;
* **lastIndexOf**(int ch) — поиск последнего входения символа;
* **lastIndexOf**(int ch, int fromIndex) — поиск последнего входения символа с указанной позиции;
* **lastIndexOf**(String str) — поиск последнего вхождения строки;
* **lastIndexOf**(String str, int fromIndex) — поиск последнего вхождения строки с указанной позиции;
* **replace**(char oldChar, char newChar) — замена в строке одного символа на другой;
* **replace**(CharSequence target, CharSequence replacement) — замена одной подстроки другой;
* **substring**(int beginIndex, int endIndex) — возвратить подстроку как строку;
* **toLowerCase**() — преобразовать строку в нижний регистр;
* **toLowerCase**(Locale locale) — преобразовать строку в нижний регистр, используя указанную локализацию;
* **toUpperCase()** — преобразовать строку в верхний регистр;
* **toUpperCase**(Locale locale) — преобразовать строку в верхний регистр, используя указанную локализацию;
* **trim()** — отсечь на концах строки пустые символы;
* **valueOf(a)** — статические методы преобразования различных типов в строку.

Методы поиска возвращают индекс вхождения или -1 если искомое не найдено. Методы преобразования как replace не изменяют саму строку а возвращают соответствующий новый объект строки. 

**Методы с регулярными выражениями**

Строки также имеют ряд методов использующие [регулярные выражения](https://m.vk.com/away.php?to=http%3A%2F%2Fdarkraha.com%2Frus%2Fjava%2Fapi%2Fjcls02.php):

* **matches**(String regex) — удовлетворяет ли строка указанному регулярному выражению;
* **replaceAll**(String regex, String rplс) — заменяет все вхождения строк, удовлетворяющих регулярному выражению, указанной строкой;
* **replaceFirst**(String regex, String rplс) — заменяет первое вхождение строки, удовлетворяющей регулярному выражению, указанной строкой;
* **split**(String regex) — разбивает строку на части, границами разбиения являются вхождения строк, удовлетворяющих регулярному выражению;
* **split**(String regex, int limit) — аналогично предыдущему, но с ограничением применения регулярного выражения к строке значением limit. Если limit>0, то и размер возвращаемого массива строк не будет больше limit. Если limit<=0, то регулярное выражение применяется к строке неограниченное число раз.

## 10 часто используемых методов для работы с массивом в Java.

    Следующие 10 методов для массивов в Java являются часто используемыми. Они имеют самый высокий рейтинг в Stack Overflow — системе вопросов и ответов о программировании.  
  
**0. Объявление массива**

String[] aArray = new String[5];  
String[] bArray = {"a","b","c", "d", "e"};  
String[] cArray = new String[]{"a","b","c","d","e"};

**1. Вывод массива в Java**

int[] intArray = { 1, 2, 3, 4, 5 };  
String intArrayString = Arrays.toString(intArray);  
   
// print directly will print reference value  
System.out.println(intArray);  
// [I@7150bd4d  
   
System.out.println(intArrayString);  
// [1, 2, 3, 4, 5]

**2. Создание ArrayList из массива**

String[] stringArray = { "a", "b", "c", "d", "e" };  
ArrayList<String> arrayList = new ArrayList<String>(Arrays.asList(stringArray));  
System.out.println(arrayList);  
// [a, b, c, d, e]

**3. Проверка массива на наличие конкретного значения**

String[] stringArray = { "a", "b", "c", "d", "e" };  
boolean b = Arrays.asList(stringArray).contains("a");  
System.out.println(b);  
// true

**4. Объединение двух массивов**

int[] intArray = { 1, 2, 3, 4, 5 };  
int[] intArray2 = { 6, 7, 8, 9, 10 };  
// Apache Commons Lang library  
int[] combinedIntArray = ArrayUtils.addAll(intArray, intArray2);

**5. Объявление массива в одной строке**

method(new String[]{"a", "b", "c", "d", "e"});

**6. Объединение элементов массива в строку** 

// containing the provided list of elements  
// Apache common lang  
String j = StringUtils.join(new String[] { "a", "b", "c" }, ", ");  
System.out.println(j);  
// a, b, c

**7. Преобразование ArrayList в массив**

String[] stringArray = { "a", "b", "c", "d", "e" };  
ArrayList<String> arrayList = new ArrayList<String>(Arrays.asList(stringArray));  
String[] stringArr = new String[arrayList.size()];  
arrayList.toArray(stringArr);  
for (String s : stringArr)  
        System.out.println(s);

**8. Преобразование массива во множество (Set)**

Set<String> set = new HashSet<String>(Arrays.asList(stringArray));  
System.out.println(set);  
//[d, e, b, c, a]

**9. Возврат массива с элементами в обратном порядке**

int[] intArray = { 1, 2, 3, 4, 5 };  
ArrayUtils.reverse(intArray);  
System.out.println(Arrays.toString(intArray));  
//[5, 4, 3, 2, 1]

**10. Удаление элемента из массива**

int[] intArray = { 1, 2, 3, 4, 5 };  
int[] removed = ArrayUtils.removeElement(intArray, 3);//create a new array  
System.out.println(Arrays.toString(removed));

    И еще — создание массива типа byte на основе значения типа int  
(прим. берем ByteBuffer, выделяем в нем 4 байта и кладем число int 8, затем все это (0, 0, 0, 8) преобразуем в массив типа byte с помощью вызова array())

byte[] bytes = ByteBuffer.allocate(4).putInt(8).array();  
   
for (byte t : bytes) {  
   System.out.format("0x%x ", t);  
}

## Проектирование Классов и Интерфейсов (Перевод статьи)

* [**Переводы**](http://info.javarush.ru/blog/translation/)

* + [**перевод**](http://info.javarush.ru/tag/%D0%BF%D0%B5%D1%80%D0%B5%D0%B2%D0%BE%D0%B4/)
  + , [**интерфейсы**](http://info.javarush.ru/tag/%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81%D1%8B/)
  + , [**классы**](http://info.javarush.ru/tag/%D0%BA%D0%BB%D0%B0%D1%81%D1%81%D1%8B/)
  + , [**проектирование**](http://info.javarush.ru/tag/%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5/)

Исходная статья: [www.javacodegeeks.com/2015/09/how-to-design-classes-and-interfaces.html](http://www.javacodegeeks.com/2015/09/how-to-design-classes-and-interfaces.html)  
  
Эта статья является частью нашего Курса Академии под названием «Продвинутая Java».Этот курс призван помочь вам наиболее эффективно использовать Java. В нем обсуждаются передовые темы, такие как создание объекта, взаимосовместимость(параллелизм), преобразование в последовательную форму, отражение и многое другое. Это будет вашим путеводителем в вашем путешествии к мастерству в Java. Убедись сам!

##### Содержание

1. Введение
2. Интерфейсы
3. Интерфейс-маркеры
4. Функциональные интерфейсы, статические методы и методы по умолчанию
5. Абстрактные классы
6. Неизменяемые (постоянные) классы
7. Анонимные классы
8. Видимость
9. Наследование
10. Множественное наследование
11. Наследование и композиция
12. Инкапсуляция
13. Final классы и методы
14. Что дальше
15. Скачать исходный код

]

#### 1. ВВЕДЕНИЕ

Независимо от того, какой язык программирования вы используете (и Java здесь не исключение), следование правильным принципам проектирования — является ключевым фактором к написанию чистого, понятного и поддающегося проверке кода; а также создавать его долгоживущим, легко поддерживающим решения проблем. В этой части урока мы собираемся обсудить фундаментальные строительные блоки, которые предоставляет язык Java, и ввести пару принципов проектирования, стремясь помочь вам сделать лучшие дизайн-решений.   
  
Точнее, мы собираемся обсудить **интерфейсы** и **интерфейсы с использованием методов по умолчанию**(новая функция Java 8), **абстрактные** и **окончательные (final) классы, неизменяемые классы, наследование, композицию** и пересмотреть правила **видимости (или доступности)**, которых мы кратко коснулись в 1 части урока [«How to create and destroy objects»](http://www.javacodegeeks.com/2015/09/how-to-create-and-destroy-objects.html).

#### 2. ИНТЕРФЕЙСЫ

В объектно-ориентированном программировании, понятие интерфейсов формирует основы развития контрактов (прим. переводчика: «Контракт» в ООП — набор четко определенных условий, регулирующих отношения между классом-сервером и его клиентами). В двух словах, интерфейсы определяют набор методов (контрактов) и каждый класс, который требует поддержки этого специфичного интерфейса, должен обеспечить реализацию этих методов: довольно простая, но мощная идея.   
  
Многие языки программирования имеют интерфейсы в той или иной форме, но Java в особенности обеспечивает поддержку языка для этого. Давайте взглянем на простое интерфейсное определение в Java.

package com.javacodegeeks.advanced.design;  
  
public interface SimpleInterface {  
void performAction();  
}

Во фрагменте выше, интерфейс, который мы назвали **SimpleInterface**, объявляет только один метод с именем**performAction**. Главные отличия интерфейсов по отношению к классам — то, что интерфейсы очерчивают, какой должен быть контакт (объявляют метод), но не обеспечивают их реализацию.  
  
Однако, интерфейсы в Java могут быть сложнее: они могут включать в себя вложенные интерфейсы, классы, подсчеты, аннотации и константы. Например: 

package com.javacodegeeks.advanced.design;  
  
public interface InterfaceWithDefinitions {  
    String CONSTANT = "CONSTANT";  
  
    enum InnerEnum {  
        E1, E2;  
    }  
  
    class InnerClass {  
    }  
  
    interface InnerInterface {  
        void performInnerAction();  
    }  
  
    void performAction();  
}

В этом более сложном примере, есть несколько ограничений, которые интерфейсы безоговорочно налагают относительно вложенных конструкций и объявлений метода, и к выполнению этого принуждает компилятор Java. Прежде всего, даже если это не было объявлено явно, каждое объявление метода в интерфейсе является **публичным (public)** (и может быть только **публичным**). Таким образом следующие объявления методов эквивалентны:

public void performAction();  
void performAction();

Стоит упомянуть, что каждый отдельный метод в интерфейсе неявно объявлен **абстрактным** и даже эти объявления метода эквивалентны:

public abstract void performAction();  
public void performAction();  
void performAction();

Что касается объявленных полей констант, дополнительно к тому что они являются **публичными**, они также неявно **статические** и помечены, как ***final***. Поэтому следующие объявления также эквивалентны:

String CONSTANT = "CONSTANT";  
public static final String CONSTANT = "CONSTANT";

И, наконец, вложенные классы, интерфейсы или подсчеты, кроме того, что являются **публичными**, также неявно объявлены как **static**. К примеру, данные объявления также эквивалентны:

class InnerClass {  
}  
  
static class InnerClass {  
}

Стиль, который вы выберете — это ваше личное предпочтение, однако знание этих простых свойств интерфейсов может спасти вас от ненужного ввода.

#### 3. Интерфейс-маркер

Интерфейс маркеры — это особый вид интерфейса, у которого нет методов или других вложенных конструкций. Как это определяет библиотека Java: 

public interface Cloneable {  
}

Интерфейс-маркеры — не контракты по сути, но отчасти полезная техника, чтобы «прикрепить" или «связать" некоторую специфическую черту с классом. К примеру, относительно **Cloneable**, класс помечен как доступный для клонирования, однако, способ, которым это можно или следует реализовать, — не является частью интерфейса. Еще один очень известный и широко используемый пример интерфейс-маркера — это**Serializable**: 

public interface Serializable {  
}

Этот интерфейс помечает класс, как пригодный для преобразования в последовательную форму (сериализацию) и десериализацию (deserialization), и снова, это не указывает способ,  как это можно или следует реализовывать.   
  
Интерфейс-маркеры занимают свое место в объектно-ориентированном программировании, хотя они не удовлетворяют главную цель интерфейса, чтобы быть контрактом.

#### 4. ФУНКЦИОНАЛЬНЫЕ ИНТЕРФЕЙСЫ, МЕТОДЫ ПО УМОЛЧАНИЮ И СТАТИЧЕСКИЕ МЕТОДЫ

С выпусков Java 8, интерфейсы получили новые очень интересные возможности: статические методы, методы по умолчанию и автоматическое преобразование из лямбд (функциональные интерфейсы).   
  
В разделе интерфейсы, мы подчеркивали тот факт, что интерфейсы в Java могут только объявлять методы, но не обеспечивают их реализацию. С методом по умолчанию, все иначе: интерфейс может отметить метод ключевым словом **default** и обеспечить реализацию для него. Например: 

package com.javacodegeeks.advanced.design;  
  
public interface InterfaceWithDefaultMethods {  
    void performAction();  
  
    default void performDefaulAction() {  
        // Implementation here  
    }  
}

Будучи на уровне экземпляра, методы по умолчанию могли быть переопределены каждой реализацией интерфейса, но теперь интерфейсы также могут включать **статические** методы, например:

package com.javacodegeeks.advanced.design;  
  
public interface InterfaceWithDefaultMethods {  
    static void createAction() {  
        // Implementation here  
    }  
}

Можно сказать, что предоставление реализации в интерфейсе наносит поражение целому замыслу контрактного программирования. Но есть много причин, почему эти функции были введены в язык Java и независимо от того, насколько они полезны или сбивающие с толку, они есть там для вас и вашего пользования.  
  
Функциональные интерфейсы это совсем другая история и они опробованы, как очень полезные дополнения к языку. В основном, функциональный интерфейс — это интерфейс всего лишь с одним абстрактным методом, объявленным в нем. ***Runnable*** интерфейс из стандартной библиотеки — это очень хороший пример этой концепции. 

@FunctionalInterface  
public interface Runnable {  
    void run();  
}

Компилятор Java по-разному обрабатывает функциональные интерфейсы и может превратить лямбда-функцию в реализацию функционального интерфейса, где это имеет смысл. Давайте рассмотрим следующее описание функции: 

public void runMe( final Runnable r ) {  
    r.run();  
}

Для вызова этой функции в Java 7  и ниже должна предоставляться реализация интерфейса **Runnable**(например используя анонимные классы), но в Java 8 достаточно передать реализацию метода run()  используя синтаксис лямбды:

runMe( () -> System.out.println( "Run!" ) );

Кроме того, аннотация ***@FunctionalInterface*** (аннотации будут раскрыты в деталях в 5 части учебника) намекает, что компилятор может проверить, содержит ли интерфейс только один абстрактный метод, поэтому любые изменения, внесенные в интерфейсе в будущем не будет нарушать это предположение.

#### 5. АБСТРАКТНЫЕ КЛАССЫ

Еще одна интересная концепция, поддерживаемая языком Java, -  понятие абстрактных классов. Абстрактные классы отчасти похожи на интерфейсы в Java 7 и очень близки интерфейсу с методом по умолчанию в Java 8. В отличие от обычных классов, нельзя создавать экземпляры абстрактного классы, но он может быть подклассом (обратитесь к разделу «Наследование» для получения более подробной информации). Что еще более важно, абстрактные классы могут содержать абстрактные методы: особый вид методов без реализации, так же, как и интерфейс. Например:

package com.javacodegeeks.advanced.design;  
  
public abstract class SimpleAbstractClass {  
    public void performAction() {  
        // Implementation here  
    }  
  
    public abstract void performAnotherAction();  
}

В этом примере, класс **SimpleAbstractClass** объявлен как ***абстрактный*** и содержит один объявленный**абстрактный** метод. Абстрактные классы очень полезны, большинство или даже некоторые части деталей реализации могут совместно использоваться с многими подклассами. Как бы там ни было, они по-прежнему оставляют дверь приоткрытой и позволяют настроить поведение присущее каждому из подкласса с помощью абстрактных методов.  
  
Стоит упомянуть, в отличие от интерфейсов, которые могут содержать только **публичные** объявления, абстрактные классы могут использовать всю мощь правил доступности, чтоб управлять видимостью абстрактного метода.

#### 6. НЕИЗМЕНЯЕМЫЕ КЛАССЫ

Неизменяемость становится все более и более важной в разработке программного обеспечения в наше время. Подъем многоядерных систем вызвало много вопросов, связанных с совместным использованием данных и параллелизмом. Но одна проблема определенно возникла: небольшое (или даже отсутствие) изменяемого состояния приводит к лучшей расширяемости (масштабируемости) и более простому рассуждению о системе.  
  
К сожалению, язык Java не обеспечивает достойную поддержку классовой неизменности. Однако, пользуясь комбинацией техник,  становится возможно проектировать классы, которые неизменны. Прежде всего, все поля класса должны быть окончательные (помечены как **final**). Это хорошее начало, но не дает гарантий. 

package com.javacodegeeks.advanced.design;  
  
import java.util.Collection;  
  
public class ImmutableClass {  
    private final long id;  
    private final String[] arrayOfStrings;  
    private final Collection< String > collectionOfString;  
}

Во вторых, следите за правильной инициализацией: если поле является ссылкой на коллекцию или массив, не назначайте те поля непосредственно из аргументов конструктора, вместе этого делайте копии. Это будет гарантировать, что состоянии коллекции или массива не будет изменено за пределами.

public ImmutableClass( final long id, final String[] arrayOfStrings,  
        final Collection< String > collectionOfString) {  
    this.id = id;  
    this.arrayOfStrings = Arrays.copyOf( arrayOfStrings, arrayOfStrings.length );  
    this.collectionOfString = new ArrayList<>( collectionOfString );  
}

И наконец, обеспечение надлежащего доступа (гетеры). Для коллекций, неизменяемый вид должен быть предоставлен в виде обертки ***Collections.unmodifiableXxx***:  
  
С массивами единственный способ обеспечить настоящую неизменяемость – это предоставить копию вместо возвращения ссылки на массив. Это может быть неприемлемо с практич еской точки зрения, так как это очень зависит от размера массива и может возложить огромное давление на сборщика мусора. 

public String[] getArrayOfStrings() {  
    return Arrays.copyOf( arrayOfStrings, arrayOfStrings.length );  
}

Даже этот маленький пример дает хорошую идею, что неизменяемость еще не гражданин первого класса в Java. Все может быть усложнено, если неизменяемый класс имеет поле, ссылающийся на объект другого класса. Те классы должны быть также неизменны, однако нет никакого способа это обеспечить.   
  
Есть несколько достойных анализаторов исходного кода в Java, как FindBugs и PMD, которые могут существенно помочь, проверяя ваш код и указывая на общие недостатки программирования Java. Эти инструменты — большие друзья любого разработчика Java.

#### 7. АНОНИМНЫЕ КЛАССЫ

В предварительной Java 8 era, анонимные классы были единственным способом обеспечить оперативное определение классов и немедленное создание экземпляра. Целью анонимных классов было уменьшить шаблон и обеспечить краткий и легкий путь представления классов как запись. Давайте взглянем на типичный старомодный путь породить новый поток в Java:

package com.javacodegeeks.advanced.design;  
  
public class AnonymousClass {  
    public static void main( String[] args ) {  
        new Thread(  
            // Example of creating anonymous class which implements  
            // Runnable interface  
            new Runnable() {  
                @Override  
                public void run() {  
                    // Implementation here  
                }  
            }  
        ).start();  
    }  
}

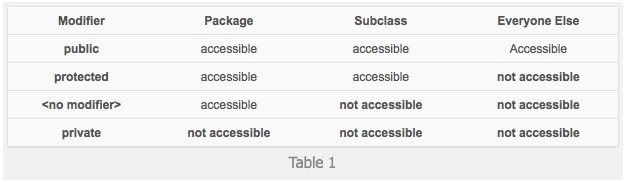
В этом примере реализация **Runnable** interface предоставляется сразу как анонимный класс. Хотя есть некоторые ограничения, связанные с анонимными классами, основные недостатки их использования — весьма подробный синтаксис конструкций, которым обязывает Java как язык. Даже просто анонимный класс, который ничего не делает, требует по меньшей мере 5 линий кода каждый раз при записи.

new Runnable() {  
   @Override  
   public void run() {  
   }  
}

К счастью, с Java 8, лямбдой и функциональными интерфейсами все эти стереотипы скоро уйдут, наконец написание кода Java будет выглядеть по настоящему кратко.

package com.javacodegeeks.advanced.design;  
  
public class AnonymousClass {  
    public static void main( String[] args ) {  
        new Thread( () -> { /\* Implementation here \*/ } ).start();  
    }  
}

#### 8. ВИДИМОСТЬ

Мы уже немного говорили о правилах видимости и доступности в Java в части 1 учебника. В этой части мы собираемся вернуться к этой теме снова, но в контексте создания подклассов.  
  
  
  
Видимость различных уровней разрешает или запрещает классам просматривать другие классы или интерфейсы (например, если они находятся в разных пакетах или вложены друг в друга) или подклассам видеть и получать доступ к методам, конструкторам и полям их родителей.  
  
В следующем разделе, наследование, мы увидим это в действии.

#### 9.  НАСЛЕДОВАНИЕ

Наследование — одно из ключевых понятий объектно-ориентированного программирования, выступающее в качестве основы построения класса связей. В сочетании с видимостью и правилами доступности, наследование позволяет проектировать классы иерархии, с возможностью расширения и поддерживания.  
  
На понятийном уровне, наследование в Java реализуется с помощью создание подклассов и ключевого словаextends, вместе с родительским классом. Подкласс наследует все публичные и защищенные элементы родительского класса. Кроме того, подкласс наследует package-private элементы родительского класса, если оба(подкласс и класс) находятся в одном пакете. При этом, очень важно, независимо от того, что вы пытаетесь спроектировать, придерживаться минимального набора метода, которые класс выставляет публично или для его подклассов. Например, давай те рассмотрим класс Parent и его подкласс Child, чтобы продемонстрировать разницу в уровнях видимости и их эффекты.

package com.javacodegeeks.advanced.design;  
  
public class Parent {  
    // Everyone can see it  
    public static final String CONSTANT = "Constant";  
  
    // No one can access it  
    private String privateField;  
    // Only subclasses can access it  
    protected String protectedField;  
  
    // No one can see it  
    private class PrivateClass {  
    }  
  
    // Only visible to subclasses  
    protected interface ProtectedInterface {  
    }  
  
    // Everyone can call it  
    public void publicAction() {  
    }  
  
    // Only subclass can call it  
    protected void protectedAction() {  
    }  
  
    // No one can call it  
    private void privateAction() {  
    }  
  
    // Only subclasses in the same package can call it  
    void packageAction() {  
    }  
}

package com.javacodegeeks.advanced.design;  
  
// Resides in the same package as parent class  
public class Child extends Parent implements Parent.ProtectedInterface {  
    @Override  
    protected void protectedAction() {  
        // Calls parent's method implementation  
        super.protectedAction();  
    }  
  
    @Override  
    void packageAction() {  
        // Do nothing, no call to parent's method implementation  
    }  
  
    public void childAction() {  
        this.protectedField = "value";  
    }  
}

Наследование — очень большая тема сама по себе, с большим количеством тонких деталей, характерных для Java. Однако, есть несколько правил, которым легко следовать, и которые могут очень помочь сохранить краткость в классовой иерархии. В Java, каждый подкласс может переопределять любые унаследованные методы его родителя, если он не был объявлен как окончательный (final).   
  
Однако, нет специального синтаксиса или ключевого слова, чтобы пометить метод, как переопределенный, что может привести к путанице. Вот почему была введена аннотация @Override:  всякий раз, когда ваша цель – переопределить наследуемый метод, пожалуйста, используйте аннотацию @Override, чтобы кратко обозначить это.   
  
Другая дилемма, с которой Java разработчики постоянно сталкиваются в проектирование — это построение классов иерархии (с конкретными или абстрактными классами) в сравнении с реализацией интерфейсов. Настоятельно рекомендуем отдавать предпочтение интерфейсам по отношению к классам или абстрактным классам, где это возможно. Интерфейсы более легкие, их проще тестировать и поддерживать, плюс ко всему, они минимизируют побочные эффект изменений реализации. Многие продвинутые техники программирования, такие как создание прокси (proxy) классов в стандартной библиотеке Java, в большей степени полагаются на интерфейсы.

#### 10. МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ.

В отличие от С++ и некоторых других языков, Java не поддерживает множественное наследование: в Java каждый класс может иметь только одного прямого родителя (с классом Object в вершине иерархии). Однако, класс может реализовывать несколько интерфейсов, и, таким образом, стекование (stacking) интерфейсов — единственный способ достигнуть (или имитировать) множественное наследование в Java.

package com.javacodegeeks.advanced.design;  
  
public class MultipleInterfaces implements Runnable, AutoCloseable {  
    @Override  
    public void run() {  
        // Some implementation here  
    }  
  
    @Override  
    public void close() throws Exception {  
       // Some implementation here  
    }  
}

Реализация нескольких интерфейсов на самом деле довольно мощная, но часто необходимость снова и снова использовать реализацию приводит к глубокой классовой иерархии, как способ преодолеть отсутствие поддержки множественного наследования в Java. 

public class A implements Runnable {  
    @Override  
    public void run() {  
        // Some implementation here  
    }  
}

// Class B wants to inherit the implementation of run() method from class A.  
public class B extends A implements AutoCloseable {  
    @Override  
    public void close() throws Exception {  
       // Some implementation here  
    }  
}

// Class C wants to inherit the implementation of run() method from class A  
// and the implementation of close() method from class B.  
public class C extends B implements Readable {  
    @Override  
    public int read(java.nio.CharBuffer cb) throws IOException {  
       // Some implementation here  
    }  
}

И так далее… Недавний выпуск Java 8 несколько решает проблему с внедрением методов по умолчанию. Из-за методов по умолчанию, интерфейсы фактические стали предоставлять не только контракт, но и реализацию. Следовательно, классы, которые реализуют эте интерфейсы, также автоматически унаследуют эти реализованные методы. Например:

package com.javacodegeeks.advanced.design;  
  
public interface DefaultMethods extends Runnable, AutoCloseable {  
    @Override  
    default void run() {  
        // Some implementation here  
    }  
  
    @Override  
    default void close() throws Exception {  
       // Some implementation here  
    }  
}  
  
// Class C inherits the implementation of run() and close() methods from the  
// DefaultMethods interface.  
public class C implements DefaultMethods, Readable {  
    @Override  
    public int read(java.nio.CharBuffer cb) throws IOException {  
       // Some implementation here  
    }  
}

Имейте в виду, что множественное наследование очень мощный, но и в тоже время опасный инструмент. Хорошо известную проблему «Ромб смерти» часто называют основным дефектом реализации множественного наследования, поэтому разработчики вынуждены проектировать классы иерархии весьма тщательно. К сожалению, интерфейсы Java 8 с методами по умолчанию также становится жертвой этих дефектов.

interface A {  
    default void performAction() {  
    }  
}  
  
interface B extends A {  
    @Override  
    default void performAction() {  
    }  
}  
  
interface C extends A {  
    @Override  
    default void performAction() {  
    }  
}

Например, следующий фрагмент кода не удастся скомпилировать:

// E is not compilable unless it overrides performAction() as well  
interface E extends B, C {  
}

На данный момент,  справедливо сказать, что Java как язык всегда пытался избежать угловых случаем объектно-ориентированного программирования, но, так как язык развивается, некоторые из тех случаем стали внезапно появляется.

#### 11. НАСЛЕДОВАНИЕ И КОМПОЗИЦИЯ.

К счастью, наследование не единственный путь спроектировать ваш класс. Другой альтернативой, которая, по мнению многих разработчиков, является намного лучшей, чем наследование, — является композиция. Идея очень проста: вместо создания иерархии классов, их нужно компоновать из других классов.   
Давайте посмотрим на этот пример:

// E is not compilable unless it overrides performAction() as well  
interface E extends B, C {  
}

Класс Vehicle состоит из двигателя (engine) и колес (плюс многие другие части, которые оставлены в стороне для простоты). Однако, можно сказать, что класс Vehicle так же является машиной (engine), так что может быть спроектирована с использованием наследования. 

public class Vehicle extends Engine {  
    private Wheels[] wheels;  
    // ...  
}

Какое решение проектирования будет правильным? Общие основные рекомендации известны как **IS-A**(является) и **HAS-A** (содержит) принципы. **IS-A** — это связь наследования: подкласс также удовлетворяет классовую спецификацию родительского класса и разновидность родительского класса (Прим. Переводчика: В книге Heading in Java принцип IS-A описан так: «Когда один класс наследует другой, мы говорим, что дочерний класс(подкласс) расширяет родительский. Если вы хотите узнать, расширяет ли одна сущность другую, проведите проверку на соответствие — IS-A (является).»). Следовательно, **HAS-A** это связь композиции: класс владеет (или содержит) объект, который ему принадлежит. В большинстве случаем, принцип HAS-A работает лучше, чем **IS-A** по ряду причин:

* Проектирование более гибкое;
* Модель более стабильная, так как изменения не распространяются через классовую иерархию;
* Класс и его композиция слабо связаны по сравнению с композицией, которая плотно связывает родителя и его подкласс.
* Логический ход мысли в классе проще, так как все его зависимости включены в нем же, в одном месте.

Как бы там ни было, наследование имеет свое место, решает ряд существующих проблем проектирования различными способами, так что не следует им пренебрегать. Пожалуйста, удерживайте эти две альтернативы у себя в голове при проектировании вашей объектно-ориентированной модели.

#### 12. ИНКАПСУЛЯЦИЯ.

Понятие инкапсуляции в объектно-ориентированном программировании заключается в скрытие всех деталей реализации (как режим работы, внутренние методы и т.д.) от внешнего мира. Преимущества от инкапсуляции в удобстве сопровождения и легкости изменений. Менее внутренние детали класса делают видимым, больше контроля разработчики имеют над изменениями внутренней реализации, без страха нарушить существующий код (реальная проблема, если вы разрабатываете библиотеку или фреймфорки (структуры), использованную многими людьми).  
  
Инкапсуляция в Java достигается с помощью правил видимости и доступности. В Java, считается, что лучше всего никогда не выставлять поля напрямую, только посредством гетеров (getters) и сетеров (setters) (если поля не помечены как окончательные (final)). Например:

package com.javacodegeeks.advanced.design;  
  
public class Encapsulation {  
    private final String email;  
    private String address;  
  
    public Encapsulation( final String email ) {  
        this.email = email;  
    }  
  
    public String getAddress() {  
        return address;  
    }  
  
    public void setAddress(String address) {  
        this.address = address;  
    }  
  
    public String getEmail() {  
        return email;  
    }  
}

Этот пример напоминает то, что называется ***JavaBeans*** в языке Java: стандартные классы Java, написаны соответственно набору соглашений, один из которых дает доступ к полям только с помощью гетер и сеттер методов.  
  
Как мы уже подчеркивали в разделе наследования, пожалуйста, всегда придерживайте минимальному контракту публичности в классе, используя принципы инкапсуляции. Все, что не должно быть публичным — должно стать приватным (или protected/ package private, зависит от проблемы, что вы решаете). В долгосрочной перспективе это вам окупится, давая вам свободу в проектировании без внесения критических изменений (или, по крайней мере, минимизируют их).

#### 13. ОКОНЧАТЕЛЬНЫЕ КЛАССЫ И МЕТОДЫ

В Java, есть способ предотвратить возможность класса стать подклассом от другого класса: другой класс должен быть объявлен как окончательный (final). 

package com.javacodegeeks.advanced.design;  
  
public final class FinalClass {  
}

Это же ключевое слово final в объявление метода предотвращает возможность переопределения метода в подклассах. 

package com.javacodegeeks.advanced.design;  
  
public class FinalMethod {  
    public final void performAction() {  
    }  
}

Нет общих правил, чтобы решить должен класс или методы быть окончательными или нет. Окончательные классы и методы ограничивают расширяемость и очень сложно думать наперед, должен или не должен класс быть унаследованным, или должен или не должен метод быть переопределен в будущем. Это особенно важно для разработчиков библиотеки, поскольку решения проектирования подобны этому могли бы существенно ограничить применимость библиотеки.  
  
Стандартная библиотека Java имеет несколько примеров окончательный классов, причем наиболее известный — это класс String. На ранней стадии, было принято данное решение, чтобы предотвратить любые попытки разработчиков появиться с собственным, «лучшим» решением реализации string.

#### 14. ЧТО ДАЛЬШЕ

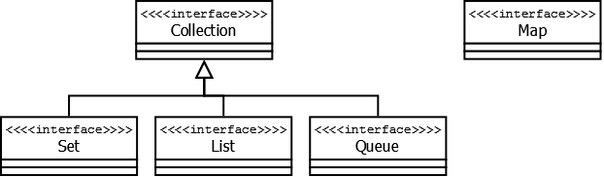
В этой части урока мы рассмотрели концепции объектно-ориентированного программирования в Java. Мы также кратко прошлись по контрактному программированию, затронули некоторые функциональные понятия и увидели, как язык развивался с течением времени. В следующей части урока мы собираемся встретиться с generics и как они меняют способ приближения типовой безопасности в программировании.

#### 15. СКАЧАТЬ ИСХОДНЫЙ КОД

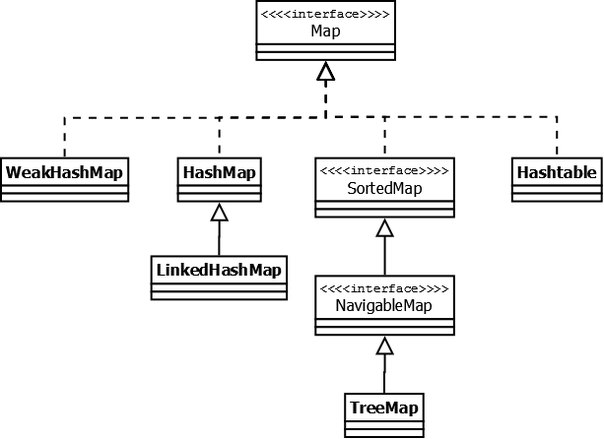
Это был урок о том, «Как Проектировать Классы и Интерфейсы».  
Вы можете скачать исходный код здесь: [advanced-java-part-3](http://a3ab771892fd198a96736e50.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/09/advanced-java-part-3.zip)»>

Справочник по Java Collections Framework

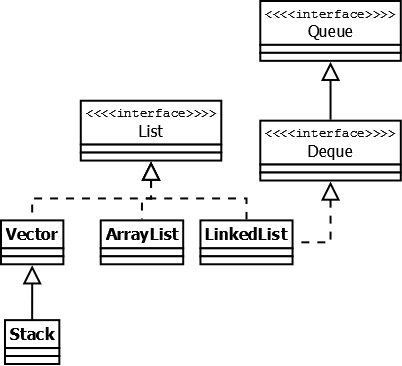
Справочник по Java Collections Framework   
  
Данная публикация не является полным разбором или анализом (не покрывает пакет java.util.concurrent). Это, скорее, справочник, который поможет начинающим разработчикам понять ключевые отличия одних коллекций от других, а более опытным разработчикам просто освежить материал в памяти.   
  
Что такое Java Collections Framework?   
Java Collection Framework — иерархия интерфейсов и их реализаций, которая является частью JDK и позволяет разработчику пользоваться большим количесвом структур данных из «коробки».   
  
Базовые понятия   
На вершине иерархии в Java Collection Framework располагаются 2 интерфейса: Collection и Map. Эти интерфейсы разделяют все коллекции, входящие во фреймворк на две части по типу хранения данных: простые последовательные наборы элементов и наборы пар «ключ — значение» (словари).

[](https://vk.com/photo232854130_378557275)

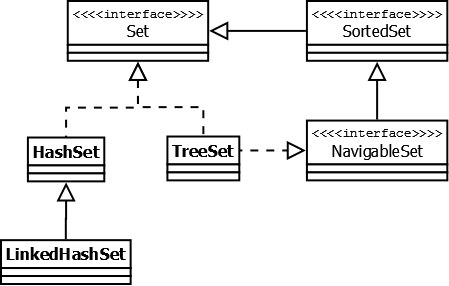
Collection — этот интерфейс находится в составе JDK c версии 1.2 и определяет основные методы работы с простыми наборами элементов, которые будут общими для всех его реализаций (например size(), isEmpty(), add(E e) и др.). Интерфейс был слегка доработан с приходом дженериков в Java 1.5. Так же в версии Java 8 было добавлено несколько новых метода для работы с лямбдами (такие как stream(), parallelStream(), removeIf(Predicate<? super E> filter) и др.).   
  
Важно также отметить, что эти медоды были реализованы непосредственно в интерфейсе как default-медоды.   
  
Map. Данный интерфейс также находится в составе JDK c версии 1.2 и предоставляет разработчику базовые методы для работы с данными вида «ключ — значение».Также как и Collection, он был дополнен дженериками в версии Java 1.5 и в версии Java 8 появились дополнительные методы для работы с лямбдами, а также методы, которые зачастую реализовались в логике приложения (getOrDefault(Object key, V defaultValue), putIfAbsent(K key, V value)).   
  
Интерфейс Map

[](https://vk.com/photo232854130_378557338)

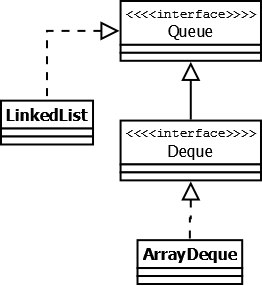
Hashtable — реализация такой структуры данных, как хэш-таблица. Она не позволяет использовать null в качестве значения или ключа. Эта коллекция была реализована раньше, чем Java Collection Framework, но в последствии была включена в его состав. Как и другие коллекции из Java 1.0, Hashtable является синхронизированной (почти все методы помечены как synchronized). Из-за этой особенности у неё имеются существенные проблемы с производительностью и, начиная с Java 1.2, в большинстве случаев рекомендуется использовать другие реализации интерфейса Map ввиду отсутствия у них синхронизации.   
  
HashMap — коллекция является альтернативой Hashtable. Двумя основными отличиями от Hashtable являются то, что HashMap не синхронизирована и HashMap позволяет использовать null как в качестве ключа, так и значения. Так же как и Hashtable, данная коллекция не является упорядоченной: порядок хранения элементов зависит от хэш-функции. Добавление элемента выполняется за константное время O(1), но время удаления, получения зависит от распределения хэш-функции. В идеале является константным, но может быть и линейным O(n). Более подробную информацию о HashMap можно почитать здесь (актуально для Java < 8).   
  
LinkedHashMap — это упорядоченная реализация хэш-таблицы. Здесь, в отличии от HashMap, порядок итерирования равен порядку добавления элементов. Данная особенность достигается благодаря двунаправленным связям между элементами (аналогично LinkedList). Но это преимущество имеет также и недостаток — увеличение памяти, которое занимет коллекция. Более подробная информация изложена в этой статье.   
  
TreeMap — реализация Map основанная на красно-чёрных деревьях. Как и LinkedHashMap является упорядоченной. По-умолчанию, коллекция сортируется по ключам с использованием принципа "natural ordering", но это поведение может быть настроено под конкретную задачу при помощи объекта Comparator, которые указывается в качестве параметра при создании объекта TreeMap.   
  
WeakHashMap — реализация хэш-таблицы, которая организована с использованием weak references. Другими словами, Garbage Collector автоматически удалит элемент из коллекции при следующей сборке мусора, если на ключ этого элеметна нет жёстких ссылок.   
  
Интерфейс List

[](https://vk.com/photo232854130_378557430)

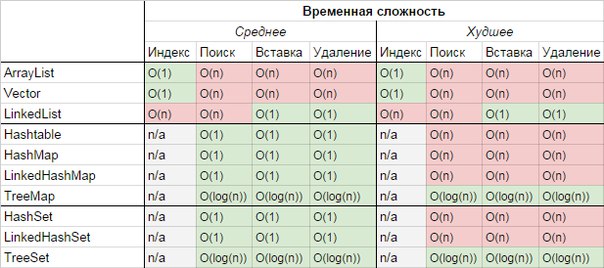
Реализации этого интерфейса представляют собой упорядоченные коллекции. Кроме того, разработчику предоставляется возможность доступа к элементам коллекции по индексу и по значению (так как реализации позволяют хранить дубликаты, результатом поиска по значению будет первое найденное вхождение).   
  
Vector — реализация динамического массива объектов. Позволяет хранить любые данные, включая null в качестве элемента. Vector появился в JDK версии Java 1.0, но как и Hashtable, эту коллекцию не рекомендуется использовать, если не требуется достижения потокобезопасности. Потому как в Vector, в отличии от других реализаций List, все операции с данными являются синхронизированными. В качестве альтернативы часто применяется аналог — ArrayList.   
  
Stack — данная коллекция является расширением коллекции Vector. Была добавлена в Java 1.0 как реализация стека LIFO (last-in-first-out). Является частично синхронизированной коллекцией (кроме метода добавления push()). После добавления в Java 1.6 интерфейса Deque, рекомендуется использовать именно реализации этого интерфейса, например ArrayDeque.   
  
ArrayList — как и Vector является реализацией динамического массива объектов. Позволяет хранить любые данные, включая null в качестве элемента. Как можно догадаться из названия, его реализация основана на обычном массиве. Данную реализацию следует применять, если в процессе работы с коллекцией предплагается частое обращение к элементам по индексу. Из-за особенностей реализации поиндексное обращение к элементам выполняется за константное время O(1). Но данную коллекцию рекомендуется избегать, если требуется частое удаление/добавление элементов в середину коллекции. Подробный анализ и описание можно почитать в этом хабратопике.   
  
LinkedList — ещё одина реализация List. Позволяет хранить любые данные, включая null. Особенностью реализации данной коллекции является то, что в её основе лежит двунаправленный связный список (каждый элемент имеет ссылку на предыдущий и следующий). Благодаря этому, добавление и удаление из середины, доступ по индексу, значению происходит за линейное время O(n), а из начала и конца за константное O(1). Так же, ввиду реализации, данную коллекцию можно использовать как стек или очередь. Для этого в ней реализованы соответсвующие методы. На Хабре также есть статья с подробным анализом и описанием этой коллекции.   
  
Интерфейс Set

[](https://vk.com/photo232854130_378557471)

Представляет собой неупорядоченную коллекцию, которая не может содержать дублирующиеся данные. Является программной моделью математического понятия «множество».   
  
HashSet — реализация интерфейса Set, базирующаяся на HashMap. Внутри использует объект HashMap для хранения данных. В качестве ключа используется добавляемый элемент, а в качестве значения — объект-пустышка (new Object()). Из-за особенностей реализации порядок элементов не гарантируется при добавлении.   
  
LinkedHashSet — отличается от HashSet только тем, что в основе лежит LinkedHashMap вместо HashSet. Благодаря этому отличию порядок элементов при обходе коллекции является идентичным порядку добавления элементов.   
  
TreeSet — аналогично другим классам-реализациям интерфейса Set содержит в себе объект NavigableMap, что и обуславливает его поведение. Предоставляет возможность управлять порядком элементов в коллекции при помощи объекта Comparator, либо сохраняет элементы с использованием "natural ordering".   
  
Интерфейс Queue

[](https://vk.com/photo232854130_378557517)

Этот интерфейс описывает коллекции с предопределённым способом вставки и извлечения элементов, а именно — очереди FIFO (first-in-first-out). Помимо методов, определённых в интерфейсе Collection, определяет дополнительные методы для извлечения и добавления элементов в очередь. Большинство реализаций данного интерфейса находится в пакете java.util.concurrent и подробно рассматриваются в данном обзоре.   
  
PriorityQueue — является единственной прямой реализацией интерфейса Queue (была добавлена, как и интерфейс Queue, в Java 1.5), не считая класса LinkedList, который так же реализует этот интерфейс, но был реализован намного раньше. Особенностью данной очереди является возможность управления порядком элементов. По-умолчанию, элементы сортируются с использованием «natural ordering», но это поведение может быть переопределено при помощи объекта Comparator, который задаётся при создании очереди. Данная коллекция не поддерживает null в качестве элементов.   
  
ArrayDeque — реализация интерфейса Deque, который расширяет интерфейс Queue методами, позволяющими реализовать конструкцию вида LIFO (last-in-first-out). Интерфейс Deque и реализация ArrayDeque были добавлены в Java 1.6. Эта коллекция представляет собой реализацию с использованием массивов, подобно ArrayList, но не позволяет обращаться к элементам по индексу и хранение null. Как заявлено в документации, коллекция работает быстрее чем Stack, если используется как LIFO коллекция, а также быстрее чем LinkedList, если используется как FIFO.   
  
Заключение   
Java Collections Framework содержит большое количество различных структур данных, доступных в JDK «из коробки», которые в большинстве случаев покрывают все потребности при реализации логики приложения. Сравнение временных характеристик основных коллекций, которые зачастую используются в разработке приложений приведено в таблице:

[](https://vk.com/photo232854130_378557570)

При необходимости, разработчик может создать собственную реализацию, расширив или переопределив существующую логику, либо создав свою собственную реализацию подходящего интерфейса с нуля. Также существует некоторое количество готовых решений, которые являются альтернативой или дополнением к Java Collections Framework. Наиболее популярными являются Google Guava и Commons Collections.