Язык Java.

Что такое java?

Java – это мультипарадигменный (часто объектно-ориентированный), сильно типизированный, кроссплатформенный язык общего назначения. Является статистическим типизированным языком.

Java применяется:

Серверные технологии

Высоконагруженные системы и базы данных

Распределенные системы

Веб-сервисы

Десктопные приложения

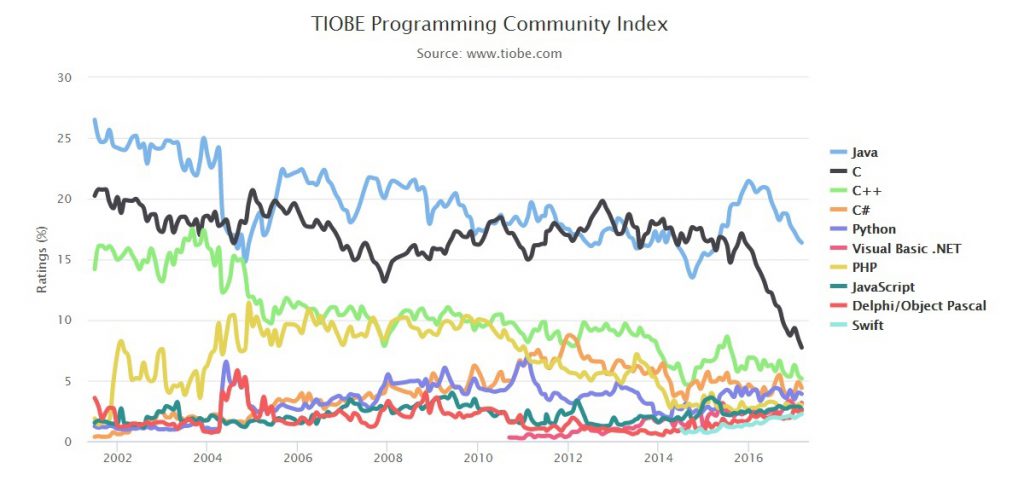
Игры

Телефоны (J2ME, Android)

Апплеты для смарт-карт

Интерактивный контент Blu-Ray

Популярность на 2018 год.



Плюсы Java

+ Java не умирает

+ Совместимость

+Статистическая (и сильная типизация)

+Строгая спецификация языка и виртуальная машина

+Библиотека

+Инструментарий

+Модель памяти

+Скорость

+Безопасность

Инструменты в Java.

Инструментами программиста являются:

Java – язык программирования

JDK (java development kit) – инструментарий разработчика

JVM (Java Virtual machine) – виртуальная машина

JRE (Java Runtime Environment) – среда исполнения

Жизненный цикл.

В языке Java используется определенный жизненный цикл. Но стоит учитывать то, что в скриптовых языках(JavaScript, Python, PHP и т.п.) нет промежуточного представления.

1. Исходный код,
2. Промежуточное представление 1,
3. Промежуточное представление N,
4. Интерпретация (исполнение),

В целом промежуточное представление делится на 2 стадии:

В памяти

В процессоре

Память.

В языке Java используется автоматическое управление, в противовес явной управления памяти. В языке Java для выделения памяти используются разные способы.

Например: Bump allocation – самый простой способ заключается в том, что есть указатель на ячейку памяти (с учетом то, что мы знаем то что до нее память заполнена, а после свободна) в нее мы сдвигаем указатель на нужное нам количество и записываем байты. Таким образом, мы уплотняем объекты.

Байт-код.

Байт-код Java — набор инструкций, исполняемых виртуальной машиной Java. Каждый код операции байт-кода — один байт. Используются не все 256 возможных значений кодов операций. 51 из них зарезервированы для использования в будущем.

Безопасность в Java

Прежде всего, java-код должен пройти верификацию байт-кода. В java нет указателя как у C. Сам же указатель – это именованная ячейка памяти.

В java есть лишь ссылки на java-объекты и ссылками нельзя провести арифметические операции. Java поддерживает приведение типов, проверку диапазона индекса массива, на переполнение стека

Базовый синтаксис Java

Инструкция, выражения и «;».

Инструкции или оператор – это наименьшая законченная часть кода.

Выражение – конструкция, созданная из переменных, операций и т.п.

«;» - разделяет инструкции.

Значение и их типы.

В языке java присутствуют значении. У каждого значения свой тип. В общем, существует всего лишь 5 значений:

* Логические.
* Целочисленные.
* Дробные.
* Символьные.
* Ссылочные.

Логические

Логические типы отвечают за логические действия.

К логической группе относят тип *boolean*. В общем boolean может применять true (истинна) и false (ложь). Все они обычно обозначают *условие.*

Целочисленные

Для целых чисел определенно 4 типа: *byte*, *int*, *long*, *short*. В java целочисленные поддерживают положительные и отрицательные числа.

Byte – 8 разрядный тип с диапазоном -128: 127. Является наименьшей по длине в байтах.

Int – 32-разрядный тип с диапазоном -2147483648: 2147483647. Является наиболее распространенным в использовании.

Short - 16 разрядный с диапазоном 32768: 32767.

Long – 64 разрядный с диапазоном -9223372036854775808: 9223372036854775807.

Символьные.

Символьный тип – это знак поддерживающий кодировку Unicode, также с ними можно производить операции.

char ch= 88 эквивалентен char ch=”X” Также его можно декрементировать и инкрементировать.

Ссылочный.

Ссылочный – хранит не самого его, а ссылку на него. К таким можно отнести String и Object, в среде разработки ссылочный тип подсвечивается белым шрифтом. Стоит добавить то, что каждый объект является ссылочным типом данных.

Факты о значениях.

1) Значение есть результат вычисления выражения (за исключение типа void)

2)Значения можно сохранять в переменные (поля, константы и т.д.)

3)Можно передать значение в методы аргументом и принимать в виде параметра.

4)При передаче в метод, присваивании значений копируется.

5)Значение нельзя изменить, но можно присвоить переменной новое значение.

Объявление локальных переменных.

Синтаксис: *поля тип имя = инициализирующее\_значение*

Например:

int I;

I = 256;

final f = 24;

Переменные.

1. Локальная переменная видима в блоке кода, в котором объявлена, включая вложенные блоки.

2. Нельзя объявить переменную с ранее используемым именем.

1. Значение final –переменной можно объявить только один раз.

Операции

Логические.

Логические операции обычно применяются с boolean и используются для логических задач. Основные операции:

|  |  |  |
| --- | --- | --- |
| Операция | Сокращенная операция. | Описание |
| & | && | & обозначает и |
| | | || | | обозначает или |
| ^ |  | Исключающие или ^| |
| == |  | Проверяет на равенство |

Арифметические.

Арифметические операторы — используются в математических выражениях таким же образом, как они используются в алгебре.

Оператор Описание

+ Складывает значения по обе стороны от оператора

- Вычитает правый операнд из левого операнда

\* Умножает значения по обе стороны от оператора

% Делит левый операнд на правый операнд и возвращает остаток

++ Инкремент - увеличивает значение операнда на 1

-- Декремент - уменьшает значение операнда на 1

Полезные вещи

* abc () – позволяет получить модуль числа.
* hash Code () - позволяет получить кэш чего-либо.
* Infinity – бесконечное число.
* NaN – число которое не больше, не меньше и не равно никакому числу (в том числе и себе). Возвращает !себя.
* java.math.BigInteger – позволяет задать любое большое целочисленное число, которое влезет в память.
* java.math.BigDemical - позволяет задать любое большое дробное число, которое влезет в память.

Преобразование.

Преобразование – это приведение переменной одного значения другой с другим типом. Минус преобразования - потеря точности.

Можно выделить два способа: расширяющий и сужающий.

Расширяющий способ преобразования.

Различия Расширяющего способа заключается в том, что не теряется точность числа. Но при этом такую операцию можно использовать, например: целочисленное значение с целочисленным значением. Также в таком способе нужно чтоб одно покрывало другое число. Например:

public class Main {

public static void main (String[] args) {

int i = Integer.MAX\_VALUE;

long l= i;

System.out.println (l);

}

}

Сужающий способ преобразования.

Различие Сужающего способа заключается в том, что теряется точность. Но при этом такую операцию можно провести со всеми типами сразу. Например:

public class Main {

public static void main (String [] args) {

double i = Double.MAX\_VALUE;

long l=(long) i;

System.out.println(l);

}

}

Массивы

Массив – объект в куче, содержащий ряд чего-либо. В общем, массив объявляется:

тип [] имя = new тип [] {содержимое}

import java.util.Arrays.\*;

public class Main {

public static void main(String[] args) {

int [] i = new int[] {34,345};

i[1] +=2;

System.out.println(i.length);

System.out.println(i[1]);

int[] copy = i.clone();

System.out.println(copy);

System.out.println(i==copy);

System.out.println(i.equals(copy));

double [][] fd ={{23.5, 37.8, 65.05}, {453.3, 3.2, 5.5}};

System.out.print(Arrays.toString(i));

}

}

Управляющие конструкции.

Предложение блок.

Предложение-блок – это строки внутри блока. Предложение-блок обычно используется в сложных управляющих конструкциях.

{

// предложение

// предложение

}

if

Синтаксис:

if (условие) предложение

[else предложение]

Например:

int i;

i= -23;

if (i>0) {

System.out.println("positive Number");

} else {

System.out.println ("negative or null");

}

Усложненная цепочка:

public class Main {

public static void main(String[] args){

int i;

i= -23;

if(i>0){

System.out.println("positive number");

}else if(i==null){

System.out.println("null number" );

}else{

System.out.println("negative number");

}

}

}

Условный (тернарный) оператор.

Тернарный оператор – это выражение, схожее с условной конструкцией if. Пример:

public class Main {

public static void main (String [] args) {

int i = 12;

System.out.println (i<0 ? "negative" : "positive");

}

}

Switch

Switch применим к известному числу в разных операциях. Switch довольно бесполезная условная конструкция.

public class Main {

public static void main (String [] args) {

int j = 12+ 40;

switch (j) {

case 52:

System.out.println("52");

break;

case 57:

System.out.println("57");

break;

default:

System.out.println("other");

}

}

}

Циклы.

While

Цикл while простой и удобный цикл в применении. Синтаксис:

while (*условие){*

*предложение*

*}*

public class Main {

public static void main (String [] args) {

int i = 14;

while (i>0){

System.out.println (i);

i--;

}

}

}

Также существует бесконечный цикл. В целом он бесполезен, но можно над ним посмеяться.

public class Main {

public static void main (String [] args) {

while (true) {

System.out.println ("Snatch");

}

}

}

Do-while

public class Main {

public static void main(String[] args) {

int i=1;

do{

i++;

System.out.println(i);

}while(i<9);

}

}

For

public class Main {

public static void main(String[] args) {

int i;

for(i=0; i<10; i++){

System.out.println(i);

}

}

}

For-each

public class Main {

public static void main(String[] args) {

int [] i = {1, 45, 56};

for(int g : i){

System.out.println(g);

}

}

}

О break и continue.

Break – позволяет выйти из цикла и т.п.

Continue – позволяет перейти на стороннюю итерации, не дорабатывая в которой находится.

Объекты, методы, классы, строки, интерфейсы и т.д.

Объекты.

Объекты часто встречается в java, так сам язык объектно-ориентированный и работа с ними является важным аспектом. Объекты являются в своем роде краеугольным камнем программирования на java.

Объект – область, зафиксированная значения в куче, обладающая фиксированной архитектурой, содержащий заголовок и полезную информацию в виде набора значения.

Для работы с объектами нужна ссылка на объект. Значение-ссылка может ссылаться на объект или null. В Java нельзя объявить ссылку на середину объекта и т.п.

Объект обладает идентичностью. Объект не будет равен никогда другому объекту, только равен себе.

Доступ к объекту нельзя получить в обход. Объект не может быть реинтерпретирован.

Любой объект также является объектом типа java.lang.Object .

Методы.

Метод – простыми словами действия в программе. Официально - это законченная последовательность действий (инструкций), направленных на решение отдельной задачи.

В java методы помечаются (). Внизу представлен синтаксис методов:

*поля* имя\_метода(){

реализация метода

}

Если метод абстрактный он может не содержать реализацию.

System.out.println (“Это метод”);

Выражение (вызов метода)

Возвратные/невозвратные.

В java методы делятся на возвратные и невозвратные методы.

Возвратные методы возвращают что либо. Такие методы в своей реализации должны возвращать, что-либо с помощью слова return. В целом они обычно используют для работы с определенными данными и нам нужны только данные.

Невозвратные методы помечаются ключевым словом void. Если они и возвращают, то явно.

+++Например:

public class Main {

int j=10;

public int inkrement(){

j +=10;

return j;

}

int i=0;

int d=1;

public void show(){

System.out.println("i равна:"+i+" d равна:"+d);

}

public static void main(String[] args) {

Main M = new Main();

m.show();

System.out.println(j);

}

}

Передача параметров в методы.

В java метод может принимать параметры. Этих параметров может быть сколько угодно. В целом параметр – это поле непосредственно используемым методом в java.

+++Пример:

class Main{

static void test(int i, String s){

System.out.println(i+s);

}

public static void main(String [] args){

test(12, "rf");

}

}

Перегрузка методов.

В Java в отличии C# и C++ отсутствует перегрузка параметров. Перегрузка – это изменение списка параметров, а то и вовсе удалить параметры.

+++Пример

public class Main{

static String s = "Это строка.";

static int i = 23;

void test(String str){

System.out.println(str);

}

void test(String str, int n){

System.out.println("тест отображает:"+str+", "+n);

}

public static void main (String [] args){

Main mcl = new Main();

mcl.test(s, i);

}

}

Varargs

Переменное количество параметров (Varargs) – это неопределенное количество параметров у метода.

Varargs часто используют в программировании так, как порой например: нужно опустить одни параметры и т.п.

Переменное количество де-факто – это массив, но массив с параметрами. Вместо стандартного знака «[]» используется «…». Так объявляется этакий массив. Также в объявление методов с переменным количеством параметров массив параметров объявляется последнюю очередь.

Такие методы могут участвовать в перегрузке и переопределении метода.

Метод с именем класса???

Можно ли назвать методы имен класса? Ответ – да. Конечно это можно, но делать это не желательно.

Пример:

public class Main {

static void Main(){

System.out.println("можно ли назвать методы классов.");

}

public static void main(String[] args) {

Main();

}

}

Снаружи бутылка, внутри пар – Александр Невский.

Конструкторы.

Конструктор – это конструирование объектов с определенными полями, но с возможностью изменений.

+++Пример:

public class Point{

int i, c;

Point(int i, int c){

this.i=i;

this.c=c;

}

void print(){

System.out.println(i+","+c);

}

public static void main(String[] args) {

Point myPoint = new Point(34, 56);

myPoint.print();

}

}

Деструкторов в java нет. Эта функция выполняется автоматически.

Неизменный объект.

В последнее время в программировании на java стало популярно использовать неизменные объекты. Их достоинство в том, что при их использовании можно не волноваться разным ошибкам.

Сам неизменный объект – это когда его видимое поведение неизменно после создания объекта.

Чтоб получить, такой конструктор все принимаемые им параметры, нужно указать как final, в таком случае налагается инициализация этих параметров.

Пример:

public class Point{

final int i, c;

Point(int i, int c){

this.i=i;

this.c=c;

}

void print(){

System.out.println(i+","+c);

}

public static void main(String[] args) {

Point myPoint = new Point(34, 56);

myPoint.print();

}

}

/\* выводит

34, 56

\*/

Вложенные классы.

Создания вложенных классов порой полезная вещь. Например: когда нужно отделить отдельную большую функцию, сделать ответвление и т.п.

Пример:

public class Point {

final int i, d;

class Anothere {

void print(){

System.out.println("из вложенного класса:"+i+","+d);

}

}

Point(int i, int d){

this.i=i;

this.d=d;

}

Anothere getAnothere(){ return new Anothere(); }

void show(){

System.out.println("из внутреннего класса:"+i+","+d);

}

public static void main (String [] args) {

Point p= new Point(23, 55);

Point.Anothere other =p.getAnothere();

other.print();

p.show();

}

}

Утилитные классы.

В Java существуют классы с закрытым конструктором, но с открытым доступом к методам. Они называются утилитными. Несколько из них:

java.util.Arrays .

java.lang.Math .

Статистические члены класса.

Статистическое поле: поле, не привязанное не к объекту, а классу. Статистическое поле может владеть одним экземпляром.

Когда приходится что-то хранить в статистическом поле, то не следует хранить, изменяемы значения. Следует использовать для хранения констант; объектов, которые пригодятся в будущем; для глобального hash-code.

В нем не может использоваться this.

Следует создавать статический метод, если де-факто он статический, и он использует состояние класса.

Создание статических вложенных классов означает, то, что он не будет владеть ссылками извне.

Внутри не статического объекта нельзя объявить статический элемент.

Класс object.

В Java все объекты унаследованы от класса object. В классе Object определен ряд методов. Некоторые актуальны и сегодня, а некоторые уже не используются, а если используются то очень редко.

Таблица методов класса object:

|  |  |
| --- | --- |
| Метод | Описание |
| Boolean equals(Object obj) | Определяет, равен ли объект к другому объекту. |
| void finalize() | Вызывается перед удалением неиспользуемого объекта. |
| getClass() | Получает класс объекта во время выполнения. |
| int hashCode() | Возвращает хэш-код, связанный с вызывающим объектом. |
| void notify() | Возобновляет выполнение потока, который ожидает вызывающего объекта. |
| void notifyAll() | Возобновляет выполнение всех потоков, которые ожидают вызывающего объекта. |
| String toString() | Возвращает строку, описывающий объект. |
| void wait() | Ожидает другого потока выполнения. |
| void wait(long millis, int nanos) | Ожидает другого потока выполнения. |
| Object clone() | -Создаёт новый объект, не отличающий от клонируемого объекта. |

Фабричный метод.

Фабричный метод – это метод создающий объект. Механизм построен на скрытие конструктора. Обычно фабричный метод возвращает новый объект с помощью return.

В общем, скрытие конструктора от пользователя хорошая методика: можно подменить реализацию, добавить более сложное кэширование. Если убрать фабричные методы программа становится менее гибкой.

Анонимный класс.

Анонимные классы активно используются для реализации нескольких методов и создания собственных методов объекта или побочного действия.

Анонимный класс - это локальный класс без имени. Можно объявить анонимный (безымянный) класс, который может расширить (extends) другой класс, использовать фабричный метод или реализовать (implements) интерфейс. Объявление такого класса выполняется одновременно с созданием его объекта посредством оператора new.

Примечания:

Анонимный класс можно объявить в интерфейсе.

Такой класс не может иметь конструктор.

Перечисления.

*Перечисление* это список именованных констант. Он содержит *константы* *перечислимого* *типа* или *самотипизированными* *константами*.

Нужно учитывать то что перечисление это класс и может иметь конструкторы, методы и поля. Кроме того каждое перечисление является подклассами Enum и может расширять/расширяться от другого перечисления.

Для перечислений используется ключевое слово enum. В начале перечисления всегда объявляется константы и лишь потом если это надо конструкторы, поля и методы. Пример простого перечисления:

enum Apple {

White\_Filling, Melba, Mantet

}

Так как перечисление это класс то и переменную перечислимого типа объявляется схожим типом. Пример создания переменной Apple:

Apple ap = Apple.Melba

Нужно отметить можно не создавать левую часть и использовать только правую часть объявления переменной. Ниже представлен пример использования Apple:

public class Main {

public static void main(String[] args) {

Apple ap = Apple.Melba;

System.out.println("Переменная ap содержит:"+ap);

ap = Apple.Mantet;

if(ap == Apple.Mantet)

System.out.println("ap содержит Apple.Mantet.");

switch (ap){

case Melba:

System.out.println("Мелба среднего размера");

break;

case Mantet:

System.out.println("Матнент округло-продулговатый");

break;

case White\_Filling:

System.out.println("Белый налив белого цвета");

break;

}

}

}

Перечисление = класс.

До этого был представлен простое перечисление. Он содержал только список констант, но кроме это может конструкторы, по которым будут создаваться константы перечисления, методы для работы с константами перечисления и поля. Ниже представлен пример более сложного перечисления:

package myPackage;

public class Main {

public static void main(String[] args) {

Transport t1 = Transport.HORSE;

Transport t2 = Transport.BOAT;

System.out.println("t1 содержит: "+t1+", "+

t1.getSpeed()+" "+t1.getWeight());

System.out.println("t2 содержит: "+t2+", "+

t2.getSpeed()+" "+t2.getWeight());

if(!t1.equalsSpeed(t2) == t2.equalsWeight(t1))

System.out.println("Они равны");

else if(t1.equalsSpeed(t2))

System.out.println("Они равны по скорости");

else if(t1.equalsWeight(t2))

System.out.println("Они равны по массе");

else

System.out.println("Они не равны");

}

}

enum Transport{

BOAT(30), CARE(120, 1.1), BUS(60),

AIRPLANE(350, 250.53), TRAIN(200), HORSE(25, 0.090);

private int speed;

private double weight;

Transport(int speed){

this.speed=speed;

weight = 1.3;

};

Transport(int speed, double weight){

this.speed=speed;

this.weight=weight;

}

public int getSpeed() {

return speed;

}

public double getWeight() {

return weight;

}

public boolean equalsSpeed(Transport transport){

return weight == transport.speed;

}

public boolean equalsWeight(Transport transport){

return weight == transport.weight;

}

}

Класс Enum

Класс Enum определяет методы для работы с перечислениями. Кроме стандартных equals (), compareTo () и прочих в нем определяется ordinal (), values (), valueOf (), name() и getDeclaringClass()

Метод ordinal () возвращает индекс вызывающего элемента в списке. Метод name() возвращает имя константы. Благодаря методу getDeclaringClass () можно получить Class <> c перечислением.

Методом values () можно получить массив всех констант перечислений. Но методом valueOf() получить один константу, если она есть.

Ниже представлен пример выше описанных методов с выше написанным перечислением:

package myPackage;

public class Main {

public static void main(String[] args) {

Transport t1=Transport.HORSE;

Class<Transport> c = t1.getDeclaringClass();

System.out.println("Имя перечисления: "+c.getName());

System.out.println("Имя t1: "+t1.name());

System.out.println("Индекс t1: "+t1.ordinal());

Transport [] transports=Transport.values();

for (Transport t : transports) {

System.out.println(t.name());

}

System.out.println("valueOf(\"BOAT\") -> "+Transport.valueOf("BOAT"));

}

}

enum Transport{

BOAT(30), CARE(120, 1.1), BUS(60),

AIRPLANE(350, 250.53), TRAIN(200), HORSE(25, 0.090);

private int speed;

private double weight;

Transport(int speed){

this.speed=speed;

weight = 1.3;

};

Transport(int speed, double weight){

this.speed=speed;

this.weight=weight;

}

public int getSpeed() {

return speed;

}

public double getWeight() {

return weight;

}

public boolean equalsSpeed(Transport transport){

return weight == transport.speed;

}

public boolean equalsWeight(Transport transport){

return weight == transport.weight;

}

}

Интерфейсы

Интерфейсы указывают классу, что они должны делать, но как они должны делать. Это обозначает, что когда класс реализует интерфейс, он должен переопределить все его методы. Интерфейсы объявляются с помощью ключевого слова interface.

Синтаксис интерфейсов.

Полное объявление интерфейсов выглядит так:

*доступ* interface *имя*{

*возвращаемый\_тип имя\_метода\_1*(*список\_параметров*);

*возвращаемый\_тип имя\_метода\_1*(*список\_параметров*);

*тип имя\_ завершенной\_переменной\_1* = *значение;*

*тип имя\_ завершенной\_переменной\_N* = *значение;*

}

Реализации интерфейсов.

Основная идея интерфейсов заключается в его реализации. Так он обеспечивает полиморфизм и для этого определен свой синтаксис:

*доступ* class *имя\_класса* [extends *суперкласс*]

implements *имя\_интерфейса\_1*, *имя\_интерфейса\_N*{

@Override

public *возвращаемое\_значение* *имя\_метода*(*список\_параметров*) {

//Тело метода

}

@Override

public *возвращаемое\_значение* *имя\_метода\_N*(*список\_параметров*) {

//Тело метода

}

}

Следует отметить, что сигнатура определяемого метода должна соответствовать сигнатуре оригинала. Также все методы, определяемые в интерфейсе, когда они будут определяться должны быть public.

Ниже приведен пример интерфейса и его реализаций:

package myPackage;

public class Main {

public static void main(String[] args) {

MyHello hello=new MyHello();

hello.hello();

hello.helloYou("Каркарыч");

HelloEng helloEng=new HelloEng();

helloEng.hello();

helloEng.helloYou("Кузинатра");

}

}

interface Hello{

void hello();

void helloYou(String name);

}

class MyHello implements Hello{

@Override

public void hello() {

System.out.println("Привет");

}

@Override

public void helloYou(String name) {

System.out.println("Привет, "+name);

}

}

class HelloEng implements Hello{

@Override

public void helloYou(String name) {

System.out.println("Hello, "+name);

}

@Override

public void hello() {

System.out.println("Hello");

}

}

Доступ к реализациям через ссылки на интерфейсы.

Можно объявить ссылку на интерфейс и в правой части создавать объекты классов, которые реализует сам интерфейс. Такое уже проделывали с суперклассами и подклассами. Ниже приведен пример переделанный выше программы под доступ через ссылки интерфейса:

package myPackage;

public class Main {

public static void main(String[] args) {

Hello hello = new HelloEng();

hello.hello();

}

}

interface Hello{

void hello();

void helloYou(String name);

}

class MyHello implements Hello{

@Override

public void hello() {

System.out.println("Привет");

}

@Override

public void helloYou(String name) {

System.out.println("Привет, "+name);

}

}

class HelloEng implements Hello{

@Override

public void helloYou(String name) {

System.out.println("Hello, "+name);

}

@Override

public void hello() {

System.out.println("Hello");

}

}

Вложенные интерфейсы.

*Вложенные интерфейсы* различаются от обычных интерфейсов тем, что он может быть public, protect, private или по умолчанию. Интерфейс можно вложить в класс так и в интерфейс. Для использования вложенного интерфейса нужно уточнить его название через точку. Ниже приведен пример использования вложенного интерфейса:

package myPackage;

public class Main {

public static void main(String[] args) {

A.NestedIF nif=new B();

if(!nif.isNegative(12))

System.out.println("Число положительное");

else

System.out.println("Число от");

}

}

class A{

interface NestedIF{

boolean isNegative(int val);

}

}

class B implements A.NestedIF{

@Override

public boolean isNegative(int val) {

return val < 0 ? true : false;

}

}

Абстрактные классы

*Абстрактный класс* позволяет определить несколько методов интерфейса, а все остальные в последующем. Такие классы объявляются с помощью ключевого слова abstract. Ниже приведен пример абстрактного класса:

package myPackage;

public class Main {

public static void main(String[] args) {

A.NestedIF nif=new C();

if(!nif.isNegative(12))

System.out.println("Число положительное");

else

System.out.println("Число отрицательное");

System.out.println(nif.Devil(12, 6));

}

}

class A{

interface NestedIF{

boolean isNegative(int val);

int Devil(int val, int num);

}

}

abstract class B implements A.NestedIF{

@Override

public boolean isNegative(int val) {

return val < 0 ? true : false;

}

}

class C extends B{

@Override

public int Devil(int val, int num) {

if(num==0)

return -1;

return val / num;

}

}

Переменные в интерфейсах.

В интерфейсах все переменные являются константами. Класс реализующий интерфейс с переменными просто попадает под область действия констант. Ниже приведен пример использования переменных в интерфейсах:

package myPackage;

import java.util.Random;

public class Main {

public static void main(String[] args) {

Question question=new Question();

AskMe ask=new AskMe();

ask.answer(question.ask());

ask.answer(question.ask());

ask.answer(question.ask());

ask.answer(question.ask());

ask.answer(question.ask());

}

}

interface SharedConstants{

int NO=0;

int YES=1;

int MAYBE=2;

int LATER=3;

int SOON=5;

//int INT

int NEVER=6;

}

class Question implements SharedConstants{

Random random=new Random();

public int ask() {

int sum= (int) (100 \* random.nextDouble());

if(sum<30)

return NO;

else if(sum<60)

return YES;

else if(sum<75)

return LATER;

else if(sum <98)

return SOON;

else

return NEVER;

}

}

class AskMe implements SharedConstants{

public void answer(int result){

switch (result){

case NO:

System.out.println("Нет");

break;

case YES:

System.out.println("Да");

break;

case MAYBE:

System.out.println("Возможно");

break;

case LATER:

System.out.println("Позже");

break;

case SOON:

System.out.println("Вскоре");

break;

case NEVER:

System.out.println("Никогда");

break;

}

}

}

Методы по умолчанию

До этого говорилось в интерфейсе создать новый метод это нарушит же существующий код. Также в интерфейсе может определяться методы для изменяемых и неизменяемых коллекций и метод remove() надо будет определять лишь для изменяемых коллекций. Все эти задачи и их аналоги ставят в тупик.

Для решений этих проблем придуманы *методы по умолчанию* или методы расширения. Они обозначаю, то что в интерфейсе будет метод с уже определенным метод, но при желании которого можно переопределить метод.

Методы по умолчанию схожи с обычными методами, но объявляются с ключевым словом default. Их особенность, в том что они не нарушают принцип «У интерфейса нет состояния».

Ниже представлен пример метода по умолчанию:

package myPackage;

import java.util.Random;

public class Main {

public static void main(String[] args) {

MyIFImp imp=new MyIFImp();

System.out.println(imp.getNumber());

System.out.println(imp.getString());

}

}

interface MyOF{

int getNumber();

default String getString(){

return "Это строка из метода getString()";

}

}

class MyIFImp implements MyOF{

@Override

public int getNumber() {

Random rand=new Random();

return (int) rand.nextInt(12);

}

@Override

public String getString() {

return "Это строка из MyIFImp";

}

}

Статические методы в интерфейсе.

В интерфейсе можно объявить статические методы с реализацией. Отличаться от статических методов из классов он не будет.

Для его вызова определен следующий синтаксис:

*имя\_интерфейса*.*имя\_статического\_метода*()

Ниже представлена переделанная верхняя программа, но с добавленным статистический методом getDefaultNumber() :

package myPackage;

import java.util.Random;

public class Main {

public static void main(String[] args) {

MyIFImp imp=new MyIFImp();

System.out.println(imp.getNumber());

System.out.println(imp.getString());

System.out.println(MyOF.getDefaultNumber());

}

}

interface MyOF{

int getNumber();

default String getString(){

return "Это строка из метода getString()";

}

static int getDefaultNumber(){

return 0;

}

}

class MyIFImp implements MyOF{

@Override

public int getNumber() {

Random rand=new Random();

return (int) rand.nextInt(12);

}

@Override

public String getString() {

return "Это строка из MyIFImp";

}

}

Пакеты в java и тип protect.

Пакеты.

В java неотъемлемой частью программирования являются пакеты. Пакет – это директория содержащая совокупность.

Пакеты состоят из:

* package – ключевое слово, обозначает пакет. Любой пакет имеет имя.
* import – импортирует классы из одного пакета в другой пакет. Классы импортируется для использования их самих.

Пакеты не разложены в иерархии, а наоборот - хаотично. К такому выводу пришли, так как наследование пакетов, суперпакетов и подпакетов – нет.

В Java существуют правила именования пакетов и их классов.

1) Пакет:

1.1) Все буквы маленькое. Доменное имя автора или компании пишется впереди.

1.2)Запрещается использовать имена: java и javax.

2) Класс:

2.1) Все имена с большой буквы.

Пример:

package ABC;

class Aclass {

int adress;

Aclass(int adress){

this.adress=adress;

}

char a = "a";

void printA(){

System.out.println("Значение символа a:"+a);

}

}

class Bclass {

int ddress;

Bclass(int ddress){

this.ddress=ddress;

}

char b = "b";

void printb(){

System.out.println("Значение символа b:"+b);

}

}

class Cclass {

int cdress;

Cclass(int cdress){

this.cdress=cdress;

}

char c = "c";

void printA(){

System.out.println("Значение символа c:"+c);

}

}

package MainClass;

import ABC.Aclass.\*;

class Main{

public static void main(String [] args){

Aclass aclass = new Aclass(234);

aclass.printA();

}

}

Protected.

*Protected* или защищенный тип – обозначает то, что помеченный ими элементами не можно видеть, использовать и т.п. в других пакетов. Помечаются ключевым словом protected.

Модификатора доступа.

Таблица модификаторов доступа.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Тип | Класс | Пакет | Подкласс | Сторонние пакеты. |
| public | Да | Да | Да | Да |
| protected | Да | Да | Да | Нет |
| default | Да | Да | Нет | Нет |
| private | Да | Нет | Нет | Нет |
|  |  |  |  |  |

Abstract и final.

Применение abstract и final:

* Abstract метод - метод который необходимо переопределить
* final класс – класс который нельзя унаследовать.
* final метод – метод который нельзя переопределить.

Статический импорт.

Статистический импорт – импорт статистических элементов из каково-то класса. Если импортировать метод с помощью статистического импорта, то можно использовать их без квалификаторов. Например:

import static java.util.Math.\*;

Исключение и их обработка.

*Исключение* – это ненормальная ситуация, возникающая во время выполнения кода, т.е. ошибка. Обработка событий является важной чертой ООП и вообще программирования на Java.

Основы обработки исключений.

Исключение - это объект, который описывает ошибку. Она *генерируется* во время выполнения метода, а после *ловится* и *обрабатывается*.

Для того чтоб обработать исключение используется пять операторов: try, catch, throws, throw, finally. Если вкратце, то они работают следующим образом. В блоке try находятся объекты, которые генерируют исключения. Оператор catch ловит исключение и его обрабатывает. В операторе throw перечисляются исключения генерируемые в методе. Оператор throw генерирует исключение. А оператор finally содержит код, который должен быть выполненным после try.

В общем, получается следующая картина синтаксиса:

try{

// Блок кода, в котором отслеживают ошибку

}catch(*тип\_исключения* exOb){

// Обработчик исключения *тип\_исключения*

}catch (*тип\_исключения\_N* exOb){

// Обработать исключение *тип\_исключения\_N*

}

Здесь *тип\_исключения* обозначает конкретное исключение, которое должно быть обработано.

Типы исключений

Все исключения являются подклассами класса Throwable. Класс Throwable находится на вершине *иерархии исключений*. У него определенно два подкласса Exception и Error.

Класса Exception – служит суперклассом уже для всех исключений, которые должны обрабатываться в прикладном коде. У него определен важный класс RuntimeException, который автоматически создается во время выполнения вашей программы. К ним можно отнести деление на нуль, ошибочная индексация массива и т.п.

Класс Error – служит суперклассом уже для всех исключений, которые не должны обрабатываться в прикладной программе. Это исключения исполняющей среды.

Необрабатываемые исключения.

Нижний код предоставляет пример типичной ошибки деления на нуль:

public class Main {

public static void main(String[] args){

int i;

int d=0;

i= 46 / d;

System.out.println("Результат вычисления: "+i);

}

}

В выводе можно будет увидеть:

Exception in thread "main" java.lang.ArithmeticException: / by zero

at myPackage.Main.main(Main.java:11)

Здесь написано, что в выполнения метода main произошла ошибка java.lang.ArithmeticException и пояснение к ошибке. Далее следует, ссылка на место где произошла ошибка.

Чтоб не происходило, исключение его следует обработать. Как показано ниже:

package myPackage;

public class Main {

public static void main(String[] args){

int i;

try {

int d = 0;

i = 46 / d;

System.out.println("Результат вычисления: " + i);

}catch (ArithmeticException exc){

System.out.println("Результат вычисления: Нет ответа ");

i =-1;

}

}

}

Сейчас стоит отметить, что если исключение не было обработано, его обработает *обработчик по умолчанию*. Оно выведет имя ошибки, и место где оно произошло.

Обработка исключений

Для обычно ручного отлавливания и обработки используется блок try и обработчик исключений catch. Они образуют единую структуру. Ранее уже было показано использование try-catch, поэтому можно рассказать о других случаях.

Один и тоже код может кидать две и более исключения сразу. Поэтому придумали нескольких операторов catch в связке с try. Ниже представлении пример нескольких операторов catch:

package myPackage;

public class Main {

public static void main(String[] args){

int i;

try {

i = 47 / 0;

System.out.println("Результат вычисления: " + i);

int[] arr = {1};

arr[66] = 56;

System.out.println("Содержимое массива: ");

for (int i1 : arr) {

System.out.println(i1);

}

}catch (ArithmeticException exc1){

System.out.println("Результат вычисления: Нет решения");

i=0;

}catch (ArrayIndexOutOfBoundsException exc2){

System.out.println("Было осуществлена операция выхода из массива");

}

}

}

Оператор throw

Для того чтоб сгенерировать исключение применяется оператор throw. Синтаксис оператора:

throw new *исключение*

где *исключение* должен быть экземпляр Throwable или его подкласса. Если так сгенерировать ошибку, тогда поток останавливает все последующие

Ниже представлен пример генерации ошибок:

package myPackage;

public class Main {

static void demoproc(){

System.out.println("Генерирование исключений");

throw new NullPointerException();

}

public static void main(String[] args){

try {

demoproc();

}catch (NullPointerException exc){

System.out.println("Поймано исключение: "+exc);

}

}

}

Оператор throws.

Если метод может сгенерировать исключение, которые не может сам его обработать исключение следует использовать оператор throws. В нем перечисляются типы исключения, кроме Error и RuntimeException и их подклассы.

Ниже представлен пример объявление метода, в котором перечисляются все исключение ими генерируемым оператор throws.

*тип имя\_метода*() throws *исключении\_1*, *исключение\_N* {

// Тело метода

}

Ниже представлен пример использования оператора throws:

package myPackage;

public class Main {

static void demoproc(){

System.out.println("Генерирование исключений");

throw new NullPointerException();

}

public static void main(String[] args){

try {

demoproc();

}catch (NullPointerException exc){

System.out.println("Поймано исключение: "+exc);

}

}

}

Оператор finally.

Если генерируется исключение, в методе может произойти координальное и изменение в выполнении. Для непредвиденных обстоятельств есть оператор finally.

Оператор finally образует блок кода, который будет сразу следовать после выполнения операторов try-catch. Ниже представлен пример его использования:

package myPackage;

public class Main {

static void procA(){

try {

System.out.println("В теле метода procA()");

throw new RuntimeException("Демонстрация");

}finally {

System.out.println("Блок кода в операторе finally в методе procA()");

}

}

static void procB(){

try {

System.out.println("В теле метода procB()");

return;

}finally {

System.out.println("Блок кода в операторе finally в методе procB()");

}

}

static void procC(){

try{

System.out.println("В теле метода procC()");

}finally {

System.out.println("Блок кода в операторе finally в методе procC()");

}

}

public static void main(String[] args){

try {

procA();

}catch (Exception e){

System.out.println("Исключение перехвачено");

}

procB();

procC();

}

}

Список основных исключений.

Непроверяемые:

OutOfMemoryError – вызывается когда не может произойти выделение памяти (new). Происходит из-за заполнения.

InternalError - возникает когда программа использует код который по логике не должен обрабатываться.

ThreadError – возникает когда один поток убивает другой поток с помощью метода thread.stop ().

Проверяемые:

IOException – возникает в случае работы с ввод\выводом.

ArithmeticException – возникает в случае работы арифметикой.

NullPointerException – возникает в случае неправильной работе с нулями.

ArrayIndexOutOfBoundsException - возникает в случае выхода из массива.

ArrayStoreException – возникает когда, была сделана попытка сохранить неправильный тип объекта в массив объектов.

IllegalArgumentException – когда был передан аргумент противоречащий чему либо, не кидается виртуальной машинной, но все равно стоит использовать.

IllegalStateException – используется когда метод не находится в соответствующем состоянии для запрошенной операции, исключение не кидается виртуальной машиной.

UnsupportedOperationException – кидается когда используется без реализации.

IndexOutOfBoundsException – схож с ArrayIndexOutOfBoundsException, но используется для списков и т.п.

Создание собственных исключений.

Есть ряд уже собственных исключений, но есть возможность делать свои исключения. Для этого следует создать класс, расширяющий Exception или Error. Ниже представлен пример создания собственных исключений:

package myPackage;

public class Main {

static void compute(int a) throws MyException{

System.out.println("В теле метода compute("+a+")");

if(a >10)

throw new MyException(a);

System.out.println("Нормальное завершение");

}

public static void main(String[] args){

try{

compute(1);

compute(23);

}catch (MyException exc){

System.out.println("Перехвачено "+exc);

}

}

}

class MyException extends Exception{

private int a;

MyException(int detail){

this.a=detail;

}

@Override

public String toString() {

return "MyException["+a+"]";

}

}

Цепочки исключений.

Изначальной исключением ArithmeticException может быть ошибка ввода-вывода IOException. Также исключение NullPointerException, может быть произошла из-за ArithmeticException.

Для таких случаев есть *цепочки исключений*. Они поддерживаются двумя конструкторами в классе Throwable:

Throwable(Throwable *причина\_исключения*)

Throwable(String *сообщение*, Throwable *причина\_исключения*)

где *причина\_исключения* это исключение признана, как возбуждающая данное исключение. Но даже если причина исключения указана в конструкторе следует воспользоваться методом initCause() и передать первоисточник. Полное объявление метода:

Throwable initCause(Throwable *причина\_исключения*)

Если нужно возвратить причину можно воспользоваться методом getCause(), с полным объявлением:

Throwable getCause()

В приведенной ниже программе демонстрируется применение цепочек:

package myPackage;

public class Main {

static void demoProc(){

NullPointerException exc= new NullPointerException();

exc.initCause(new ArithmeticException("Причина"));

throw exc;

}

public static void main(String[] args){

try{

demoProc();

}catch (NullPointerException exc){

System.out.println("Перехвачено "+exc);

System.out.println("Причина исключения: "+exc.getCause());

}

}

}

Многократный перехват.

*Многократный перехват* - позволяет два разных исключений обработать, как одно в операторе catch. Это удобно, когда два исключения в своем решении используют одинаковый код.

Для этого все исключение следует писать через знак короткого ИЛИ (|). Ниже представлен синтаксис многократного перехвата:

try{

// Блок кода

} catch(*исключение\_1* | *исключение\_2* | *исключение\_N* *имя*){

// Обработка

}

Ниже представлен пример использования многократного перехвата:

package myPackage;

public class Main {

public static void main(String[] args){

int i=12, b=0;

int [] vals={1, 2, 3};

try {

int result = i / b;

vals[12] = 112;

}catch (ArithmeticException | ArrayIndexOutOfBoundsException exc){

System.out.println("Перехвачено "+exc);

}

}

}

Generic

К примеру, отдельный стек может оперировать только строками и это неудобно, ведь по сути операции могут быть одинаковыми и для Integer. Класс коллекция не будет определяться для Integer, Short, Long, Double, Float, Character и API классов и ваших объектов. А когда семантика класса распределяется стразу на несколько типов, к примеру, Number и должен оперировать Integer и Long. Все эти ситуации усугубляют программирование.

Благодаря generics(или обобщений) можно составить алгоритм, который позволит одному стеку оперировать Integer и Thread, поставляя к объекту необходимый тип. Класс коллекции можно объявить обобщенным и хранить только Callable.

Конечно, можно использовать Object и также поставлять значения, но тогда можно поставить к раннему Integer String и получить небезопасный кусок кода, который к тому же придется все время приводить к желаемому типу.

Что такое обобщение?

*Обобщение* – это параметризованный тип. Он позволяет создавать классы параметры оперируемых данных, которых передаются в качестве аргумента. Используя обобщения можно создать класс, оперирующий сразу несколькими типами.

Простой пример обобщения.

Сразу же стоит привести пример обобщения:

package myPackage;

public class Main{

public static void main(String[] args){

MyGen<Integer> iObj=new MyGen<>(23);

iObj.showType();

System.out.println("Содержимое: "+iObj.getOb());

// Второй объект типа MyGen, который оперирует строками

MyGen<String> sObj=new MyGen<>("Оперируемые данные");

sObj.showType();

System.out.println("Содержимое: "+sObj.getOb());

}

}

class MyGen<T>{

T ob;

MyGen(T ob){

this.ob=ob;

}

T getOb(){

return ob;

}

void showType(){

System.out.println("Тип "+ob.getClass().getName());

}

}

Результат сразу виден на лицо. Один класс смог оперировать сразу Integer и String. А теперь посмотрим, как все это получилось.

Сначала объявляется обобщенный класс MyGen<>. Объявление:

class MyGen<T> {

То, что класс обобщен мы указываем благодаря ромбикам <>. В него мы вставляем любую заглавную букву, т.е. параметризованный тип. Вообще в обществе определенными буквами помечают определенный тип. К примеру, ключ помечают K.

Следующей строке создается объект типа T. Его тип будет в будущем поставлен.

T ob;

Конструктор мы передаем значение, которое в будущем поставим. Это и есть из одних достоинств generics – защита типов. Это значит, что там не может быть вдруг Long.

Следует присмотреться к созданию объекта и его синтаксису:

MyGen<Integer> iObj=new MyGen<>(23);

В данном случае ромбы ставятся после имя класса и ему передают тип и до скобок.

Примечания относительно обобщений:

У обобщений есть ряд примечаний:

* Обобщенные типы действуют со ссылками, а не примитивами.
* Обобщенные типы различаются по аргументам.
* Обобщение одного типа нельзя присвоить другому с другим типом.

Обобщенный класс двумя параметрами.

Можно создавать классы с двумя, тремя и так далее типами. Для этого в объявление в ромбике написать сразу два типа. Они ничем не отличаются с классами одним параметром. Ниже представлен вариант использования обобщенного класса двумя классами:

package myPackage;

public class Main{

public static void main(String [] args) {

TwoGen<String, Integer> siObj=new TwoGen<>(

"Содержимое 1", 123);

siObj.showType();

System.out.println("Значения: "+siObj.getOb()+"" +

", "+siObj.getOd1());

TwoGen<Integer, Long> ilObj=new TwoGen<>(12, 12l);

ilObj.showType();

System.out.println("Содержимое: " +ilObj.getOb()+

", "+ilObj.getOd1());

}

}

class TwoGen<T, V>{

private T ob;

private V ob1;

TwoGen(T ob, V od1){

this.ob=ob;

this.ob1=od1;

}

T getOb(){

return this.ob;

}

V getOd1(){

return this.ob1;

}

void showType(){

System.out.println("Тип T "+ob.getClass().getName());

System.out.println("Тип V "+ob1.getClass().getName());

}

}

Общая форма обобщения:

Теперь можно примести синтаксис объявления обобщенного класса:

class *имя\_класса* <*список\_параметров\_типов*> {

Объявление ссылки на обобщенный класс:

*имя\_класса* <*список\_аргументов\_типов*> *имя* =

new *имя\_класса* <> (*список\_аргументов\_констант*);

Ограниченные типы.

Давайте напишем программу, которая просто вычисляла среднее арифметическое массива. Вроде в этом нет ничего трудного. Но как мы знаем подставить можно любой тип. И это влечет то, что можно подставить, к примеру, Runtime и получить ошибку.

Поэтому следует *ограничить* тип классом Number. Это означает, что теперь можно подставить только Number и его подкласса(Integer, Long и т.п.).

Для ограничения типа используется ключевой extend, поэтому тип можно в своем роде расширить. Ниже представлена форма объявления обобщенного класса с ограниченным типом:

class *имя\_класса* <T extends *суперкласс*> {

При объявлении ссылки можно не писать extends *суперкласс*, а просто поставить тип. Также это можно сочетать и с другими типами.

Теперь ниже представлен конечный вариант программы находящая средне арифметическое число:

package myPackage;

public class Main{

public static void main(String [] args) {

Integer [] idate= {1, 2, 3, 5, 4};

Stats<Integer> iObj=new Stats<>(idate);

System.out.println("Среднее значение: "+iObj.average());

Float [] fdate= {1F, 3F, 6F, 2F, 4F};

Stats<Float> fObj=new Stats<>(fdate);

System.out.println("Среднее значение: "+fObj.average());

}

}

class Stats<T extends Number> {

T [] data;

Stats(T[] data){

this.data=data;

}

double average() {

double sum=0;

for(int i=0; i<data.length; i++)

sum +=data[i].doubleValue();

return sum /data.length;

}

}

Следует объявить то, что можно накладывать не только классы, но и интерфейсы. Если есть только интерфейс его можно писать сразу через extends, а если интерфейс и класс следует применять следующий синтаксис:

class *имя\_класса* <T extends *суперкласс* & *интерфейс* & *интерфейс\_N*>

Таким образом, T должен быть подклассом *суперкласс* и реализовать *интерфейс*.

Применение метасимвола.

Типовая безопасность это прекрасно. Но было бы неплохо внести метод sameAvg(), который узнавал, равен ли результат среднеарифметического одного класса с другим. Вроде бы легко, но он будет работать с такими же типами. Метод:

boolean sameAvg(Stats<T> obj){

if(average() == obj.average())

return true;

return false;

}

Вроде должен сработать, но при работе с обобщением разным типом вылетает ошибка. А почему? Мы не знаем, какой конкретно тип будет у обобщенного класса и еще мы указываем его тип T, что значит, он будет такой же тип как и сравнимого.

Выходом из ситуации служит метасимволили маска. *Маска* – указывает на неизвестный тип и обозначается знаком ?. Маска не изменяет тип создаваемых объектов. Он *совпадает* с любым достоверным типом.

Ниже приведен пример верхний программы с небольшими изменениями:

package myPackage;

public class Main{

public static void main(String [] args) {

Integer [] idate= {1, 2, 3, 5, 4};

Stats<Integer> iObj=new Stats<>(idate);

System.out.println("Среднеарифметическое: "+iObj.average());

Float [] fdate= {1F, 3F, 6F, 2F, 3F};

Stats<Float> fObj=new Stats<>(fdate);

System.out.println("Среднеарифметическое: "+fObj.average());

System.out.println("Равны: "+iObj.sameAvg(fObj));

}

}

class Stats<T extends Number> {

T [] data;

Stats(T[] data){

this.data=data;

}

double average() {

double sum=0;

for(int i=0; i<data.length; i++)

sum +=data[i].doubleValue();

return sum /data.length;

}

boolean sameAvg(Stats<?> obj){

if(average() == obj.average())

return true;

return false;

}

}

Ограниченные метасимволы.

Маска также может быть ограничена. Для примера рассмотрим ветвь, которая инкапсулирует координаты:

class TwoD {

int x, z;

TwoD(int x, int z){

this.x=x;

this.z=z;

}

}

class TreeD extends TwoD{

int y;

TreeD(int x, int z, int y){

super(x, z);

this.y=y;

}

}

class FourD extends TreeD{

int t;

FourD(int x, int z, int y, int t) {

super(x, z, y);

this.t=t;

}

}

И теперь нам нужен массив, который хранил координаты и по возможности определяющий ряд методов для управления:

class Data<T extends TwoD> {

T [] nums;

Data(T [] array){

this.nums=array;

}

// Методы

}

Допустим, нужен метод, который возвращал X Z. Это просто, потому что тип ограничен TwoD и может использовать X Z. Но если требуется вывести X Z и Y из TreeD? Ведь Data<TwoD> может иметь только X Y. Для этих целей может подойти *ограниченный метасимвол аргумент*. Он позволяет указать верхнюю или нижнюю границу для аргумента типа.

Полная программа:

package myPackage;

public class Main{

public static void main(String [] args) {

TwoD [] td = {

new TwoD(1, 56),

new TwoD(2, 123),

new TwoD(123, 56)

};

Data tdData=new Data(td);

tdData.showXY(tdData);

FourD [] fd= {

new FourD(23, 56, 3, 12),

new FourD(1,1,1,1),

new FourD(34, 12, 23, 34)

};

Data fdData=new Data(fd);

fdData.showXY(fdData);

fdData.showXYZ(fdData);

fdData.showAll(fdData);

}

}

class TwoD {

int x, z;

TwoD(int x, int z){

this.x=x;

this.z=z;

}

}

class TreeD extends TwoD{

int y;

TreeD(int x, int z, int y){

super(x, z);

this.y=y;

}

}

class FourD extends TreeD{

int t;

FourD(int x, int z, int y, int t) {

super(x, z, y);

this.t=t;

}

}

class Data<T extends TwoD> {

T [] nums;

Data(T [] array){

this.nums=array;

}

void showXY(Data<? extends TwoD> obj){

System.out.println("Координаты X Y: ");

for(int i=0; i<obj.nums.length; i++)

System.out.println(obj.nums[i].x+" "+

obj.nums[i].z);

System.out.println();

}

void showXYZ(Data<? extends TreeD> obj){

System.out.println("Координаты X Z Y:");

for(int i=0; i<obj.nums.length; i++)

System.out.println(obj.nums[i].x+" "+

obj.nums[i].y+" " +

obj.nums[i].z);

System.out.println();

}

void showAll(Data<? extends FourD> obj){

System.out.println("Координаты X Z Y T: ");

for(int i=0; i<obj.nums.length; i++)

System.out.println(obj.nums[i].x+" "+

obj.nums[i].z+" "+

obj.nums[i].y+" "+

obj.nums[i].t);

}

}

В общем, чтобы установить верхнюю границу для метасимвола, следует воспользоваться приведенной ниже формой метасимвольного выражения:

<? extends *суперкласс*>

Имеется возможность указать нижнюю границу для маски, используя оператор super. Форма наложения нижней границы:

<? super *подкласс*>

На месте метасимвольного аргумента может быть только суперклассы для указанного *подкласса*. Это выражение исключает из себя *подкласс*.

Обобщенные методы

Было бы неплохо создать метод, который находил среднеарифметическое массива. Раннее уже была написана программа, но она много занимает. Благодаря обобщенным методам функционал прошлой программы можно сделать короче.

package myPackage;

import com.sun.istack.internal.NotNull;

public class Main{

public static void main(String [] args){

Stats stats=new Stats();

Integer [] array={1, 3, 5};

System.out.println(stats.average(array));

if(stats.isIn(2, array))

System.out.println("Значение есть");

else

System.out.println("Значения нет");

}

}

class Stats {

@NotNull

<T extends Number> double average(T [] obj){

double sum=0;

for(int i=0; i<obj.length; i++)

sum += obj[i].doubleValue();

return sum / obj.length;

}

<T extends Comparable<T>, V extends T> boolean isIn(T obj1, V [] y){

for(int i=0; i<y.length; i++)

if(obj1==y[i])

return true;

return false;

}

}

В общем, полное объявление параметризованных методов:

<*список\_параметров\_типа*> *возвращаемый\_тип имя\_метода* (*параметры*) {*тело*}

Обобщенные конструкторы

Их очень редко можно встретить, а особенно писать, но все же они есть. Ниже приведен не актуальный пример обобщенного конструктора:

package myPackage;

public class Main{

public static void main(String [] args){

GenCons test=new GenCons(120);

GenCons test1=new GenCons(12.4F);

test.showVal();

test1.showVal();

}

}

class GenCons{

private final double val;

<T extends Number> GenCons(T arg){

val=arg.doubleValue();

}

void showVal(){

System.out.println("val "+val);

}

}

Обобщенные интерфейсы

Представим нужно написать методы min(), max() для переопределения в других ситуациях (добиш, как в интерфейсе). Для этих целей подойдет обобщенный интерфейс. Ниже представлена программа:

package myPackage;

public class Main{

public static void main(String [] args){

Integer [] IntArr={1, 2, 3, 4, 8};

MyClass<Integer> intObj= new MyClass<>(IntArr);

System.out.println("Максимальное число "+intObj.max());

System.out.println("Минимальное число "+intObj.min());

}

}

interface MinMax<T extends Comparable<T>> {

T min();

T max();

}

class MyClass<T extends Comparable<T>> implements MinMax<T>{

T[] obj;

MyClass(T[] obj){

this.obj=obj;

}

@Override

public T max() {

T value=obj[0];

for(int i=1; i<obj.length; i++)

if(obj[i].compareTo(value) >0) value=obj[i];

return value;

}

@Override

public T min() {

T value=obj[0];

for(int i=1; i<obj.length; i++)

if(obj[i].compareTo(value) <0) value=obj[i];

return value;

}

}

Исходя, данной программы можно составить синтаксис объявить полный синтаксис объявления:

interface *имя*<*список\_параметра\_типа*> {

Когда полное реализации:

class *имя\_класса*<*список\_параметра\_типа*>

implements *имя\_интерфейса*<*список\_аргументов\_типа*>{

Иерархия обобщенных классов

Обобщенный суперкласс

Ниже представлен пример обобщенного суперкласса:

package myPackage;

public class Main{

public static void main(String [] args){

Gen2<Integer, String> isObj=new Gen2<>(123, "ТЕСТ");

isObj.showType();

System.out.println("Содержимое V: "+isObj.getV());

}

}

class Gen<T> {

T obj;

Gen(T obj){

this.obj=obj;

}

void showType(){

System.out.println("Тип суперкласса "+obj.getClass().getName());

}

}

class Gen2<T, V> extends Gen<T> {

V obj;

Gen2(T obj, V obj2) {

super(obj);

this.obj=obj2;

}

V getV(){

return obj;

}

}

Из примера можно понять, что тип суперкласса переносится на обобщенный подкласс. Здесь можно было сделать, чтоб у подкласса был только T.

Обобщенный подкласс

Можно создать иерархию, где у суперкласса есть обобщенный подкласс. Ниже представлена такая иерархия:

package myPackage;

public class Main{

public static void main(String [] args){

Gen<String, Integer> siObj=new Gen<>(

123, "Строчка", 12);

System.out.println("Сумма суперкласса: "+siObj.getSum());

System.out.println("Значение T: "+siObj.obj);

System.out.println("Значение V: "+siObj.obj2);

}

}

class NonGen {

private int sum;

NonGen(int sum) {

this.sum = sum;

}

public int getSum() {

return sum;

}

}

class Gen<T ,V> extends NonGen{

T obj;

V obj2;

Gen(int sum, T obj1, V obj2) {

super(sum);

this.obj=obj1;

this.obj2=obj2;

}

public T getObj() {

return obj;

}

public V getObj2() {

return obj2;

}

}

Переопределение метода в обобщенном классе.

И у обобщенного класса возможно также без проблем переопределить метод. Ниже приведен пример переопределения:

package myPackage;

public class Main{

public static void main(String [] args){

Gen<Integer> iObj=new Gen<>(123);

iObj.getObj();

MyGen<String> sObj=new MyGen<>("Object");

sObj.getObj();

}

}

class Gen<T> {

T obj;

Gen(T obj){

this.obj=obj;

}

public T getObj() {

System.out.println("Содержимое obj: "+obj);

return obj;

}

}

class MyGen<T> extends Gen<T>{

MyGen(T obj) {

super(obj);

}

@Override

public T getObj() {

System.out.println("Метод getObj из класса Gen2: "+obj);

return obj;

}

}

Сравнение типов в иерархии типов во время выполнения.

Ниже представлена программа показывающая сравнение типов во время выполнения:

package myPackage;

public class Main{

public static void main(String [] args){

Gen<Integer> iObj=new Gen<>(123);

Gen2<Integer> iObj2=new Gen2<>(456);

if(iObj instanceof Gen<?>)

System.out.println("iObj является экземпляром Gen<T>");

if(iObj2 instanceof Gen2<?>)

System.out.println("iObj2 является экземпляром Gen2<T> \n");

if(iObj2 instanceof Gen<?>)

System.out.println("iObj2 является экземпляром Gen<T>");

}

}

class Gen<T> {

T obj;

Gen(T obj){

this.obj=obj;

}

public T getObj() {

System.out.println("Содержимое obj: "+obj);

return obj;

}

}

class Gen2<T> extends Gen<T> {

Gen2(T obj) {

super(obj);

}

}

Стирание типов.

В связи с тем, что внесение изменений в синтаксис языка, т.е. компилятора сделает старый код не рабочим. Но с другой стороны обобщения были необходимыми и имели высший приоритет. Чтобы убить эти проблемы была предпринята использовать *стирание*.

Оказалось, во время выполнение выполнений обобщений нет. Их стирают и заменяют на Object. Если на тип поставлено ограничение ставится суперкласс, а когда не указано, заменяют на Object. После все значения приводятся к определенному типу, который определяется аргументом типа. Подход стирания обозначает отсутствие generics во время выполнения.

Сырой тип

*Сырой тип* или унаследованный код – это код, написанный до JDK 5, в котором появились первые обобщения. Его требуется поддерживать и не обязательно переделывать код (хотя это лучше), а использовать Object.

Ниже представлен пример кода на JDK 8:

class Gen<T>{

private T obj;

Gen(T obj){

this.obj=obj;

}

public T getObj() {

return obj;

}

}

Ниже представлена версия до JDK 5:

class NonGen{

private Object obj;

NonGen(Object obj){

this.obj=obj;

}

public Object getObj() {

return obj;

}

}

Не трудно понять, что в версии до обобщений защиты типов. В версии после обобщений она как раз есть. До обобщений было неудобно их использовать и приходилось постоянно приводит ради не допущения ошибки. После обобщений стало удобно, и перестала нужда приводить тип.

Ниже представлена полная программа:

package myPackage;

public class Main{

public static void main(String [] args){

Gen<Integer> iObj=new Gen<>(123);

System.out.println("Данные: "+iObj.getObj());

NonGen Obj=new NonGen("Сырой тип");

System.out.println("Данные: "+Obj.getObj());

NonGen otherObj=new NonGen(12);

Obj = otherObj;

System.out.println("Измененные данные: "+Obj.getObj());

}

}

class Gen<T>{

private T obj;

Gen(T obj){

this.obj=obj;

}

public T getObj() {

return obj;

}

}

class NonGen{

private Object obj;

NonGen(Object obj){

this.obj=obj;

}

public Object getObj() {

return obj;

}

}

Мостовые методы

Когда переопределенный метод из подкласса в результате стирания не совпадает с тем, что получено при стирании в суперклассе. В таких ситуациях применяют *мостовой метод,* он использует стирание суперкласса и вызывает соответствующий метод со стиранием суперкласса. Стоит, отметить такие методы генерируются компилятором.

Они живут на стадии байт-кода и поэтому, чтоб их увидеть следует применять рефлексию. Ниже представлен пример мостового метода:

package myPackage;

import java.lang.reflect.Method;

public class Main{

public static void main(String [] args){

Gen2 sObj = new Gen2("Мостовые методы");

sObj.getObj();

Class<?> cl=sObj.getClass();

Method [] methods=cl.getDeclaredMethods();

System.out.println("Методы: ");

for(int i=0; i<methods.length; i++)

System.out.println(methods[i]);

}

}

class Gen<T>{

T obj;

Gen(T obj){

this.obj=obj;

}

public T getObj() {

return obj;

}

}

class Gen2 extends Gen<String>{

Gen2(String obj) {

super(obj);

}

@Override

public String getObj() {

System.out.println("Вызван из класса Gen2: "+obj);

return obj;

}

}

Ошибки и ограничения на Generics.

Ограничения

У обобщений есть ряд ограничений. К ним относят:

* Получить тип по параметру типа нельзя.
* Тип не может быть статическим.
* Типы ограничены на массивы.
* Обобщенный класс не может расширять Throwable .

Ошибка неоднозначности.

Неоднозначность происходит, когда при стирании типов одинаковых типов их не могут различить. Ниже демонстрируется ситуация когда может произойти ошибка неоднозначности:

package myPackage;

public class Main{

public static void main(String [] args){

Gen<String, String> sObj=new Gen<>(

"String", "Generics");

sObj.setT("Строка");

System.out.println("Содержимое: "+sObj.getObj()+", "+sObj.getV());

}

}

class Gen<T, V>{

T obj;

V obj1;

Gen(T obj, V obj1){

this.obj=obj;

this.obj1=obj1;

}

public T getObj() {

return obj;

}

public V getV() {

return obj1;

}

public void setT(T obj) {

this.obj = obj;

}

public void setV(V obj1) {

this.obj1 = obj1;

}

}

Коллекции.

Краткий обзор коллекций.

*Каркас коллекции* framework - стандартизируют способы управлений группами объектов в прикладных программах. В java предусмотрены реализации стандартных коллекций, хотя можно создавать и свои.

*Коллекцию* можно представить, как динамический массив.

*Алгоритмы* оперируют коллекциями и о определенны, как статические методы в классе Collection. Такие методы доступны всем методам, не требуют реализации и представляют собой средства для манипулирования коллекциями.

Класс Collection от чего и начинаются, коллекции связан с интерфейсом *Iterable*.Интерфейс Iterable - определяет *итератор*, который обеспечивает способ *перебора содержимого коллекций*.

Помимо коллекций определен ряд интерфейсов и классов *отображений*. В них хранится *ключ-значение*. Но отображения не являются коллекциями, но при желании их можно обработать, как коллекции.

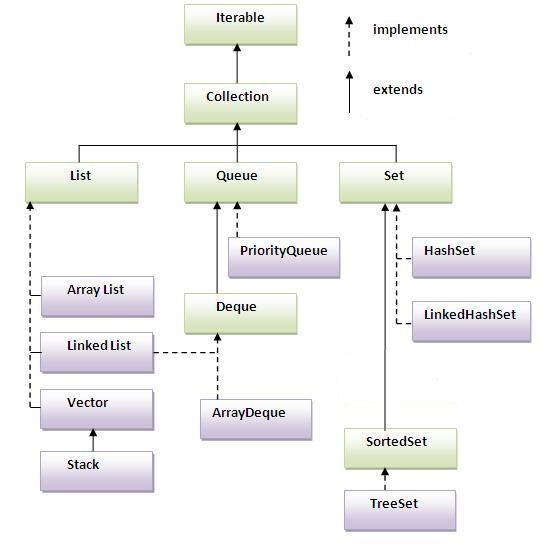
Все классы и интерфейсы находятся в пакете java.util. Коллекции просто предлагают выход из разных проблем.

Иерархия классов, интерфейсов коллекций и стандартные коллекции.

В java из стандартных коллекций в виде интерфейсов выделяют: *Set*, *List*, *Queue*. В целом каждый из них обозначает:

* List – список.
* Set –множество.
* Queue – очередь, расширяет интерфейс Deque, который в свою сторону обозначает двойную очередь.

Иерархия:



Интерфейс Collection.

Интерфейс collection лег в основу каркасов коллекций. Он должен быть реализованным всеми классами коллекций. Объявление:

interface Collection<E>.

где E обозначает тип объектов, которые будет содержать коллекция.

Интерфейс Collection расширяет Iterable. Это значит, что все коллекции можно перебирать в стиле for-each.

Методы данного интерфейса легли в основные методы у коллекций. Методы:

* boolean add(E e) -Обеспечивает, чтобы эта коллекция содержала указанный элемент.
* boolean addAll(Collection<? extends E> c) - Добавляет все элементы в указанную коллекцию к этой коллекции
* void clear() -Удаляет все элементы из этой коллекции
* boolean contains(Object o)- Возвращает true, если эта коллекция содержит указанный элемент.
* boolean containsAll(Collection<?> c) - Возвращает true, если эта коллекция содержит все элементы указанной коллекции.
* boolean equals(Object o) - Сравнивает указанный объект с этой коллекцией для равенства.
* int hashCode() -Возвращает значение хэш-кода для этой коллекции.
* boolean isEmpty() - Возвращает true, если эта коллекция не содержит элементов
* Iterator<E> iterator() -Возвращает итератор по элементам в этой коллекции.
* default Stream<E>parallelStream() - Возвращает возможно параллель Stream с этой коллекцией в качестве источника.
* boolean remove(Object o) - Удаляет один экземпляр указанного элемента из этой коллекции, если он присутствует .
* boolean removeAll(Collection<?> c) - Удаляет все элементы этой коллекции, которые также содержатся в указанной коллекции.
* default boolean removeIf(Predicate<? super E> filter) - Удаляет все элементы этой коллекции, которые удовлетворяют заданному предикату.
* boolean retainAll(Collection<?> c) - Сохраняет только элементы в этой коллекции, которые содержатся в указанной коллекции.
* int size() - Возвращает количество элементов в этой коллекции.
* default Spliterator<E> spliterator() - Создает Spliterator над элементами в этой коллекции.
* default Stream<E> stream() - Возвращает последовательность Streamс этой коллекцией в качестве ее источника.
* Object[] toArray() - Возвращает массив, содержащий все элементы в этой коллекции.
* <T> T[] toArray(T[] a)- Возвращает массив, содержащий все элементы в этой коллекции; тип выполнения возвращаемого массива - тип указанного массива.

Интерфейс Set.

Коллекция set – коллекция уникальных элементов, добиш не может содержать копий. Эта коллекция не может содержать сама себя, хотя другие коллекции позволяют.

Особенность данной коллекции в том, что в ней отсутствуют новые методы. Содержит уточненный контракт. Также один set-коллекция равна другой set-коллекции по содержимому.

В случае не модифицирующей реализации в set необходимо переопределить size (), iterator ().

Интерфейс SortedSet.

Данный интерфейс расширяет интерфейс Set и определяет поведение множеств, отсортированных в порядке возрастания.

Методы интерфейса SortedSet:

* Comparator <? super E> comparator () - возвращает компаратор, используемый для упорядочения элементов в этом наборе, или null, если этот набор использует естественный порядок его элементов.
* E first() - возвращает первый (самый низкий) элемент, текущий в этом наборе.
* SortedSet<E> headSet (E *конец*) - возвращает часть коллекции, элементы, которого меньше, чем *конец.*
* E last() - Возвращает последний (самый высокий) элемент на данный момент.
* SortedSet<E> subSet (E *fromElement*, E *toElement*) – возвращает часть коллекции, элементы варьируется от *fromElement* до *toElement*.
* SortedSet<E> tailSet (E *fromElement*) - возвращает часть коллекции, элементы которые больше или равны элементу.

Исключения:

* NoSuchElementException – кидается если в вызванном множестве отсутствуют элементы.
* ClassCastException – кидается если заданный объект несовместим с элементами.
* NullPointerException – кидается если используется пустой объект с null.
* IllegalArgumentException – кидает в случае неправильного аргумента.

NavigableSet

Этот интерфейс расширяет интерфейс SortedSet и определяет поведение коллекции, извлечение элементов из которой на основании наиболее точного совпадения с заданным значением или значениями.

Методы

E ceiling (E e) - возвращает наименьший элемент в этом наборе, который больше или равен данному элементу, или null если нет такого элемента.

Iterator<E> descendingIterator() - возвращает итератор по элементам в этом наборе в порядке убывания порядке убывания.

NavigableSet<E> descendingSet () - возвращает обратный порядок элементов, содержащихся в этом наборе.

E floor (E e) - возвращает наибольший элемент в этом наборе, меньший или равный данному элементу, или null, если такого элемента нет.

NavigableSet<E> headSet (E toElement, boolean inclusive) - возвращает часть набора, элементы меньше (или равны, если inclusive верно) toElement.

E higher(E e) возвращает наименьший элемент, который строго больше, чем данный e, или null.

E lower(E e) - Возвращает наибольший элемент, который строго меньше, чем данный элемент, или null.

E pollFirst() - извлекает и удаляет первый (самый низкий) элемент или возвращает, null если.

E pollLast() -извлекает и удаляет последний (самый высокий) элемент или возвращает, null.

NavigableSet<E> subSet (E *fromElement*, boolean *fromInclusive*, E *toElement*, boolean *toInclusive*) - возвращает часть элементы, которого колеблется от fromElement до toElement.

NavigableSet<E> tailSet (E fromElement, boolean inclusive) - возвращает часть элементов, которые больше (или равны, если inclusive верно) fromElement.

Интерфейс List

Интерфейс list – определяет такое поведение коллекции, которое сохраняет последовательность элементов. Элементы могут быть введены или извлечены по индексу, который начинается с нуля. Список может сохранять повторные элементы. Полное объявление:

interface List<E>

Данный интерфейс имеет ряд своих методов:

E get (int index) – возвращает элемент по индексу.

int indexOf (Object o) – возвращает индекс на элемент.

int lastIndexOf (Object o) – получить последний найденный элемент.

ListIterator<E> listIterator () – расширенная версия метода iterator () для list.

ListIterator<E> listIterator (int index) – iterator с определенного индекса.

List<E> subList (int fromIndex, int toIndex) -Возвращает представление части этого списка между указанным fromIndex, inclusive и toIndex, исключительным.

void remove () - удаляет элемент по индексу.

void add (int index, E element) – добавляет в список элемент с индексом.

E set (int index, E element) – заменить элемент на другой.

ListIterator<E> extends Iterator<E>

ListIterator<E> - расширенная версия iterator, с возможностью двигаться назад. Методы:

int nextIndex () – возвращает следующий индекс.

int previousIndex () - возвращает обратный индекс.

boolean hasPrevious () – возвращает индекс на котором он стоит.

Queue.

Расширяет интерфейс Collection и определяет поведение очереди с принципом “Первый вошел – первый обслужен”.

Методы:

E element() – возвращает элемент из головы, но не удаляет, если коллекция пуска кидает исключение.

boolean offer(E e) - вставляет элемент в очередь.

E peek() –возвращает и не удаляет голову очередь, если коллекция пуста, возвращает null.

E poll() – возвращает и удаляет голову очереди, если коллекция пуста, возвращает null.

E remove() - извлекает и удаляет голову очереди, если коллекция пуска кидает исключение.

Dequeue.

Данный интерфейс расширяет интерфейс Queue. и определяет поведение с двумя принципами: “ Первый вошел – первый обслужен ” или как стек “Последний вошел - первый обслужен”.

Методы:

void addFirst (E элемент) - вставляет элемент в начало стека, если это можно сделать немедленно, не нарушая емкости, бросает IllegalStateException если нет места.

void addLast(E элемент) - вставляет элемент в конец стека, если это возможно сделать немедленно, не нарушая емкости, бросает IllegalStateException, если нет места.

Iterator<E> descendingIterator () – возвращает обратный итератор.

boolean offer(E *элемент*) -вставляет *элемент* в очередь, представленную этим deque (другими словами, в хвосте этого deque), если это возможно сделать немедленно, не нарушая емкости.

boolean offerFirst(E *элемент*) - вставляет указанный элемент в начале этого окна, если это не будет нарушать ограничения емкости.

E pop () – возвращает и удаляет голову очереди.

void push ()- выводит элемент в начало очереди.

E removeFirst() - извлекает и удаляет первый элемент этого deque.

Стандартные списки.

Коллекция ArrayList.

Класс или коллекция ArrayList реализует интерфейс List и расширяет класс AbstractList. ArrayList -служит для динамический массивов с их наращиванием и ослаблением. Сам класс обобщенный и имеет три конструктора:

* ArrayList () – коллекция с возможностью наращивания и ослабления.
* ArrayList (Collection<? extends E> c) – списочный массив, инициализируемый элементами из коллекции.
* ArrayList (int *емкость*) – создает коллекцию с начальной емкостью, добиш размером.

Пример:

import java.util.\*;

// Sorry Holy People. Holy thanks’ you.

public class TestCollectionArrayList{

public static void main(String []args){

ArrayList<String> Holylist = new ArrayList<>();

System.out.println("Size of the Holylist: "+Holylist.size());

//добавляем элементы

Holylist.add("Jesus");

Holylist.add("Prophet Muhammad");

Holylist.add("Buddha");

Holylist.add("Confucius");

Holylist.add(2, "Yahweh");

System.out.println("Content after the certification: "+Holylist);

Holylist.remove(2);

System.out.println("\n"+"Size of list:"+Holylist+"\n"+"Content: "+Holylist);

System.out.println(Holylist.get(2));

// Характеристики:

System.out.println("\characteristics:");

if(Holylist.isEmpty()){

System.out.println("Collection is empty");

} else {

System.out.println("Collection is not empty");

}

System.out.println("Size of the Holylist: "+Holylist.size());

System.out.println("Hash-Code in Holylist:"+Holylist.hashCode());

}

}

Ручная установка, наращивание или ослабление количества осуществляется с помощью следующих методов:

* void ensureCapacity (int *емкость*) – увеличивает.
* void trimToSize ()

Получение массива из коллекции типа ArrayList имеет ряд преимуществ:

* Ускорение выполнения некоторых операций.
* Передача массива в качестве параметра методам, которые не перегружаются, чтобы принимать коллекции непосредственно.
* Интеграция нового кода из унаследованного кода.

Для преобразования используются методы:

Object [] toArray ()

<T>T [] toArray (T массив [])

Пример:

import java.util.\*;

public class ExampleArrayListMassive {

public static void main (String [] args) {

ArrayList<Integer> al = new ArrayList<Integer>();

al.add(1);

al.add(5);

al.add(6);

System.out.println("Содержимое: "+al);

Integer [] ai = new Integer[al.size()];

ai = al.toArray(ai);

int sum = 0;

for(int i : ai) sum +=i;

System.out.println("Сумма: "+sum);

}

}

LinkedList

Класс или коллекция LinkedList расширяет класс AbstractSequentalList и реализует интерфейсы List, Queue, Dequeue.Представляет структуру данных связного списка. Класс является обобщенным и имеет 2 конструктора:

* LinkedList()
* LinkedList(Collection<? extends E> c)

Так, как данный класс реализует очереди, то ему доступен ряд методов:

* addFirst () – выводит элементы из начала списка.
* addLast () – выводит элементы из конца списка.
* getFirst () – выводит первый элемент.
* removeFirst () – удаляет 1 элемент.
* getLast ()
* removeLast ()

Пример:

import java.util.\*;

public class HelloWorld{

public static void main(String []args){

LinkedList<String> ll = new LinkedList<>();

ll.add("f");

ll.add("d ");

ll.addLast("x");

ll.addFirst("a");

ll.add(2, "c");

System.out.println("Content: "+ll);

ll.removeLast();

System.out.println("new LinkedList:"+ll);

String s = ll.get(3);

ll.set(2, s+ "Изменено");

System.out.println("LinkedList:"+ll);

}

}

Класс HashSet.

Данный класс реализует AbstractSet и реализует Set. Используется для создания коллекций, для хранения хэш-таблиц.

*Хеширование* – содержимое ключа служит для определения однозначного значения *хэш-кодом*. Хэш-код далее служит индексом по сохранению данных. Преобразования в хэш-код – автоматический процесс. Преимущество хеширования постоянство выполнения методов: add(), contains(), remove(), size().

Конструкторы:

* HashSet() – способ по умолчанию.
* HashSet(Collection<? extends E> c)
* HashSet (int *емкость)* – задается *емкость* хэш-множества (по умолчанию - 16).
* HashSet( емкость, *коэффициент\_заполнения* )

Данный класс не сохраняет последовательность. Не имеет своих методов, за исключением подклассов.

Пример:

import java.util.\*;

public class HashSetExample{

public static void main(String []args){

HashSet<String> hs = new HashSet();

hs.add("Бета");

hs.add("Альфа");

hs.add("Гамма");

hs.add("Ипсилон");

hs.add("Омега");

System.out.println("HashSet:"+hs);

}

}

Класс LinkedHashSet.

Данный класс расширяет класс HashSet, не добавляет методов, но сохраняет элементы в том порядке в котором введены.

Пример:

import java.util.\*;

public class LinkedHashSetExample{

public static void main(String []args){

LinkedHashSet<String> hs = new LinkedHashSet();

hs.add("Бета");

hs.add("Альфа");

hs.add("Гамма");

hs.add("Ипсилон");

hs.add("Омега");

System.out.println("HashSet: "+hs);

}

}

Выводит:

HashSet:[Бета, Альфа, Гамма, Ипсилон, Омега]

Класс TreeSet.

Класс TreeSet расширяет AbstractSet и реализует NavigableSet. Создает коллекцию, где используется *древовидная структура.* Объекты сохраняют в отсортированном порядке по нарастающей. Время доступа и излечения довольно быстрая и поэтому ее следует использовать большой и отсортированной информации. Конструкторы:

* TreeSet()
* TreeSet(Collection<? extends E> c)
* TreeSet(Comparator<? super E> *компаратор*)
* TreeSet(SortedSet<E> *ss*)

Пример:

import java.util.\*;

public class TreeSetExample{

public static void main(String []args){

TreeSet<String> tr = new TreeSet<>();

tr.add("A");

tr.add("B");

tr.add("C");

tr.add("D");

tr.add("E");

tr.add("F");

tr.add("J");

System.out.println("Content: "+tr);

System.out.println("Фрагмент:"+tr.subSet("A", "E"));

System.out.println(tr.floor("J"));

tr.pollFirst();

System.out.println(tr);

}

}

Класс PriorityQueue.

Данный класс расширяет AbstractQueue и реализует Queue. Класс PriorityQueue создает очередь по приоритетам на основании компаратора очереди.

Конструкторы:

* PriorityQueue() -создает PriorityQueue с начальной емкостью, по умолчанию (11).
* PriorityQueue (Collection<? extends E> *c*) – создает коллекцию со всеми элементами коллекции *c.*
* PriorityQueue(int *емкость*) – создает коллекцию с указанной емкостью.
* PriorityQueue(int *емкость*, Comparator<? super E> comparator)
* PriorityQueue(PriorityQueue<? extends E> c)
* PriorityQueue(SortedSet<? extends E> c)

Если компаратор не указан, то ставится компаратор по умолчанию. Компаратор по умолчанию выбирается по типам, которые обрабатывает коллекция и размещает элементы по нарастающему размеру.

Вызвав метод comparator можно получить ссылку на компаратор используемый очередью. Объявление:

Comparator <? super E> comparator ()

Ниже представлен пример использования PriorityQueue с обратным компаратором:

import java.util.\*;

public class Main {

public static void main(String[] args) {

PriorityQueue<String> queue = new PriorityQueue<>(Comparator.reverseOrder());

queue.add("Толя");

queue.add("Аркаша");

queue.add("Кеша");

queue.add("Гриша");

System.out.println(queue);

System.out.println("poll() -> "+queue.poll());

System.out.println(queue);

System.out.println("peek() -> "+queue.peek());

System.out.println(queue);

System.out.println("poll() -> "+queue.poll());

System.out.println(queue);

String s;

while ((s=queue.poll()) != null) {

System.out.println("poll() -> " + s);

System.out.println(queue);

}

System.out.println("Содержимое: "+queue);

}

}

Класс ArrayDeque.

Данный класс расширяет AbstractCollection и реализует Deque, он не имеет своих методов и создает динамический массив без ограничения емкости.

Конструкторы:

* ArrayDeque()
* ArrayDeque (int *размер*) – двусторонняя очередь с указанным размером.
* ArrayDeque (Collection)

Пример:

import java.util.\*;

public class ExampleArrayDeque{

public static void main(String []args){

ArrayDeque<String> ard =new ArrayDeque<>();

ard.push("J");

ard.push("A");

ard.push("V");

ard.push("S");

ard.push("E");

ard.push("J");

ard.push("O");

System.out.print("Взятие из стека:");

while(ard.peek() != null)

System.out.print(ard.pop()+"");

System.out.println("");

}

}

Класс EnumSet.

Данный класс расширяет класс AbstractSet и реализует Set. Он служит для создания множества, предназначенного для работы с ключами перечислений. Объявление:

class EnumSet<E extends Enum<E>>

У него нет конструкторов, вместо этого для создания объектов используют фабричные методы. Методы:

static <E extends Enum<E>> EnumSet<E> allOf(Class<E> c) - c набор перечислений, содержащий все элементы в указанном типе элемента.

static <E extends Enum<E>> EnumSet<E> complementOf(EnumSet<E> s) –создает множества тип EnumSet и добавляет необходимые объекты (s).

static <E extends Enum<E>> EnumSet<E> copyOf(Collection<E> c) - создает набор перечислений, инициализированный из указанной коллекции.

static <E extends Enum<E>> EnumSet<E> complementOf(EnumSet<E> s) – создает множество типа EnumSet, содержащие элементы из множества s.

static <E extends Enum<E>> EnumSet<E> of(E v) – создает множества типа EnumSet, содержащие элементы v.

static <E extends Enum<E>>

EnumSet<E> range (E *начало*, E *конец*) – создает набор перечислений, содержащий все элементы от *начало* и *конец.*

Применение Iterator и for-each.

Доступ к элементам через итератор.

Итератор следует использовать, когда требуется перебрать все элементы коллекции, например – вывести каждый элемент.

Итератор – объект класса реализующий Iterator или ListIterator. В интерфейсе Iterator можно организовать цикл для перебора элементов и их удаления, ListIterator для двустороннего обхода списка и видоизменения.

Методы интерфейса Iterator:

default void forEachRemaining (Consumer<? super E> *действие* ) – выполняет *действие* над каждым необработанным элементом.

boolean hasNext() – возвращает true, если коллекция еще имеет элементы.

E next() - возвращает следующий элемент коллекции.

void remove() – удаляет текущий элемент итерации.

Методы интерфейса ListIterator:

boolean hasPrevious () - Возвращает если сзади в списке имеется элемент – true.

int nextIndex() - возвращает индекс элемента, который будет возвращен последующим вызовом next().

E previous() – возвращает предыдущий элемент в списке.

Применение интерфейса Iterator.

Прежде, чем использовать итератор для коллекций нужно сделать следующие действия:

* Вызвать итератор с помощью iterator().
* Организовать цикл, в котором вызывается метод hasNext(). Перебирать содержимое до конца.
* Получить каждый элемент вызывая метод next().

Пример:

import java.util.\*;

public class IteratorExample{

public static void main(String []args){

ArrayList<String> ar = new ArrayList<>();

ar.add("J");

ar.add("A");

ar.add("C");

ar.add("W");

ar.add("S");

ar.add("V");

System.out.print("Исходное содержимое: ");

Iterator<String> itr= ar.iterator();

while(itr.hasNext()){

String element =itr.next();

System.out.print(element+" ");

}

System.out.println();

ListIterator<String> lit =ar.listIterator();

while(lit.hasNext()){

String element= lit.next();

lit.set(element+"+");

}

System.out.print("Видоизмененная версия:");

itr = ar.iterator();

while(itr.hasNext()){

String element= itr.next();

System.out.print(element+" ");

}

System.out.print("\nОбратная версия:");

while(lit.hasPrevious()){

String element=lit.previous();

System.out.print(element+" ");

}

}

}

Цикл for-each, как альтернатива.

Если требуется просто перебрать элементы коллекции, то можно использовать цикл for-each. Пример:

import java.util.\*;

public class For\_EachExample{

public static void main(String []args){

ArrayList<Integer> arl = new ArrayList<>();

arl.add(1);

arl.add(4);

arl.add(6);

arl.add(9);

arl.add(13);

arl.add(23);

System.out.print("Содержание: ");

for(int v : arl) System.out.print(v+" ");

}

}

Итератор-разделитель и интерфейс Spliterator.

Итератор-разделитель является одной из разновидности итераторов и определяется в интерфейсе Spliterator. Служит для параллельного итерации, программирования и т.п. Хотя и можно использовать его и в сторонних случаях. Главной особенностью является то, что в одном методе могут быть сразу два и более метода.

Методы из интерфейса Spliterator:

int characteristics() - возвращает набор характеристик этого итератора-разделителя.

long estimateSize() - возвращает оценку количества элементов, которые будут встречаться при итерации, если количество нельзя получить возвращается Long.MAX\_VALUE

default void forEachRemaining(Consumer<? super T> *действие*) – выполняет *действие* над каждым элементом итерации.

default Comparator<? super T> getComparator() – возвращает компаратор используемый данной итерацией или null если используется естественное упорядочение.

default long getExactSizeIfKnown() – возвращает количество оставшихся элементов, если установлен размер.

default boolean hasCharacteristics (int *c*) – возвращает true, если у итератора-разделителя имеются характеристики, передаваемые через *c*.

boolean tryAdvance(Consumer<? super T> *действие* ) выполняет *действие* над следующим элементом.

Spliterator<T> trySplit() – разделяет итератор-разделитель, возвращает ссылку на новый итератор.

Пример:

import java.util.\*;

public class For\_EachExample{

public static void main(String []args){

ArrayList<Double> arl = new ArrayList<>();

arl.add(1.0);

arl.add(2.0);

arl.add(3.0);

arl.add(4.0);

arl.add(5.0);

System.out.print("Содержимое списочного массива arl:\n");

Spliterator<Double> spl =arl.spliterator();

while(spl.tryAdvance((n) -> System.out.println(n) ));

System.out.println();

}

}

Создание объектов пользовательских классов коллекций.

Коллекции не ограничены встроенными объектами. В них можно использовать и просто объекты. Пример:

import java.util.\*;

class People{

private String name;

private String surname;

private int age;

People(String name, String surname, int age){

this.name=name;

this.surname=surname;

this.age=age;

}

@Override

public String toString(){

return name+" "+surname+" "+age;

}

}

public class Example{

public static void main(String []args){

LinkedList<People> ll = new LinkedList<>();

ll.add(new People ("Андрей", "Пуллынен", 27));

ll.add(new People ("Анна", "Кудрявцева", 15));

ll.add(new People ("Альфира", "Кургановна-Дружинина", 49));

for(People element : ll) System.out.println(element+"");

}

}

Обращение с отображениями.

Основы.

*Отображение* или ассоциативная карта– объект, сохраняющий связи между ключами и значениями в виде пар *“ключ-значение”*. По ключу можно найти его значение. Ключи являются в бытие, однозначны и уникальны, а значения могут быть дублированным.

Интерфейсы для работы с отображениями не реализует интерфейс Collection и значит, не реализуют интерфейс Iterator, следует и не работает и for-each. Хотя есть свои способы просмотра отображения.

Интерфейсы отображений:

* Map – отображает однозначные ключи и значения.
* Map.Entry – описывает элемент отображения (пару “ключ-значение”). Является внутренним классом интерфейса Map.
* NavigableMap - расширяет интерфейс SortedMap для извлечения элементов по принципу Navigable.
* SortedMap – расширяет NavigableMap, для того чтоб ключи располагались по нарастающей.

Ключ – это объект, используемый для последующего извлечения данных. Задавая эту пара, можно размещать элементы в виде объекта Map. Map – является коллекцией из-за его структуры, но при этом Map не коллекция из-за его целей и в какой-то степени это паттерн, но и не паттерн.

Интерфейс Map.

Данный интерфейс отображает однозначные ключи на значения. Map это обобщенный интерфейс и объявляется:

interface Map<K, V>

где K – тип ключа, где V – тип хранимых в ней отображение значений. Данный интерфейс содержит много методов:

* int size () – аналогичен методу из Collection.
* boolean isEmpty() - аналогичен методу из Collection.
* boolean containsKey(Object *key*) – проверяет на наличие пару на ключ.
* boolean containsValue(Object *value)* –проверяет пару на наличие значения
* V put (Object *key*, Object *value*) –добавляет в коллекцию новую пару.
* V remove(Object *key*) – удаляет пару.
* void putAll(Map<? extends K, ? extends V> *отображение*) - добавляет сразу еще одну коллекцию.
* void clear() –чистить коллекцию.
* Set<K> keyset () – возвращает множество ключей.
* Collection<V> values () - возвращает коллекцию значений.
* Set<Map.Entry<K, V>> entrySet () – возвращает множество, содержащие все записи из вызывающего изображения.
* V getOrDefault<Object K, Object *заданное\_значение*) - возвращает K, если такого нет то пару из *заданное\_значение*.
* V get (Object k) – возвращает нужный элемент.
* default V putIfAbsent(K key, V value) – связывает несвязанный ключ со значением.
* default V replace(K key, V value) – заменяет запись для ключа в том случае, если сейчас в нем отображается значение.

Интерфейс SortedMap.

Размещает записи в отображении по нарастанию ключей. Расширяет интерфейс Map. Является обобщенным.

Исключения:

* NoSuchElementException – кидается если вызывающий элемент пуст.
* ClassCastException – если типы не совместимы.
* NullPointerException – кидается при попытке работы с пустым объектом.
* IllegalArgumentException – кидается если указан не правильный аргумент.

Методы

Comparator<? super K> comparator() - возвращает компаратор, используемый для заказа ключей на этой карте.

K firstKey() - возвращает первый ключ из данного отображения.

SortedMap<K, V> headMap (K *конец*) – возвращает часть представления ключи которой меньше, чем *конец*.

SortedMap<K, V> tailMap (K *начало*) *–* возвращает часть отсортированного отображения с ключа *начало*.

K lastKey() -возвращает последний ключ, на отображении.

Интерфейс NavigableMap.

Определяет поведение, в котором все элементы отсортированы и могут вызываться по наиболее точному совпадению.

Интерфейс Map.Entry.

Методы:

* boolean equals (Object *объект*) – сравнивает пары.
* V getValue () – возвращает значение из отображения.
* K getKey () – возвращает ключ из отображения.
* int hashCode () –возвращает хэш-код данного отображения.

Классы отображений.

Класс HashMap.

Данный класс определяет поведение для хеширования отображений. Элементы не упорядочены, хаотичны. Класс HashMap не содержит своих методов.

Пример:

import java.util.\*;

public class HashMapExamlple{

public static void main(String [] args){

HashMap<String, Integer> hm= new HashMap<>();

hm.put(“Я”, new Integer(12000));

hm.put(“Аноним”, new Integer(5000));

hm.put(“Банк\_Обман”, new Integer(23500000));

Set<Map.Entry<String, Integer>> set = hm.entrySet();

for(Map.Entry<String, Integer> me : set){

System.out.println(“Ключ вкладчика “ +me.getKey()+”. Счет вкладчика:”+me.getValue());

}

int balance = hm.get(“Аноним”);

hm.put(“Аноним”, balance + 3000);

System.out.println(“\nНовый счет анонима “+hm.get(“Аноним”));

}

}

TreeMap

Данный класс определяет поведение, в котором элементы упорядочены в древовидной структуре. Класс TreeMap предоставляет быстрое извлечение и надежное хранение. Упорядочены они от меньшего к большему элемента. Конструкторы:

* TreeMap () – создает пустое древовидное отображения.
* TreeMap (Comparator<? super K> *компаратора*) - создает пустое древовидное отображения, но с указанием компаратора.
* TreeMap (Map<K, ? extends V> *m*) – создает древовидное отображение с инициализирующие элементы из коллекции m, с естественным порядком.
* TreeMap (SortedMap<K, ? extends V> *sm*) – схож с верхним конструктором, но сохраняется порядок sm.

Пример:

import java.util.\*;

class TreeMapExamlple{

public static void main(String [] args){

TreeMap<String, Integer> tm = new TreeMap<>();

tm.put("Я", new Integer(45000));

tm.put("Аноним", new Integer(30000));

tm.put("George Washington", new Integer(1000000));

tm.put("Stalin", new Integer(550000));

tm.put("Winston Churchill", new Integer(5000000));

System.out.println(tm);

Set<Map.Entry<String, Integer>> set = tm.entrySet();

for(Map.Entry<String, Integer> me : set){

System.out.println("\nКлюч вкладчика "+me.getKey()+". Счет вкладчи"+me.getValue());

}

}

}

Классы LinkedHashMap.

Класс LinkedHashMap расширяет HashMap. У него связанная структура, которая сохраняется при добавлении. Это помогает организовать итерацию с вводом элементов. Также можно назначить порядок, в котором к ним последний раз осуществлялся доступ.

Конструкторы:

* LinkedHashMap () - создается вариант по умолчанию.
* LinkedHashMap (Map<? extends *K*, ? extends V> *m*) – создается отображение с инициализирующими элементы из *m*.
* LinkedHashMap (int *емкость*) – создает отображение с начальной *емкостью*.
* LinkedHashMap (int *емкость*, flout *коэффициент\_заполнения*) – создается отображение с начальной емкостью и *коэффициентом\_заполнения*.
* LinkedHashMap (int *емкость*, flout *коэффициент\_заполнения*, boolean *порядок*) – в *порядок* определяет тип расположения элемента. Если true используется порядок доступа или false последнего входа.

Класс IdentityHashMap.

Данный класс схож LinkedHashMap, но при сравнении он сравнивает ссылок на равенство. Класс IdentityHashMap расширяет AbstractMap и реализует Map. Объявление:

class IdentityHashMap (*K*, *V*) – где *K* тип ключей, а *V* тип значений.

В javadoc написано, что класс не предназначен для общего применения.

Класс EnumMap.

Данный класс расширяет AbstractMap и реализует Map. Он предназначен для работы ключами с значениями типа enum. Объявление:

class EnumMap<*K* extends Enum<*K*>, V>

Конструкторы:

* EnumMap (Class<K> *тип\_ключа*) – создает пустое отображение для хранения значений, с ключами типа *тип\_ключа*.
* EnumMap (Map<K, ? extends V> m) – создает отображение с элементами из Map m.
* EnumMap (EnumMap<K, ? extends V> em) – создается отображения типа EnumMap инициализирующее отображение из em.

Компараторы.

*Естественный порядок* – это порядок где за A идет B, а после C и т.д. Классы типа TreeSet и TreeMap поддерживают естественный порядок. Но есть возможность задавать саму порядок коллекции, указывая в параметры множеству, списка и отображения Comparator.

Comparator является обобщенным и объявляется следующим образом, где T тип сравниваемых объектов:

Comparator <T>

Данный интерфейс является функциональным и содержит функцию compare() с полным объявлением:

int compare (T *объект\_1*, T *объект\_2*)

Обычно он возвращает ноль, если *объект\_1* равен *объект\_2*; он возвращает положительное число, если *объект\_1* больше *объект\_2*; в остальных случаях отрицательное число.

В нем также определяется метод equals(), который на равенство компаратору передаваемый в параметр *объект*. Полное объявление:

boolean equals (Object *объект*)

Использую метод reserved() можно получить компаратор, изменяющий на обратный порядок сравнивания, с которым компаратором связывался, т.е. место A, B, C идет C, B, A . Полное объявление:

default Comparator <T> reserved()

Методом naturalOrder() можно получить компаратор с натуральным порядком сравнения. Полное объявление:

static <T extends Comparable<? super T>> Comparator<T> naturalOrder()

Методом reserveOrder() можно получить компаратор с обратный ему компаратор, т.е. если у компаратора естественный порядок он вернет компаратор с обратным порядком. Полное объявление:

static <T extends Comparable<? super T>> Comparator<T> reserveOrder()

В интерфейсе Comparator есть еще один метод, который в случае первого совпадения равен, то будет применятся второй *компаратор*. Также в двух формах есть *ключ передачи*, который получает следующее сравнение. Где ключ передачи может быть String::trim и тому подобные. Это метод thenComparing() имеет три формы:

default Comparator <T> thenComparing(Comparator<? super T> *второй\_компаратор*)

default <U extends Comparable<? super U> Comparator <T> thenComparing(

Function <? super T, ? extends U> *ключ\_передачи*)

default <U extends Comparable<? super U> Comparator <T> thenComparing(

Function <? super T, ? extends U> *ключ\_передачи*,

Comparator<? super T> *второй\_компаратор*)

Также существует три формы для примитивов: int, long и double.

Пример компаратора.

В ниже представленной программе создается обычный компаратор:

package myPackage;

import java.util.Comparator;

import java.util.TreeSet;

public class Main {

public static void main(String[] args) {

TreeSet<String> set=new TreeSet<>(new MyComp());

set.add("A");

set.add("B");

set.add("C");

set.add("D");

set.add("E");

System.out.println("Множество:");

for(String s : set)

System.out.println(s);

}

}

// Создаем компаратор путем реализации интерфейса Comparator

class MyComp implements Comparator<String> {

@Override

public int compare(String o1, String o2) {

return o1.compareTo(o2);

}

}

Также можно было сделать с лямбдой:

TreeSet<String> set=new TreeSet<> ((o1, o2) -> o1.compareTo (o2));

Ниже представлена программа, в которой из входных строк, которые обозначают имя и фамилию сравниваются фамилии. Если сравнение фамилий равны, используется второй компаратор, который сравнивает имена. Код:

package myPackage;

import java.util.Comparator;

import java.util.Map;

import java.util.TreeMap;

public class Main {

public static void main(String[] args) {

Comparator<String> mc=new CompLastName().

thenComparing(String::toString, new CompName());

TreeMap<String, Double> map=new TreeMap<>(mc);

map.put("Джон Руан", new Double(124.50));

map.put("Александр Капитолийский", new Double(12.67));

map.put("Екатерина Катовна", new Double(67.20));

map.put("Мухаммед Аббас", new Double(46.23));

for(Map.Entry<String, Double> e : map.entrySet())

System.out.println(e.getKey()+" счет: "+e.getValue());

map.put("Джон Руан", map.get("Джон Руан") + 12);

System.out.println("\n"+"Новый счет Руана: "+map.get("Джон Руан"));

}

}

class CompLastName implements Comparator<String> {

@Override

public int compare(String o1, String o2) {

int i, j;

i=o1.lastIndexOf(' ');

j=o2.lastIndexOf(' ');

return o1.substring(i).compareToIgnoreCase(o2.substring(j));

}

}

class CompName implements Comparator<String> {

@Override

public int compare(String o1, String o2) {

return o1.compareTo(o2);

}

}

Алгоритмы коллекций.

В Collection Framework определяется ряд алгоритмов, которых можно применять к коллекциям и отображениям. Все они определяются в классе Collections.

Определяется ряд *проверяемых* методов, которые называют «динамический типизируемый представлением» коллекций. Они возвращают коллекцию, которая будет проверять входные в нее объекты и любой ввод несовместимого объекта генерирует ClassCastException. К ним относят checkedSet(), checkedList(), checkedMap() и checkedCollection().

Некоторые методы возвращает копию коллекцию, но при этом копия становится потокобезопасной. К ним относят synchronizedList(), synchronizedSet(), synchronizedMap(), synchronizedQueue() и т.п. Нужно учитывать, что итераторы могут действовать только в блоках с synchronized.

Ряд методов возвращают копию коллекции, но при этом в копию нельзя добавить элементы, т.е. final. К таким методам можно отнести unmodifiedList(), unmodifiedSet() и т.п.

Методом shuffle() можно перетасовать коллекцию; reserve() можно перевернуть коллекцию; методом sort() отсортировать коллекцию.

Методы min() и max() можно вернуть максимальный и минимальный объект коллекции. Ниже представлен пример использования некоторых методов:

package myPackage;

import java.util.ArrayList;

import java.util.Collections;

public class Main {

public static void main(String[] args) {

ArrayList<Integer> list=new ArrayList<>();

list.add(12);

list.add(34);

list.add(-2);

list.add(32);

System.out.print("Содержимое list: ");

for (Integer i : list)

System.out.print(i+" ");

ArrayList<Integer> list1=new ArrayList<>();

list1.add(39);

list1.add(-57);

list1.add(-12);

list1.add(33);

list1.add(159);

System.out.print("\n"+"Содержимое list1: ");

for(Integer i : list1)

System.out.print(i+" ");

list.addAll(list1);

System.out.print("\n"+"Содержимое двух листов: ");

for (Integer i : list)

System.out.print(i+" ");

Collections.shuffle(list);

System.out.print("\n\n"+"Перетасованная коллекция: ");

for(Integer i : list)

System.out.print(i+" ");

Collections.reverse(list);

System.out.print("\n"+"Перевернутый list: ");

for(Integer i : list)

System.out.print(i+" ");

Collections.sort(list,

(val, val1) -> val1.compareTo(val));

System.out.print("\n"+"Отсортированный list: ");

for(Integer i : list)

System.out.print(i+" ");

int maxVal = Collections.max(list);

int minVal = Collections.min(list);

System.out.println("\n\n"+"Минимальный элемент list: "+maxVal);

System.out.println("Максимальный элемент list: "+minVal);

}

}

Класс Arrays

Класс Arrays определяет ряд методов для работы с массивами. Эти методы помогают восполнить пробелы между массивами и коллекциями

Метод asList() создает список, исходя из переданного ему *массива*. Полное объявление метод asList():

static <T> List asList (T… *массив*)

Методом binarySearch() можно осуществить двоичный поиск *значения* в переданный массив. У него определенно ряд форм: для примитивов; для примитивов с указанием начала и конца поиска; параметризованных типов; параметризованных типов и компараторов; для Object. Если *массив* и *значение* не совпадают, генерируется исключение ClassCastException. Некоторые формы метода:

static int binarySearch (int [] *массив*, int *значение*)

static int binarySearch (int [] *массив*, int *начало*, int *конец*, int *значение*)

static <T> int binarySearch (T *массив*, T *значение*, Comparator <? super T> *компаратор*)

Метод copyOf() копирует передаваемый ему *массив*, а длина источника передается в качестве *длины*. Еще в некоторых формах указывают *результирующий тип* это тип возвращаемого типа. Если длина указана как отрицательная генерируется исключения NegativeArraySizeException, а если в массиве есть null генерируется NullPointerException. Если *результирующий тип* не совместим с источником, генерируется ArrayStoreException. Если длина больше длины копируемого массива копия наполняется нулями, а если наоборот укорачивается.

Метод copyOfRange() схож с copyOf(), но в отличии у него указывают *начало* и *конец* копирования. Если *начало* больше *конца* генерируется IllegalArgumentException.

Метод equals() сравнивает *массив1* и *массив2*. Метод deepEquals() сравнивает двухмерные массивы *массив\_1* и *массив\_2*.

Метод fill() присваивает всем элементам *массива* определенное *значение.* Также можно указать *начало* и *конец* части массива элементы, которые будет заменены на *значение*.

Метод sort() сортирует переданный в параметры *массив*. Также в версии, где используется параметризованный тип, указывают *компаратор*.

Также есть вариант метода sort(), сортирует массив параллельно – parallelSort(). Исходный *массив* указывается в параметры.

Класс Arrays поддерживает интерфейс Spliterator. Это можно сделать помощью метода spliterator(). Итератор-разделитель можно применить к массивам с int, long, double и параметризованный тип T.

Класс Arrays поддерживает интерфейс Stream. Это можно сделать помощью метода stream(). Его можно применить к массивам с int, long, double и параметризованный тип.

Ниже представлен пример использования некоторых методов:

package myPackage;

import java.util.Arrays;

import static java.util.Arrays.\*;

public class Main{

public static void main(String[] args) {

int [] array = new int[10];

for (int i = 0; i < 10; i++)

array[i] = -3 \* i;

System.out.print("Исходный массив: ");

display(array);

fill(array, 3, 6, -1);

System.out.print("Массив после fill():");

display(array);

sort(array);

System.out.print("Отсортированный массив:");

display(array);

int [] array1=new int[10];

for (int i = 0; i < 10; i++)

array1[i]= 3 \* i;

System.out.print("Исходный массив array1: ");

display(array1);

int index = binarySearch(array1, 9);

System.out.println("Индекс числа 9: "+index);

boolean res= Arrays.equals(array, array1);

System.out.println("array и array1 одинаковы: "+res);

System.out.print("Результат array с Stream API: ");

Arrays.stream(array).filter(val -> val > -11).

boxed().

distinct().

map(val -> val+1).

forEach(val -> System.out.print(val+" "));

}

static void display(int [] arr){

for(int i : arr)

System.out.print(i+" ");

System.out.println();

}

}

Унаследованные классы и интерфейсы.

До появления каркаса Collection Framework был определен ряд классов, которые служили как коллекциями. С выходом полноценных коллекций все они были после внедрены в каркас коллекций.

Интерфейс Enumeration

Данный интерфейс предшествовал интерфейсу Iterator. В современном коде не рекомендуется употреблять интерфейс Enumeration, но его использует несколько классов в API. Он объявляется следующим образом:

interface Enumeration<E>

Где E обозначает тип перечислимых объектов. В общем использование интерфейса Enumeration схоже как и использование Iterator.

Сначала нужно узнать есть ли что еще перебирать методом hasMoreElements() в цикле while, а далее методом nextElement() получить следующий элемент.

Класс Vector

Vector служит как динамический массив. Он близок к ArrayList, но он синхронизирован и содержит много устаревших методов. Вектор полностью поддерживает коллекций. Класс Vector объявляется следующим образом, где тип E обозначает тип сохраняемых объектов:

class Vector <E>

Ниже перечислены все конструкторы. Где *размер* обозначается начальный размер вектора. Также можно указать *инкремент*, который обозначает на сколько увеличится размер если количество станет больше длины вектора. Также есть версия с начальной коллекцией. Конструкторы:

Vector ()

Vector (int *размер*)

Vector (int *размер*, int *инкремент*)

Vector (Collection <? extends E> *коллекция*)

Ниже перечислены основные методы:

void addElement(E *объект*) – Вводит в вектор *объект*.

void removeElement (E *объект*) – Выводит объект из вектора.

int capacity() – Возвращает объем вектора.

int size() – Возвращает размер вектора.

E firstElement() – Возвращает первый элемент вектора.

E lastElement() – Возвращает последний элемент вектор.

Enumeration<E> elements() – Возвращает Enumeration с элементами из вектора.

Ниже представлен пример создания вектора и использования выше перечисленных методов:

package myPackage;

import java.util.Enumeration;

import java.util.Vector;

public class Main{

public static void main(String[] args) {

Vector<Integer> v = new Vector<>(4, 8);

v.addElement(123);

v.addElement(345);

v.addElement(567);

v.addElement(789);

v.addElement(12);

v.addElement(234);

v.addElement(456);

System.out.println("Емкость: "+v.capacity());

System.out.println("Размер: "+v.size());

System.out.println("Первый элемент: "+

v.firstElement());

System.out.println("Последний элемент: "+

v.lastElement());

System.out.print("Содержимое вектора: ");

Enumeration<Integer> vEnum = v.elements();

while (vEnum.hasMoreElements())

System.out.print(vEnum.nextElement()+" ");

}

}

Класс Stack <E>

Класс Stack создает очередь с принципом «Первый пришел – последний обслужен» т.е. стек. Он является производным от Vector. Он определяет конструктор по умолчанию и определяется следующим образом, где E тип элементов в стеке:

class Stack <E>

Он содержит методы класса Vector и дополняет своими методами. Методы:

|  |  |
| --- | --- |
| boolean isEmpty() | Проверяет пуст ли стек |
| E peek() | Возвращает верхний элемент не удаляя его |
| E pop() | Возвращает верхний элемент удаляя его |
| E push(E *элемент*) | Добавляет *элемент* в стек, возвращая его. |
| int search(Object *объект*) | Ищет *объект* в стеке и возвращает его индекс относительно верха |

Ниже представлен пример использование стека:

package myPackage;

import java.util.Stack;

import java.util.Vector;

public class Main{

static void popshow(Stack<String> stack){

String s = stack.pop();

System.out.println("pop() -> "+s);

System.out.println("Стек "+stack);

}

static void peekshow(Stack<String> stack){

String s = stack.peek();

System.out.println("peek() -> "+s);

System.out.println("Стек "+stack);

}

public static void main(String[] args) {

Stack<String> st = new Stack<>();

st.push("Лягушка Пеппе");

st.push("Буль умным как Петя");

st.push("Минут 5 10 15 ...");

st.push("Пацан, это ряпчик");

System.out.println("Начальный стек: "+st);

int index = st.search("Лягушка Пеппе");

System.out.println("Индекс искомого элемента: "+index);

peekshow(st);

popshow(st);

popshow(st);

peekshow(st);

popshow(st);

popshow(st);

if(st.isEmpty())

System.out.println("Стек пуст");

else

popshow(st);

}

}

Класс Dictionary.

Абстрактный класс Dictionary предшествовал Map и предоставляет собой хранилище пар «ключ – значение» или словаря. Он рекомендуется к использованию, но все его функции заменяет отображения.

Класс Dictionary является обобщенным, где U обозначает ключ, а V значение. Полное объявление данного класса:

class Dictionary<U, V>

Чтоб ввести пару нужно использовать метод put() передав ему *ключ* и *значение*. Методом get() получить значение по переданному *ключу*. Для того чтоб удалить значение нужно воспользоваться методом remove() передав ему *ключ*.

Методом size() можно узнать длину словаря. Узнать пуст ли словарь можно методом isEmpty(). Методами elements() и keys() можно получить Enumeration для значение и ключей. Также есть возможность получить множество ключей словаря методом keyset().

Класс Hashtable.

Класс Hashtable реализует хэш-таблицу и предшествует HashMap. Он является производным от Dictionary. Он отличается от HashMap тем, что он синхронизирован. Он объявляется приведенным ниже образом, где U тип ключа, а V тип значения:

class Hashtable <U, V>

В хэш-таблице может хранится только объекты, реализующие equals() и hashCode(). Ниже приведены конструкторы данного класса:

Hashtable ()

Hashtable (int *размер*)

Hashtable (int *размер*, float *коэффициент\_заполнения*)

Hashtable (Map <? extends U, ? extends V> *отображение*)

Где *размер* указывают начальный размер, где *коэффициент заполнения*, указывают в каком процентом отношении увеличится когда таблица превысит размер.

Ниже представлен пример использования хэш-таблиц:

package myPackage;

import java.util.Hashtable;

import java.util.Iterator;

public class Main{

public static void main(String[] args) {

Hashtable<String, Double> table=

new Hashtable<>(5, 0.90f);

table.put("Джон Руан", 3700.60);

table.put("Алеша Тугодумович", 2350.3);

table.put("Алексей Безанализный", 1003.67);

table.put("Чувак с горы", 7000.0);

table.put("Кабан Мандаринович", 3760.12);

Iterator<String> iterator = table.keySet().iterator();

iterator.forEachRemaining(

str -> System.out.println(str+": "+table.get(str)));

table.put("Чувак с горы", table.get("Чувак с горы") - 3000);

System.out.println("\n"+"Счет чувака с горы: "+table.get("Чувак с горы"));

}

}

Класс Properties.

Данный класс является производным от HashTable и содержит пару, которые является строчками. У него определяется конструктора, первый это по умолчанию, а второй позволяет указать *свойство по умолчанию*:

Properties ()

Properties (Properties *свойство\_по\_умолчанию*)

Ниже перечислены некоторые методы:

String getProperties (String *ключ*, String *свойство\_по\_умолчанию*) –

Ищет значение по *ключу*. Если оно не будет не найдено вернется *свойство по умолчанию*.

String getProperties (String *ключ*) –

Ищет значение по *ключу*, если оно не будет найдено вернется null.

Object setProperties (String *ключ*, String *значение*) –

Связывает *ключ* с *значением*. Возвращается предыдущее значение ключа или null.

void store (Writer *поток*, String *описание*) –

Вводит элементы после строки *описания* данные в *поток* ввода.

void store (OutputStream *поток*, String *описание*) –

Вводит элементы после строки *описания* данные в *поток* ввода.

void load (Reader *поток*) – Выводит все элементы в *поток*.

void load(InputStream *поток*) – Выводит все элементы в *поток*.

Среди методов стоит отметить load() и store(). Благодаря им данный класс по-прежнему часто используется для создания маленьких баз данных. Ниже представлен пример создания базы данных. Здесь содержаться названия стран и их столицы, при этом страна и столица вводится пользователем.

Сначала создается capitals типа Properties, который будет содержать пары. После создается буфер reader, который будет считывать данные с System.in. Поток stream типа FileInputStream читает файл geographic\_dictionary.dat, который будет содержать содержимое Properties. Если файл содержит информации, то она выводится метод load() в Properties.

Далее идет процесс записи в capitals и далее метод store () данные выводятся в файл. После (если все выше описанное сработало) идет процесс поиска столицы. Это осуществляется метод get ().

import java.io.\*;

import java.util.\*;

public class Main {

public static void main(String[] args) throws IOException{

Properties capitals = new Properties();

BufferedReader reader=

new BufferedReader(

new InputStreamReader(System.in));

String country, capital;

boolean changed = false;

try(FileInputStream stream = new FileInputStream("geographic\_dictionary.dat")){

if(stream != null)

capitals.load(stream);

}catch (IOException exc){

System.out.println(exc);

}

System.out.println("Для остановки введите Exit");

do{

System.out.println("Введите страну: ");

country = reader.readLine();

if(country.equals("Exit")) continue;

System.out.println("Введите столицу");

capital = reader.readLine();

capitals.setProperty(country, capital);

changed = true;

}while (!country.equals("Exit"));

if(changed){

FileOutputStream f =

new FileOutputStream("geographic\_dictionary.dat");

capitals.store(f, "Страна - Столица");

f.close();

}

do{

System.out.println("Введите название страны: ");

country = reader.readLine();

if(country.equals("Exit")) continue;

System.out.println(capitals.get(country));

}while (!country.equals("Exit"));

}

}

Аннотации.

*Аннотация* или *метаинформация*  содержат семантику программы. Но следует учитывать, то что аннотации никак не меняют порядок выполнения программы.

Метаинформация может использоваться различными инструментами во время разработки или развертывания информации. Также ее может использовать компилятор во время генерации кода.

Все связанное с аннотациями содержатся в пакете java.lang.annotation. В нем определено две основных аннотации. Первая это Retention, которая определяет время жизни аннотации, а вторая Target к чему будет привязана аннотация.

Аннотации помечаются в начале значком собачки или эмаил @.

Правила привязок аннотаций.

Аннотация Retention определяет время жизни аннотации. Все они определенны в классе RetentionPolicy. Общая форма объявления Retention:

@Retention (*правило\_утверждения*)

В нижней таблице представлены все правила:

|  |  |
| --- | --- |
| SOURCE | Содержится в исходном коде |
| CLASS | Сохраняется в файле с расширением .class во время компиляции |
| RUNTIME | Сохраняется в файле с расширением .class во время компиляции и выполнения |

Аннотация Target определяет привязку аннотации. Все значения определяются в классе ElementType. Полное объявление аннотации Target:

@Target (ElementType [] *тип*)

Пример аннотации

В нижней программе создается аннотация, которая применима к методам и живет в исходном коде:

package myPackage;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

public class Main {

@MyAnnotation(str="method", val = 100)

static void method(){

System.out.println("method()");

}

public static void main(String[] args) {

method();

}

}

@Target({ElementType.METHOD})

@Retention(RetentionPolicy.SOURCE)

@interface MyAnnotation{

String str();

int val() default -1;

}

Значения по умолчанию.

В выше использовались *значения по умолчанию*. Они позволяют, когда какой-то параметр аннотация второстепенен, использовать для него значение по умолчанию. Для этого используется ключевое слово default. Полное объявление членов аннотации значением по умолчанию:

*тип имя\_члена\_аннотации*() default *значение*

Получение метаинформации

с помощью рефлексии.

Интерфейс AnnotatedElement.

Интерфейс AnnotatedElement представляет рефлексию для аннотаций. В нем определено 4 метода. Первый это getAnnotation(), который возвращает одну аннотацию, которая должна совпадать с тип *аннотаций*.

Методом getDeclaredAnnotations() возвращает массив не унаследованных аннотаций. Полное объявление метода:

Annotations [] getDeclaredAnnotations()

Метод isAnnotationPresent() проверяет, связана ли аннотация объекта с *типом аннотации*. Полное объявление метода:

boolean isAnnotationPresent (Class <? extends Annotation> *тип\_аннотация*)

Пример получения метаинформации.

Чтобы получить информацию с аннотаций нужно воспользоваться рефлексией. Для этого нужно сделать ниже перечисленные действия:

* Получить объект класса
* Получить объект класса Class<?>
* Получить объект метода, поля и т.д.

Ниже представлен пример получения метаинформации:

package myPackage;

import java.lang.annotation.\*;

import java.lang.reflect.Method;

public class Main {

@MyAnnotation(str="Пример аннотации", val = 34)

public static void infoMeto(){

Main ob=new Main();

try{

Class<?> c = ob.getClass();

Method cMethod = c.getMethod("infoMeto");

MyAnnotation annotation=cMethod.getAnnotation(MyAnnotation.class);

System.out.println("Аннотация infoMeto(): "+annotation.str()+" "+annotation.val());

} catch (NoSuchMethodException e) {

System.out.println("Метод не найден");

}

}

public static void main(String[] args) {

infoMeto();

}

}

@Retention(RetentionPolicy.RUNTIME)

@interface MyAnnotation{

String str();

int val() default -1;

}

Получение всех аннотаций.

Можно получить сразу все аннотации, удерживаемые RUNTIME проаннотированные определенному объекту методом getAnnotations(). Полное объявление метода:

Annotation [] getAnnotations()

Ниже приведен пример программы с рефлексией, демонстрирующая получения всех аннотаций класса и метода:

package myPackage;

import java.lang.annotation.\*;

import java.lang.reflect.Method;

@MyAnnotation(str = "Пример вывода", val = 100)

@What(str = "всех аннотаций", value = 100)

public class Main {

@MyAnnotation(str="Пример вывода", val = 34)

@What(str = "всех аннотаций", value = 34)

public static void infoMeto(){

Main ob=new Main();

Annotation [] annotations=ob.getClass().getAnnotations();

System.out.println("Аннотация класса: ");

for(Annotation a : annotations){

System.out.println(a);

}

try {

Method m=ob.getClass().getMethod("infoMeto");

Annotation [] annotations1=m.getAnnotations();

for(Annotation a : annotations1){

System.out.println(a);

}

} catch (NoSuchMethodException e) {

System.out.println(e);

}

}

public static void main(String[] args) {

infoMeto();

}

}

@Retention(RetentionPolicy.RUNTIME)

@interface MyAnnotation{

String str();

int val() default -1;

}

@Retention(RetentionPolicy.RUNTIME)

@interface What{

String str() default "Тест";

int value();

}

Одночленные аннотации.

Как не трудно догадаться, что *одночленная аннотация* состоит из одного члена. Она допускает короткую форму указания члена, но единственный член должен быть value(). Ниже представлен пример создания и использования одночленной аннотаций:

package myPackage;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.reflect.Method;

public class Main {

@MySingle(100)

public static void myMeth() {

Main obj=new Main();

try {

Method m = obj.getClass().getMethod("myMeth");

MySingle annotation=m.getAnnotation(MySingle.class);

System.out.println(annotation.value());

} catch (NoSuchMethodException e) {

System.out.println("Метод не найден");

}

}

public static void main(String[] args) {

myMeth();

}

}

@Retention(RetentionPolicy.RUNTIME)

@interface MySingle{

int value();

}

Аннотация-маркер.

*Аннотация-маркер* – это специальный маркер, который не содержит членов и применяется для метки к чему-либо.

Самое лучшее применение для марок это выяснить присутствует ли метка методом isAnnotationPresent().

Ниже представлен пример использования аннотации-маркера:

package myPackage;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.reflect.Method;

public class Main {

@Marker

public static void test(){

Main ob=new Main();

try {

Method m=ob.getClass().getMethod("test");

if(m.isAnnotationPresent(Marker.class))

System.out.println("Аннотация-маркер @Marker присутствует");

} catch (NoSuchMethodException e) {

System.out.println("Метод не найден");

}

}

public static void main(String[] args) {

test();

}

}

@Retention(RetentionPolicy.RUNTIME)

@interface Marker {}

Повторяющие аннотации.

*Повторяющая аннотация* позволяет повторяться у объекта аннотацию. Чтобы создать такую аннотацию нужно повесить на создаваемую аннотацию повесить аннотацию @Repeatable и в ее поле value() передать *контейнер*, который хранит повторяющие аннотации. Такой контейнер указывается в виде аннотации, для которой поле value является массивом типа повторяющейся аннотации.

Следовательно, чтобы сделать аннотацию повторяющейся, прежде нужно создать контейнерную аннотацию, а затем указать ее тип в качестве аргумента аннотации @Repeatable.

Ниже представлен пример создания и использования повторяющей аннотации:

package myPackage;

import java.lang.annotation.\*;

import java.lang.reflect.\*;

public class Main {

@MyAnnotation(name= "Первая аннотация", value = -1)

@MyAnnotation(value = 12)

public static void test(String name, int val){

Main ob=new Main();

try {

Method m=ob.getClass().getMethod("test",

String.class, int.class);

Annotation anno=m.getAnnotation(MyRepeated.class);

System.out.println(anno);

} catch (NoSuchMethodException e) {

System.out.println("Метод не найден");

}

}

public static void main(String[] args) {

test("тест", 10);

}

}

@Retention(RetentionPolicy.RUNTIME)

@interface MyRepeated{

MyAnnotation [] value();

}

@Retention(RetentionPolicy.RUNTIME)

@Repeatable(MyRepeated.class)

@interface MyAnnotation {

String name() default "test";

int value();

}

Ограничения на аннотации.

Одна аннотация не может расширять другую. Все методы аннотации не могут принимать параметров. Аннотацию невозможно обобщить. Методы не могут использовать оператор throws. Методы должны возвращать ниже перечисленных типов:

* Примитивный тип
* Объект типа String и Class
* enum
* Аннотация
* Массив выше перечислимого

Встроенные аннотации.

@Inherited

Данная аннотация предназначена для других аннотаций. Она влияет только на классы. Она обозначает, что привязанная к ней аннотация будет видится в подклассе.

@FunctionalInterface

Он обозначает, что интерфейс является функциональным. Если она не привязана к функциональному интерфейсу вылетает ошибка.

@Override

Данная метка предназначена только для методов. Обозначает, что метод должен переопределять метод суперкласса. Если она не правильно привязана, вылетает ошибка.

@SuppressWornings

Эта аннотация обозначает, что следует подавить одно или несколько предупреждений, которые могут быть выданы компилятором. Подавляемые предупреждения указываются по имени в строковой форме.

@Deprecated

Эта аннотация-маркер обозначает, что объявление устарело и должно быть заменено более новой формой.

@Document

Используется только аннотациями. Обозначает, что аннотация должна быть документирована.

@SafeVarargs

Данная аннотация применяется к методу или конструктору и указывает, что код не осуществляет потенциально опасных операций со своим varargs.

Дескрипторы.

Отличают третий вид комментариев *документирующим комментарием*, которые предназначены для составления документации в javadoc т.е создает HTLM файл содержащий информацию о проекте или ее части.

Они начинаются /\*\* и заканчиваются \*/. Внутри документирующего комментария находятся дескрипторы. Дескрипторы начинаются с @, но не относятся к аннотация. К примеру дескриптор @author должен содержать информацию о авторе.

Основные дескрипторы.

Для javadoc определяется ряд дескрипторов. Многие активно используют HTLM данные. Кроме можно легко вносить HTML данные в комментирующий документ.

Дескрипторы начинающие с @ называются автономными и находятся сначала документирующего комментария. Дескрипторы начинающиеся с { называются встроенными и находятся в внутри.

Таблица дескрипторов:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Объявление | Описание | Класс | Метод | Поле | Конструктор | Интерфейс | Пакет |
| @author | Содержит информацию о авторе | + | - | - | - | + | + |
| {@code} | Отображает информацию в исходном виде | + | + | + | + | + | + |
| @deprecated | Рекомендует не использовать | + | + | + | + | + | + |
| {@docRoot } | Путь документации | + | + | + | + | + | + |
| @exception | Исключение | - | + | - | + | - | - |
| @link | Вставляет встроенную ссылку | + | + | + | + | + | + |
| {@inheritDoc} | Наследует комментарий от суперкласса | - | + | - | - | - | - |
| {@literal} | Отображает информацию в исходном виде | + | + | + | + | + | + |
| @param | Документирует параметр метода | + | + | - | + | + | - |
| @return | Документирует возвращаемое значение метода | - | + | - | - | - | - |
| @see | Документирует ссылку | + | + | + | + | + | + |
| @serial | Документирует сериализуемое поле | - | - | + | - | - | - |
| @serialData | Документирует данные, записанные методами writeObject и writeExternal. | - | + | - | - | - | - |
| @serialField | Документирует компонент ObjectStreamField | - | - | + | - | - | - |
| @since | Прошлую версию | + | + | + | + | + | + |
| @throw | Документирует исключение | - | + | - | + | - | - |
| {@value} | Отображает значение следующей за ним константы, которой должно являться поле static | - | - | + | - | - | - |
| @version | Версия | + | - | - | - | + | + |

Пример использования дескрипторов и генерации javadoc.

Ниже представлен пример использования многих выше описанных дескрипторов. Для того чтоб сгенерировать документ javadoc в IntelliJ IDEA в проекте перейти в Tools -> кнопка Generate Javadoc.

package myPackage;

/\*\*

\* Tests tools for generating javadoc documents

\* @author Alexander Curly

\* @version 1.2

\* @since 1.0

\*/

public class Main {

/\*\*

\* New constructor

\* @since 0.2

\* @param value input parameter

\* {@code this.value=value;}

\*/

public Main (int value){

this.value=value;

}

/\*\*

\* Old constructor

\* @since 0.1

\* @deprecated better use first constructor

\* \*/

public Main(){

}

/\*\*

\* he store my value.

\*/

int value;

/\*\*

\* Get value

\* @deprecated

\* @return value

\*/

public int oldGetValue(){

return value;

}

/\*\*

\* Get value

\* @return value from class Main.

\*/

public int getValue() {

return value;

}

/\*\*

\* This method set value field value

\* @param value when was setting to value

\*/

public void setValue(int value) {

this.value=value;

}

/\*\*

\* Make object string description

\* @return Object string description

\* @see Main#getValue()

\*/

@Override

public String toString() {

return this.getClass().getCanonicalName()+

": "+this.getValue();

}

/\*\* Parse string to int for value.

\* @param str this line will be turned into a number

\* @return the method returns a string passed to the parameters

\* @see Main#setValue(int)

\*/

public String parseToValue(String str) {

setValue(Integer.parseInt(str));

return str;

}

/\*\*

\* Started program.

\* @param args it is input command

\*/

public static void main(String[] args) {

}

}

Для генерации javadoc о пакете следует использовать package-info.java. Для того чтоб создать такой файл следует воспользоваться src (или пакет) -> кнопка package-info.java. Есть второй вариант с созданием HTML файла и самостоятельно описать пакет.

/\*\*

\* It's my package for examples

\* @author Alex Curly

\* @version 1

\* @since 0.9

\* @deprecated

\*

\*/

package myPackage;

Лямбда выражений.

Основы.

*Лямбда выражений* – это анонимный метод, который служит для реализации функционального интерфейса, а не для собственного выполнения. Он в своем роде анонимный класс для реализации метода интерфейса.

*Функциональный интерфейс* – это интерфейс, который содержит один абстрактный метод, который может определяться через лямбду. Нужно учитывать то, что он может содержать еще методы по умолчанию и статические методы. Такие методы называют SAM один абстрактный метод.

В функциональном интерфейсе определяется *целевой тип*. Это означает то, что лямбда выражений должна подходить к целевому типу.

Лямбда выражений.

Лямбда представляется лямбо-оператор в виде « -> ». Она разделяется на левую часть, где параметры для лямбды и правую сторону, где *тело лямбы выражений*. Лямбда выражений может быть блочным или строчным. Пример простой лямбды:

() -> 123.4

Здесь лямбда просто возвращает 123.4. Эта лямбда аналогична следующему методу:

double getValue() { return 123.4; }

Ниже представлен пример блочной лямбды выражений:

() -> { return (120+3.4);}

Ниже представлен пример передачи параметров лямбде:

(str) -> str.toUpperCase()

Функциональные интерфейсы.

Функциональный интерфейс определяет параметр и тип лямбды. Это значит если единственный абстрактный метод void то и лямбда не может возвратить, что-либо.

Также функциональный интерфейс может содержать и статические методы, как и методы по умолчанию. Функциональные интерфейсы помечаются аннотацией @FunctionalInterface.

Ниже приведен пример объявления функционального интерфейса, с абстрактным методом, который поднимает входящую строку:

@FunctionalInterface

interface Functional{

String toUpperCaseString(String str);

}

Ниже представлена лямбда определяющая SAM:

Functional fun= (str) -> str.toUpperCase ();

Данная лямбда принимает в параметры str. Здесь параметр известен, так как он определяется в целевом типе. Ниже представлен полный пример лямбды выражений и функционального интерфейса:

package myPackage;

public class Main {

public static void main(String[] args) {

Functional fun= (str) -> str.toUpperCase();

System.out.println(fun.toUpperCaseString("Это строка будет поднята."));

}

}

@FunctionalInterface

interface Functional{

String toUpperCaseString(String str);

}

Обобщенные функциональные интерфейсы

Лямбда не может быть обобщенным, так как в процессе стирки типов потеряется целостность типов. Но можно обобщить функциональный интерфейс, а далее использоваться типом.

Ниже представлен пример в виде лямбды, находящей среднеарифметическое число:

package myPackage;

public class Main {

public static void main(String[] args) {

Average<Integer> average= (num) -> {

int result=0;

for (int i = 0; i < num.length; i++) {

result += num[i];

}

return result/num.length;

};

System.out.println(average.average(new Integer[]{1, 3, 5, 15}));

}

}

@FunctionalInterface

interface Average<T extends Number>{

T average(T[] val);

}

Передача лямбды выражений в параметры.

Передача лямбды выражений в параметры это очень важное добавлений в Java 8. Это, к примеру, позволяет код, передаваемый в параметры его же и выполнить.

Можно передать лямбду или определить сразу же в параметры. Принимающий аргумент должен быть экземпляром функционального интерфейса.

Ниже представлен пример передачи лямбды выражений в параметры:

package myPackage;

public class Main {

static String StringOperation(Function function, String inStr){

return function.fun(inStr);

}

public static void main(String[] args) {

Function reserve= (str) -> {

String res= " ";

int i;

for(i=str.length()-1; i>=0; i--)

res += str.charAt(i);

return res;

};

System.out.println("Перевернутая строка: "+StringOperation(reserve, "Кудрявцев"));

System.out.println("Поднятая строка: "+StringOperation((str) -> str.toUpperCase(), "нижняя строка"));

}

}

@FunctionalInterface

interface Function{

String fun(String str);

}

Лямбды и исключения.

Лямбда может также генерировать исключения. Для этого в операторе throws у абстрактного метода следует перечислить исключения (если они проверяемые).

Ниже представлен пример лямбды и исключений:

package myPackage;

public class Main {

public static void main(String[] args) {

Average average= (num -> {

if(num.length == 0) {

throw new EmptyArrayException();

}

double result=0;

for (int i = 0; i < num.length ; i++) {

result += num[i];

}

return result/num.length;

});

double res=0;

double res1=0;

try {

res=average.average(new double[]{1.0, 23.5});

res1=average.average(new double[]{});

} catch (EmptyArrayException e) {

System.out.println("Ошибка: "+e);

}

System.out.println("Среднеарифметическое массива: "+res+", "+res1);

}

}

@FunctionalInterface

interface Average{

double average(double [] num) throws EmptyArrayException;

}

class EmptyArrayException extends Exception{

EmptyArrayException() {

super("Массив пуст”);

}

}

Захват переменных.

У лямбд есть ряд ограничений на захват переменных:

* Нельзя определять локальные переменные.
* Все локальные переменные в лямбдах final.
* Оператор this в лямбдах отсутствует.
* Можно объявить переменную экземпляра.
* Можно определить только статические переменные.

Ссылки на метод.

*Ссылки на методы* позволяет ссылаться на метод, не вызывая его. Связан с лямбдами выражений так, как должен поддерживать целевой тип. Для него введен новый синтаксис в виде « :: ». У ссылок на методы делятся на разные виды.

Ссылка на статический метод.

Для создания ссылки на статический метод используется следующий синтаксис:

*имя\_класса*::*имя\_метода*

Такой ссылкой можно пользоваться везде, где не нарушатся целевой тип. Ниже представлен пример ссылки на статический метод:

package myPackage;

public class Main {

static String StringOp(StringFunc fun, String str){

return fun.fun(str);

}

public static void main(String[] args) {

System.out.println(

StringOp(MyStringMethod::reserve,

"Перевернутая строка"));

}

}

@FunctionalInterface

interface StringFunc{

String fun(String str);

}

class MyStringMethod{

public static String reserve(String str){

String res=" ";

for (int i = str.length()-2; i >=0 ; i--)

res +=str.charAt(i);

return res;

}

}

Ссылки на метод экземпляра.

Чтоб создать ссылку на метод экземпляра следует воспользоваться следующим синтаксисом:

*имя\_экземпляра* :: *имя\_метода*

Ниже приведен пример переделанной выше программы с использованием ссылки на метод экземпляра:

package myPackage;

public class Main {

static String StringOp(StringFunc fun, String str){

return fun.fun(str);

}

public static void main(String[] args) {

MyStringMethod objStr=new MyStringMethod();

System.out.println(

StringOp(objStr::reserve,

"Перевернутая строка"));

}

}

@FunctionalInterface

interface StringFunc{

String fun(String str);

}

class MyStringMethod{

public String reserve(String str){

String res=" ";

for (int i = str.length()-2; i >=0 ; i--)

res +=str.charAt(i);

return res;

}

}

Ссылка на метод экземпляра с использованием объекта такого класса.

Иногда требуется указать метод экземпляра с

Ссылки на конструкторы.

Ссылки на методы можно создавать как и ссылки на методы. Их можно присвоить к любому функциональному интерфейсу, определяющий метод совместимый с конструктором. Ниже представлен синтаксис ссылки конструктор:

*имя\_класса* :: new

Ниже представлен пример ссылки на конструктор:

package myPackage;

public class Main {

public static void main(String[] args) {

FunctionUse factory=Use::new;

Use user=factory.creatUse("Саша", 13);

Use user1=factory.creatUse("Рома", 36);

System.out.println(user.toString());

System.out.println(user1.toString());

System.out.println("Они одинаковы: "+user.equals(user1));

System.out.println("По умолчанию: "+new Use().toString());

}

}

class Use{

private String name;

private int year;

Use(){

this.name="Джорш";

this.year=23;

}

Use(String name, int year){

this.name=name;

this.year=year;

}

@Override

public String toString() {

return name+", "+year;

}

@Override

public boolean equals(Object obj) {

if(this==obj)return true;

return false;

}

}

@FunctionalInterface

interface FunctionUse{

Use creatUse(String name, int year);

}

Ссылки на обобщенные методы.

Ссылками на методы можно пользоваться и обобщенные методы. Главное чтоб он не нарушал целевой тип. Ниже представлен пример обобщенной ссылки на метод:

package myPackage;  
  
import java.util.ArrayList;  
import java.util.Collections;  
  
public class Main {  
  
 static int maxVal(MyClass one, MyClass two){  
 return one.getVal()-two.getVal();  
 }  
  
 public static void main(String[] args) {  
  
 ArrayList<MyClass> list=new ArrayList<>();  
 list.add(new MyClass(12));  
 list.add(new MyClass(23));  
 list.add(new MyClass(1));  
 list.add(new MyClass(-1));  
  
 MyClass myClass= Collections.*max*(list, Main::*maxVal*);  
  
 System.*out*.println("Максимальное значение: "+myClass.getVal());  
  
 }  
}  
  
class MyClass{  
  
 private int val;  
  
 MyClass(int val) {  
 this.val=val;  
 }  
  
 public int getVal() {  
 return val;  
 }  
}

Пакет java.lang.function.

После внедрений лямбы и ссылок на методы был внедрен новый пакет java.lang.function. Он позволяет не определять свои функциональные интерфейсы. В нем есть следующие интерфейсы:

Function <T, R> - Возвращает операцию над объектом T, возвращая R. Определяет метод apply().

Consumer <T> - Выполняет операцию над объектом T. Содержит метод accept().

Supplier <T> - Возвращает объект T. Содержит метод get().

BinaryOperator <T> - Выполняет логическую операцию над двумя объектами T. Содержит метод apply().

UnaryOperator <T> - Выполняет унарную операцию над объектом T. Содержит метод apply().

Predicate <T> - Определяет подходит ли объект T к определенному условию. Содержит метод test().

Ниже представлен пример использования интерфейса Function:

package myPackage;

import com.sun.istack.internal.NotNull;

import java.util.function.Function;

public class Main {

public static void main(String[] args) {

@NotNull

Function<String, Integer> fun= (str) -> {

int result=0;

try {

result = Integer.parseInt(str);

}catch (NumberFormatException exc){

System.out.println("Ошибка: "+exc);

return -1;

}

return result;

};

System.out.println("Числовое представление строки: "+fun.apply("1225"));

}

}

Stream API.

*Поток данных* в Stream API определяется как канал передачи данных т.е. последовательность объектов. Он предназначается от работы с коллекциями и массивами. Он представляет легкую сортировку, фильтрацию, накопления, распараллечивания и т.д. коллекций. Stream API отличается от потоков ввода-вывода и потоков Thread или Runnable.

У стрима должен быть *источник*. Чтоб создать источник можно застримить стрим, написать функцию генерации чисел и т.д.

Стрим также состоит из *промежуточных операций* и одной *терминальной операции*. Их отличия в том, что промежуточные возвращают новый стрим, а терминальные не возвращают.

Все стримы *ленивы*. Это обозначает, что все промежуточные операции выполнятся только во время терминальной операции.

Стримы не могут хранить данные, а только перемещаются, фильтруются, сортируются, обрабатываются и т.д.

Некоторые операции *коротко замкнутые*. Это значит, что они не перебирают весь стрим, так как уже знают результат. К примеру множество чисел необязательно перебирать метод count() уже знает количество элементов в множестве.

Нужно учитывать, что все операции с Consumer лучше не использовать в реальном коде, а только в тесте. Эта причина в том, что такие операции имеют побочные значения, когда Stream API задумался как без побочный.

Интерфейс потока.

Для работы со стримами предназначен пакет java.util.stream. В нем определен главный интерфейс BaseStream.

В нем определяются основные функциональные возможности Stream API. Интерфейс BaseStream является обобщенным и объявляется нижним образом:

interface BaseStream <T, S extends BaseStream <T, S>>

где T тип данных в стриме, а S обозначает данные расширяемые BaseStream. Следует сразу отметить, что он расширяет интерфейс AutoCloseable и поэтому со Stream API можно использовать с try с ресурсами.

Ниже представлены все методы, определенные в интерфейсе BaseStream:

void close() – Закрывает этот поток, вызывая вызовы для всех обработчиков этого потока.

boolean isParallel() – Возвращает, будет ли этот поток, если операция терминала будет выполняться, будет выполняться параллельно.

Iterator<T> iterator() – Возвращает итератор для элементов этого потока.

S onClose(Runnable *поток*) – Возвращает эквивалентный поток с дополнительным обработчиком.

S parallel() – Возвращает эквивалентный поток, параллельный.

S sequential() – Возвращает эквивалентный поток, который является последовательным.

Spliterator<T> spliterator() – Возвращает разделитель для элементов этого потока.

S unordered() – Возвращает эквивалентный поток, который неупорядочен.

Производным от BaseStream является интерфейс Stream. Он объявляется приведенным образом:

interface Stream<T>

Где T тип элементов в стриме. В интерфейсе Stream определятся промежуточные и терминальные операции. Все избранные промежуточные операции перечислены ниже:

<R> Stream<R> map (Function<? super T, ? extends R> *отображение* ) – создает стрим с указанным *отображением*.

DoubleStream mapToDouble (ToDoubleFunction<? super T> *отображение*) – создает стрим, где элементы Double с указанным *отображением*.

IntStream mapToInt (ToIntFunction<? super T> *отображение*) – создает стрим, где элементы Integer с указанным отображением.

Stream <T> sorted() – сортирует элементы в стриме.

Stream<T> filter (Predicate<? super T> *предикат*) – фильтрует стрим с указанным *предикатом*.

Stream<T> limit (long *количество*) – указывает максимальное *количество* элементов.

…

Ниже представлен список избранных терминальных операций:

long count () – считает количество элементов стрима.

boolean anyMatch (Predicate<? super T> *предикат*) - возвращает true, если *предикат* верен для одного элемента.

boolean allMatch (Predicate<? super T> *предикат*) – возвращает true, если *предикат* верен для всех элементов.

boolean noneMatch (Predicate<? super T> *предикат*) - возвращает true, если *предикат* не верен для всех элементов.

void forEach (Consumer<? super T> *действие*) – выводит каждый элемент после операций по указанному *действию*.

Optional<T> max (Comparator <? super T> *компаратор*) – возвращает максимальное число по *компаратору*.

Optional<T> min (Comparator <? super T> *компаратор*) – возвращает минимальное число по

*компаратору*.

T rebuce (T *идентичное\_значение*, BinaryOperator<T> *накопитель*) – возвращает результат исходя из элементов в вводящем потоке.

Object [] toArray() – создает массив элементов в стриме.

<R, A> R collect(Collector<? super T, A, R> *коллектор*) – накапливает элементы в изменяемый контейнер. Где R тип результата, A внутренний накапливаемый тип, T тип элементов в стриме.

Для создания стрима следует найти или создать источник. По отношению стрим не может модифицировать источник и поэтому главнее источник. Есть различные способы задать источник:

* Stream.empty () – создает пустой стрим.
* Stream.of (x, h, q) – создает поток с данными элементами.
* Stream.ofNullable (x) – создает стрим с одним элементом.
* Stream.generate (*функция*) – создает стрим с элементы из *функции*, например Math:: random.
* Stream.iterate (*первый\_элемент*, *функция\_изменитель*) – создает стрим с 1 элементом и функцией, которая изменяет 1 элемент.
* collection.stream () – создает стрим из элементов определенной коллекции.
* Array.stream (array) - создает стрим из массива. Он не может применять массив из байтов.
* …

Простой пример Stream API.

Ниже представлен простой пример Stream API. Здесь массив строк проверяется на наличие пустого элемента, коллекция Integer,в которой из всех четных чисел находят большее, коллекцию сортирует по возрастанию и находит квадратные корни, применяя метод mapToDouble(), умножая корни на сто. Код:

package myPackage;

import java.util.ArrayList;

import java.util.Iterator;

import java.util.Optional;

import java.util.stream.Stream;

public class Main {

public static void main(String[] args) {

String[] arr= new String[]{"Александр", "Коля", "Алексей", ""};

Stream.of(arr).

filter(ae -> !ae.isEmpty()).

forEach((ae) -> System.out.println(ae));

ArrayList<Integer> list=new ArrayList<>();

list.add(new Integer(123));

list.add(new Integer(321));

list.add(new Integer(12));

list.add(new Integer(456));

list.add(new Integer(654));

Optional<Integer> res=list.stream().

filter(val -> val % 2==0).

max(Integer::compare);

// Вывести максимальное число, а если оно не найдено -1

System.out.println("Максимальное число: "+res.orElse(-1));

Iterator<Integer> sort\_collection=

list.stream().

sorted().

iterator();

System.out.println("Сортированная коллекция: ");

while (sort\_collection.hasNext()) {

System.out.print(sort\_collection.next()+" ");

}

System.out.println("");

// Найти квадратный корень, если он double умножить на 100

list.stream().

map(val -> Math.sqrt(val)).

mapToDouble((val) -> val \* 100).

forEach((ae) -> System.out.println("Квадратный корень: "+ae));

}

}

Метод distinct().

Метод distinct() убирает повторяющие элементы и возвращает стрим без повторяющихся элементов. Полное объявление метода:

Stream <T> distinct()

Метод skip().

Метод skip() пропускает n количество стрима. Полное объявление:

Stream<T> skip()

Метод peek().

Метод peek() возвращает поток, состоящий из элементов этого потока, дополнительно выполняющий предоставленное действие для каждого элемента. Отличается от forEach() в том, что он относится к промежуточным операциям. Полное объявление метода:

Stream <T> peek(Consumer <? super T> *действие*)

Пример методов skip(), peek() и distinct().

Ниже представлен пример использования методов skip(), peek() и distinct(). Здесь пропускается первые четыре элемента, после в методе peek() находится квадратный корень элемента и присваивает его статичной переменной и в конце все повторяющие элементы убираются. Код:

package myPackage;

import java.util.stream.Stream;

public class Main {

static double i;

public static void main(String[] args) {

Stream.of(0, 0, 0, 0, 1, 3, 5, 12, 5 , 234, 1).

skip(4).

peek(val -> i = Math.sqrt((double) val)).

distinct().

forEach(val -> System.out.println(val+", квадратный корень "+i+"; "));

}

}

Метод flatMap()

Метод flatMap() схож с методом map(), но с отличием он позволяет преобразовать элемент в ноль, многомерный массив в обычный массив. Метод flatMap() можно показать так:

[ [1, 3, 8, 11] [12, 3, 5] [5, 15] ] -> flatMap() -> [1, 3, 8, 11, 12, 3, 5, 5, 15]

Ниже представлен пример его использования:

package myPackage;

import java.util.stream.IntStream;

import java.util.stream.Stream;

public class Main {

public static void main(String[] args) {

Stream.of(1, 4, 0, 2, 8).

flatMapToInt(val -> IntStream.range(0, val)).

forEach(value -> System.out.print(value+" "));

}

}

Метод allMatch()

Данный метод проверяет все элементы стрима на удовлетворение определенному предикату. Полное объявление метода:

boolean allMatch (Predicate <? super T> *предикат*)

Метод anyMatch()

Данный метод проверяет один элемент стрима на удовлетворение определенному предикату. Полное объявление метода:

boolean anyMatch (Predicate <? super T> *предикат*)

Метод noneMatch()

Данный метод проверяет все элементы на не удовлетворению определенному предикату. Полное объявление метода:

boolean noneMatch (Predicate <? super T> *предикат*)

Пример методов allMatch(), anyMatch() и noneMatch().

Ниже представлен пример использования выше приведенных методов:

package myPackage;

import java.util.ArrayList;

public class Main {

public static void main(String[] args) {

ArrayList<Integer> list=new ArrayList<>();

list.add(2);

list.add(4);

list.add(12);

list.add(23);

boolean res1 = list.stream().

allMatch(val -> val <= 4);

System.out.println("Удовлетворяет ли вся коллекция по предикату: "+res1);

res1=list.stream()

.anyMatch(val -> val==(10+2));

System.out.println("Удовлетворяет ли элемент коллекции по предикату: "+res1);

res1=list.stream().

noneMatch(val -> val ==0);

System.out.println("Не удовлетворяет ли коллекция предикату: "+res1);

}

}

Операции сведения.

Операции, которые возвращают одно нужное значение называются *операции сведений*. К ним относят min(), max(), count() и reduce().

Методы min() и max() возвращают соответственно минимальное и максимальное значение. Метод count() возвращает количество элементов в потоке.

Метод reduce() обобщает все верхнее. Он возвращает определенное значение. У него определено два объявления:

Optional<T> reduce (BinaryOperator<T> *накопитель*)

T reduce (T *значение\_идентичности*, BinaryOperator<T> *накопитель*)

Где *накопитель* функция, которая оперирует двумя значения. Где *значение\_идентичности* значение, которое в случае 0 + x = x будет *значение\_идентичности* \* x = x.

Операция накопления должна удовлетворять следующим правилам:

* Без сохранения состояния.
* Без вмешательства в источник.
* Ассоциативность.

Без сохранения состоятельности обозначает, что операция сведения обозначает состояние и каждый элемент обрабатывается отдельно. Где ассоциативность это математическая ассоциативность т.е. 12 \* (8 \* 5) может быть представлена (12 \* 8) \*5.

Ниже представлен пример операции сведения reduce():

package myPackage;

import java.util.ArrayList;

import java.util.Optional;

public class Main {

public static void main(String[] args) {

ArrayList<Integer> list=new ArrayList<>();

list.add(123);

list.add(63);

list.add(34);

list.add(12);

Optional<Integer> res=list.stream().

reduce((a, b) -> a\*b);

System.out.println("Произведение в виде Optional<Integer>: "+res.orElse(-1));

int val=list.stream().

reduce(1, (a, e) -> a\*e);

System.out.println("Произведение в виде Integer: "+val);

int val2=list.stream().

reduce(2, (a, e) -> a\*e);

System.out.println("Произведение в виде Integer c значением идентичности 2: "+val2);

System.out.println();

list.stream().

forEachOrdered((num) -> System.out.print(num+" "));

int evenProduct=list.stream().reduce(1, (a, e) -> {

if(e % 2==0) return a \* e;

else return a;

});

System.out.println("\n"+evenProduct);

}

}

Параллельные стримы.

В java 8 представлен удобный каркас для создания многоядерных программ. Stream API также можно распараллелить. Для того чтоб задать параллельную обработку можно использовать два метода.

Чтоб первоначально создать параллельный стрим у коллекции следует использовать метод parallelStream(). А если нужно последовательный стрим преобразовать в параллельный стрим можно воспользоваться методом parallel().

После создания параллельного стрима все операции будут параллельными, при условии чтоб среда многоядерная.

Ниже представлен первая операция сведения из верхней программы в параллельном стриме:

package myPackage;

import java.util.ArrayList;

public class Main {

public static void main(String[] args) {

ArrayList<Double> list=new ArrayList<>();

list.add(12.4);

list.add(2.9);

list.add(17.4);

list.add(5.1);

double res=list.parallelStream().reduce(1.0,

(a, b) -> a \* Math.sqrt(b), (a, b) -> a \* b);

System.out.println(res);

}

}

Здесь форма метода reduce() использует объединитель и выглядит нижним образом:

<U> U reduce (U *значение\_идентичности*,

BiFunction<U, ? super T, U> *накопитель*, BinaryOperator<U> *объединитель*)

Здесь *объединитель* объединяет результаты из других потоков. Также можно из параллельного стрима сделать последовательный стрим. Для этого следует вызвать метод sequential().

С параллельными и последовательными стримами возникает вопрос об упорядоченности элементов. Если источник упорядочен, то стрим упорядочен, но порой можно добиться большей производительности не упорядоченности. В такой случае отдельные части будут обрабатываться отдельно. Для того чтоб задать не упорядоченный стрим можно воспользоваться методом unordered().

Накопление

Стрим получает поток данных из источника, но возможно и наоборот. Эта операция называется *накопление*.

Для накопления в стриме определен метод collect():

<R, A> R collect(Collector<? super T, A, R> *накопитель*)

Здесь R тип результата, T тип входных данных, A тип накапливаемых данных внутри. Параметр *накопитель* обозначает функцию порядок обработки коллекции.

В JDK 8 определенно два коллектора. Первый для списков, а второй для множеств. Все они определенны в классе Collectors в виде статических методов:

static <T> Collector<T, ?, List<T>> toList()

static <T> Collector<T, ?, Set<T>> toSet()

Все выше сказанное демонстрируется в приведенной ниже программе:

package myPackage;

import java.util.ArrayList;

import java.util.List;

import java.util.Set;

import java.util.stream.Collectors;

public class Main {

public static void main(String[] args) {

ArrayList<NamePhoneEmail> myList=new ArrayList<>();

myList.add(

new NamePhoneEmail("Саша", "+791563451", "@Барсук007"));

myList.add(

new NamePhoneEmail("Мамкин Хакер", "Я тебя по айпи взломаю", "@mama\_i\_like"));

myList.add(

new NamePhoneEmail("Свидетель Иегова", "555-5555", "@Pohol\_Armagedon"));

myList.add(new NamePhoneEmail("Сатана", "666", "@Иди\_грешник"));

List<NamePhone> list=myList.stream().

map(a -> new NamePhone(a.name, a.phone)).

collect(Collectors.toList());

for (NamePhone people : list) {

System.out.println(people.name+": "+people.phone);

}

Set<NamePhone> setOfPeople=myList.stream().

map((a) -> new NamePhone(a.name, a.phone)).

collect(Collectors.toSet());

System.out.println("\nДанные из множества: \n");

for (NamePhone people : setOfPeople) {

System.out.println(people.name+": "+people.phone);

}

}

}

class NamePhoneEmail{

String name;

String phone;

String email;

NamePhoneEmail(String name, String phone, String email){

this.name=name;

this.email=email;

this.phone=phone;

}

}

class NamePhone{

String name;

String phone;

NamePhone(String name, String phone){

this.name=name;

this.phone=phone;

}

}

Третья форма метода collect() позволяет управлять над процессом накопления:

<R> R collect (Supplier<R> *адресат*, BiConsumer<R, ? super T> *накопитель*,

BiConsumer<R, R> *объединитель*)

Где *адресат* порядок создания объекта, хранящий результат. А функции *накопителя* и *объединителя* аналогичны с методом reduce (). Если нужно коллекция LinkedList, то в верхнюю программу можно внести следующий код:

LinkedList<NamePhone> list1=myList.stream().

map(a -> new NamePhone(a.name, a.phone)).

collect(() -> new LinkedList<>(),

(l, e) -> l.add(e),

(listA, listB) -> listA.addAll(listB));

System.out.println("\nДанные из LinkedList: \n");

for (NamePhone people: list1) {

System.out.println(people.name+": "+people.phone);

}

Также можно использовать ссылки на методы и верхний код превратится:

LinkedList<NamePhone> set=myList.stream().

map(a -> new NamePhone(a.name, a.phone)).

collect(

LinkedList::new,

LinkedList::add,

LinkedList::addAll);

Ниже представлен пример создания своих коллекторов. Он представлен в виде коллектора находящий максимальный и минимальный элемент по указанному компаратору.

package myPackage;

import java.util.\*;

import java.util.function.BiFunction;

import java.util.stream.Collector;

import java.util.stream.Stream;

public class Main {

static<T, R> Collector<T, ?, Optional<R>> minMax(

Comparator<? super T> comparator,

BiFunction<? super T, ? super T, ? extends R> finisher){

class Accumulator{

T min;

T max;

boolean present;

void add(T value){

if(present){

if(comparator.compare(value, min) <0) min= value;

if(comparator.compare(value, max) >0) max= value;

}else {

min=max=value;

present=true;

}

}

Accumulator combiner(Accumulator other){

if(!other.present) return this;

if(!present) return other;

if(comparator.compare(other.min, min) <0) min=other.min;

if(comparator.compare(other.max, max) >0) max=other.max;

return this;

}

}

return Collector.of(Accumulator::new,

Accumulator::add,

Accumulator::combiner,

accumulator -> accumulator.present

? Optional.of(finisher.apply(accumulator.min, accumulator.max))

: Optional.empty());

}

public static void main(String[] args) {

Optional<String> optional = Stream.of("one", "two", "three", "four", "five", "six", "seven", "eight").

collect(minMax(Comparator.comparingInt(String::length), (min, max) -> min+"|"+max));

System.out.println(optional.get());

}

}

Сначала разберемся, как работает коллектор, т.е. интерфейс коллектора. Он состоит из через четырех методов.

* Supplier<A> supplier () – создание накопителя.
* BigConsumer<A, T> accumulator () – добавление в накопитель.
* BinaryOperator<A> combiner () – склеивает два накопителя.
* Function<A, R> finisher () – преобразовывает накопитель в результат.
* Set<Characteristics> characteristics () – флаги (ерунда).

Такое уже создавалось раннее в виде третей форме метода collect(). Здесь комбайнер склеивает результат, но склеивать результат нужно в случае параллельного стрима, но передавая null генерируется NullPointerException. В документации написано, что писать комбайнер обязательно, но если стрим не должен, быть параллельным можно написать лямбду генерирующую исключение. Здесь преобразователь тривиальная вещь. Она требуется если результат накопления требуется привести к чему либо и т.д. Флаги же можно назвать ненужными.

В общем чтоб создать свой коллектор следует реализовать все выше методы и вернуть коллектор статическим методом Collector.of(). Все это объединяют в один метод, в моем случае minMax().

Внутренний класс Accumulator содержит минимальное и максимальное число, а также флажком, который содержит значение на пуст и не пуст.

Метод combiner() склеивает два аккумулятора в один. Если other пуст возвращает вызывающий или наоборот.

Методы findFirst() и findAny()

Метод findAny() возвращает первый попавшийся элемент стрима. В параллельных стримах это может быть действительно любой элемент, который лежал в разбитой части последовательности. Полное объявление:

Optional findAny()

Метод findFirst() возвращает гарантированно первый элемент стрима. Полное объявление метода:

Optional findFirst()

Ниже представлен пример использования данных методов:

package myPackage;

import java.util.ArrayList;

import java.util.Optional;

public class Main {

public static void main(String[] args) {

ArrayList<String> list=new ArrayList<>();

list.add("Меркурий");

list.add("Венера");

list.add("Земля");

list.add("Марс");

list.add("Юпитер");

list.add("Сатурн");

list.add("Уран");

list.add("Нептун");

Optional<String> first=list.stream().findFirst();

System.out.println("Первый элемент: "+first.get());

Optional<String> anyStr=list.parallelStream().findAny();

System.out.println("Случайный элемент: "+anyStr.get());

}

}

Операции с примитивами.

Для примитивов определено три интерфейса стрима для примитивов. Ниже они перечислены:

* IntStream
* LongStream
* DoubleStream

У них определен ряд интересных методов. Ниже они перечислены:

OptionalDouble average() – Возвращает среднеарифметическое чисел

sum() – возвращает суммирование всех чисел.

IntSummaryStatistics summaryStatistics() – возвращает объект IntSummaryStatistics.

Здесь показан класс IntSummaryStatistics, в нем определены методы удобные при составлении статистики. Метод getMin() возвращает минимальное число, getSum() возвращает суммирование всех чисел и т.д.

Ниже представлен пример операций с примитивами:

package myPackage;

import java.util.IntSummaryStatistics;

import java.util.stream.DoubleStream;

import java.util.stream.IntStream;

import java.util.stream.LongStream;

public class Main {

public static void main(String[] args) {

double average = DoubleStream.of(1.3, 3.8, 34.3, 61.6).

average().

getAsDouble();

System.out.println("Среднеарифметическое чисел: " + average);

long sum = LongStream.range(2, 16).sum();

System.out.println("Сумма всех чисел: " + sum);

IntSummaryStatistics statistics =IntStream.rangeClosed(1, 20).

summaryStatistics();

System.out.format("%nМинимальное число: %d%n", statistics.getMin());

System.out.format("Максимальное число: %d%n", statistics.getMax());

System.out.format("Количество чисел: %d%n", statistics.getCount());

System.out.format("Среднеарифметическое чисел: %f %n", statistics.getAverage());

System.out.format("Сумма всех чисел: %d%n", statistics.getSum());

}

}

Применение Stream API

Ниже приведен пример применения Stream API:

package myPackage;

import java.util.\*;

import java.util.function.BiFunction;

import java.util.stream.Collector;

import java.util.stream.Collectors;

import java.util.stream.Stream;

public class Main {

static<T, R> Collector<T, ?, Optional<R>> minMax(

Comparator<? super T> comparator,

BiFunction<? super T, ? super T, ? extends R> finisher){

class Accumulator{

T min;

T max;

boolean present;

void add(T value){

if(present){

if(comparator.compare(value, min) <0) min= value;

if(comparator.compare(value, max) >0) max= value;

}else {

min=max=value;

present=true;

}

}

Accumulator combiner(Accumulator other){

if(!other.present) return this;

if(!present) return other;

if(comparator.compare(other.min, min) <0) min=other.min;

if(comparator.compare(other.max, max) >0) max=other.max;

return this;

}

}

return Collector.of(Accumulator::new,

Accumulator::add,

Accumulator::combiner,

accumulator -> accumulator.present

? Optional.of(finisher.apply(accumulator.min, accumulator.max))

: Optional.empty());

}

public static void main(String[] args) {

// Создать коллекцию, которая хранит всех студентов.

List<Student> students = Arrays.asList(

new Student("Alex", Speciality.Physics, 1),

new Student("Smite", Speciality.Biology, 4),

new Student("Julia", Speciality.Biology, 2),

new Student("Steve", Speciality.History, 4),

new Student("Mike", Speciality.Finance, 1),

new Student("George", Speciality.Biology, 2),

new Student("Richard", Speciality.History, 1),

new Student("Kate", Speciality.Psychology, 2),

new Student("Sasha", Speciality.ComputerScience, 4),

new Student("Maximilian", Speciality.ComputerScience, 3),

new Student("Tim", Speciality.ComputerScience, 5),

new Student("Ann", Speciality.Psychology, 1)

);

// Выводим всех студентов по их курсах.

System.out.println("Курсы и их студенты: ");

students.stream().

collect(Collectors.groupingBy(Student::getCourse)).

entrySet().

forEach(System.out::println);

// Вывести количество студентов на каждом

System.out.println("\nКоличество студентов на каждом курсе: ");

students.stream().

collect(Collectors.groupingBy(Student::getSpecial, Collectors.counting())).

forEach((s, count) -> System.out.println("На " + s + " курсе учатся: " + count));

// Все курсы в университете с студентами.

System.out.println("\nВсе курсы в университете, через запятую: ");

String res = students.

stream()

.map(student -> student.getName()).

collect(Collectors.joining(", "));

System.out.println(res+"\n");

// Вывести всех студентов и их курсов отсортировано

System.out.println("Студент\tспециальность");

students.stream().

distinct().

forEach(student -> System.out.println(student.getName()+"\t"+student.getSpecial()));

//Создать map, где ключ имя студента, а специальность значение.

Map<String, Speciality> map =students.stream().

collect(Collectors.toMap(

Student::getName,

Student::getSpecial));

// Используя свой коллектор, вывести максимального и минимального студента по имени

Optional<String> min\_max = students.stream().

collect(

minMax(Comparator.comparing

(Student::getName),

(max, min) ->

min.getName()+", "+min.getCourse()+"|"+max.getName()+", "+max.getCourse()));

System.out.println("\n"+"Максимальный и минимальный студент по имени: "+min\_max.get());

Lecturer biolLecture=new Lecturer(Speciality.Biology, "Роберт Кох");

biolLecture.addFromList(students);

Optional<String> min\_max\_biol= biolLecture.list()

.collect(minMax(

Comparator.comparing(Student::getName),

(max, min) -> min.getName()+"|"+max.getName()));

System.out.println("\n"+"Максимальный и минимальный студент биологии по имение: "+min\_max\_biol.get());

long lengthBiol=biolLecture.list().count();

System.out.println("Количество студентов на факультете биологии: "+lengthBiol);

}

}

class Student{

private Speciality special;

private int course;

private String name;

Student(String name, Speciality special, int course){

this.name=name;

this.special=special;

this.course=course;

}

@Override

public String toString() {

return "Студент "+getName()+" учится "+getSpecial()+". Курс "+getCourse();

}

public int getCourse() {

return course;

}

public String getName() {

return name;

}

public Speciality getSpecial() {

return special;

}

public void setName(String name) {

this.name = name;

}

public void setCourse(int course) {

this.course = course;

}

public void setSpecial(Speciality special) {

this.special = special;

}

}

class Lecturer{

private ArrayList<Student> students=new ArrayList<>();

private Speciality spec;

private String name;

Lecturer(Speciality spec, String name){

this.spec=spec;

this.name=name;

}

public String getName() {

return name;

}

public Speciality getSpec() {

return spec;

}

public void setName(String name) {

this.name = name;

}

public void setSpec(Speciality spec) {

this.spec = spec;

}

void add(Student student){

if(student.getSpecial()==spec) {

throw new ArrayStoreException();

}

students.add(student);

}

void addAll(Student... student){

for (int i = 0; i < student.length ; i++) {

if(student[i].getSpecial()==spec)

throw new ArrayStoreException();

students.add(student[i]);

}

}

void addFromList(List<Student> outList){

for (int i = 0; i < outList.size(); i++) {

if(outList.get(i).getSpecial() == spec)

students.add(outList.get(i));

}

}

public Stream<Student> list(){

return students.stream();

}

}

enum Speciality {

Biology, ComputerScience, Economics, Finance,

History, Physics, Psychology

}

Обработка символьных строк.

В Java *символьная строка* является последовательной строкой. Строки представляют собой объекты класса String. Он представляет удобные методы для работы с строками. Также объект String может создаваться многими способами.

Все объекты String не возможно изменить. Поэтому все методы изменяющие строку на деле возвращают новую строку. Но если требуется видоизменяющая последовательность символов то Java предлагает два класса: StringBuilder и StringBuffer.

Классы String, StringBuilder и StringBuffer определенны в пакете java.lang и все реализует интерфейс CharSequence. Выше классы объявлены с модификатором final и не могут иметь подклассов.

Класс String.

Избранные конструкторы и создание строки.

Для создания пустой строки следует воспользоваться стандартным конструктором. Например, ниже представлен создания экземпляра:

String s = new String ();

Для создания символьной строки инициализируемая массивом символов используется приведенный ниже конструктор:

String (char [] *массив*)

Ниже приведен пример создания символьной строчки, которая инициализируется массивом символов переданный в параметры:

char [] chars = {'S', 't', 'r', 'i', 'n', 'g'};

String s = new String(chars);

Также определяется похожий конструктор, но позволяющий дополнительно указать диапазон т.е. часть массива:

String (char [] *массив*, int *начало*, int *длина*)

Ниже представлен пример инициализирования последовательности символов на основе указанного массива символов и указания части массива, которую будет использовать:

char [] chars = {'S', 't', 'r', 'i', 'n', 'g'};

String s = new String(chars, 1, 4);

Использую конструктор, который позволяет создать объект String на последовательности символов другой строки:

String (String *строка*)

Ниже представлен пример в виде создания последовательности строки на последовательности символов другого объекта String:

char [] chars = {'S', 't', 'r', 'i', 'n', 'g'};

String s = new String(chars, 1, 4);

String s2 = new String(s);

Можно обойтись без явного оператора new и использовать литералы. Для каждого строкового литера Java автоматически создает объект типа String. Ниже представлен пример использования строкового литерала:

String s = "String";

Также существуют конструкторы для 8-разрядных символов ASCII и указание части массива, для объекта StringBuilder и StringBuffer.

Кроме создания одной строки можно воспользоваться сцеплением строк. Ниже представлен пример сцеплении строки:

int age = 25;

String s = "Ему "+age+" лет.";

Избранные методы.

Получение длины строки.

Получить количество символов, из которых состоит последовательность символов можно легко получить методом length (). Полное объявление данного метода:

int length ()

В нижней программе демонстрируется использование метода length ():

public class Main {

public static void main(String[] args) {

String s = "Ему 25 лет.";

System.out.println(s);

System.out.println("Длина строки: "+s.length());

}

}

Методы charAt () и getChars()

Чтобы извлечь единственный символ из строки по индексу следует вызвать метод charAt() и указать *индекс*. Полное извлечение метода:

char charAt (int *индекс*)

Метод getChars () извлекает часть строки. Полное объявление приведено ниже, где *начало* и *конец* указывается индекс начала и конца подстроки, в параметры *массив* передается массив куда будет записываться подстрока.

void getChars (int *начало*, int *конец*, char [] *массив*, int *начало\_записи*)

Ниже представлен приме использования выше описанных методов:

public class Main {

public static void main(String[] args) {

String s = "Class String extends" +

" abstract class CharSequence ";

char ch = s.charAt(7);

System.out.println("Символ по индексу 7\'"+ch+"\'");

char [] chars = new char[6];

s.getChars(6, 12, chars, 0);

System.out.print("Часть строки: ");

for(char c : chars)

System.out.print(c);

}

}

Методы getBytes () и toCharArray ().

Метод toCharArray преобразует символы у вызывающей строчки в массив символов. Ниже приведено полное объявление данного метода:

char [] toCharArray ()

Метод getBytes () служит альтернативой toCharArray(). Он возвращает представление строки в виде байтов. Он часто используется для переноса строки в другие средства, которые не использует Unicode. Полное объявление:

byte [] getBytes ()

Ниже представлен пример использования выше описанных методов:

import java.util.Arrays;

public class Main {

public static void main(String[] args) {

String s = "Class String extends" +

" abstract class CharSequence";

System.out.println(Arrays.toString(s.getBytes()));

System.out.println(Arrays.toString(s.toCharArray()));

}

}

Методы equals (), equalsIgnoreCase() и операция ==.

Для сравнения двух строк следует вызвать equals(), а чтоб проверить игнорируя регистр следует воспользоваться equalsIgnoreCase(). Полные объявление методов:

boolean equals (Object *строка*)

boolean equalsIgnoreCase (Object *строка*)

Операция равенства ( == ) определяет ссылаются две ссылки на один тот же экземпляр т.е. адреса. Ниже приведен пример использования сравнения строк:

public class Main {

public static void main(String[] args) {

String s = "text";

String s1 = "Text";

String s2 = "Pepe";

String s3 = new String(s);

System.out.println(s + " .equals() "+s1+": "+s.equals(s1));

System.out.println(s1 + " .equalsIgnoreCase()"

+s+": "+s.equalsIgnoreCase(s1));

System.out.println(s + " .equals () "+s2+": "+s.equals(s2));

System.out.println(s + " == "+s3 +": "+(s == s3));

}

}

Метод compareTo().

Для сортировки и прочих операция не хватает знать равна ли одна строка другой строке. Метод compareTo() проверяет больше, меньше или равна ли строка с вызывающей строкой. Полное объявление метода:

int compareTo (String *строка*)

Он возвращает отрицательное число если вызывающая строка меньше *строка*; положительное число если вызывающая строка больше *строка* и если равны возвращается 0.

Ниже представлен пример использования compareTo() в сортировки пузырьком:

import java.util.Arrays;

public class Main {

static void sort(String [] array){

for(int i = 0; i<array.length; i++){

for (int j = i + 1; j < array.length; j++) {

if(array[j].compareTo(array[i]) < 0){

String s = array[i];

array[i] = array[j];

array[j] = s;

}

}

}

}

public static void main(String[] args) {

String [] arr = {"Java", "java.nio",

"java.text", "java.util",

"java.lang", "java.io",

"java.time", "java.net"};

System.out.println("Исходный массив: "+Arrays.toString(arr));

sort(arr);

System.out.println("Отсортированный массив: "+Arrays.toString(arr));

}

}

Методы startWith(), endsWith () и regionMatches().

Метод startsWith () определяет начинается ли вызывающий объект String с указанной *строки*. Метод endsWith () определяет заканчивается ли вызывающий объект String указанной *строкой*. Полные объявления данных методов:

boolean startsWith (String *строка*)

boolean endsWith (String *строка*)

Кроме выше описанной первой формы объявление у метода startWith присутствует и вторая форма позволяющая указать индекс подстроки:

boolean startsWith (String *строка*, int *индекс*)

Метод regionMatches() сравнивает на равенство часть вызывающей строки и часть сравниваемой строки. Присутствует версия с игнорированием регистра и без. Полные объявления:

boolean regionMatches (int *начало\_строки*, String *сравниваемая\_строка* , int *начало*, int *длина*)

boolean regionMatches (boolean *игнорировать\_регистра* , int *начало\_строки*,

String *сравниваемая\_строка* , int *начало*, int *длина*)

Ниже представлен пример использования выше описанных методов:

public class Main {

public static void main(String[] args) {

String s = "Foobar";

String s1 = "barFoo";

System.out.println("В начала строки идет Foo: "+s.startsWith("Foo"));

System.out.println("В конце строки идет bar: "+s.endsWith("bar"));

System.out.println("В начале подстроки с 2 индекса идет ob: “+s.startsWith(“ob”, 2));

System.out.println(«Части строк равны: «+

s.regionMatches(2, s1, 2, 4));

}

}

Поиск в строках.

Класс String предоставляет два метода для поиска в строке символа или подстроки. Первый это indexOf() находит первое вхождение символа или подстроки и возвращает индекс вхождения. У него определяется три формы:

int indexOf(char *символ*)

int indexOf(String *строка*)

int indexOf(String *строка*, int *начало*)

int indexOf(char *строка*, int *начало*)

Здесь параметры *символ* и *строка* определяется то, что ищется и если оно не будет найдено возвращается -1. Где *начало* обозначает индекс с которого начинается поиск.

Второй это lastIndexOf(), который находит последнее вхождение символа или подстроки. У него также определяется 4 формы и действуют они схоже как и indexOf().

Ниже представлен пример использования выше описанных методов:

public class Main {

public static void main(String[] args) {

String text = "We are hate a sally liberals!\n" +

"We are like a cool liberals!\n";

System.out.println(text);

System.out.println("indexOf('a') -> "+

text.indexOf('a'));

System.out.println("lastIndexOf('a') -> "+

text.lastIndexOf('a')+"\n");

System.out.println("indexOf(\"hate\") -> "+

text.indexOf("hate"));

System.out.println("lastIndexOf(\"sally\") -> "+

text.lastIndexOf("sally")+"\n");

System.out.println("indexOf(\"liberals\", 35) -> "+

text.indexOf("liberals", 35));

System.out.println("lastIndexOf(\"We\", 60) -> "+

text.lastIndexOf("We", 60));

}

}

Метод trim (), replace () и concat().

Метод trim () возвращает копию без пробелов в начале и в конце вызывающей строки. Он имеет следующею форму объявления:

String trim()

Метод concat () соединяет две строки. Он выполняет операцию сцепления и особых преимуществ не приносит. Полное объявление:

String concat (String *строка*)

Метод replace () заменяет все вхождения символа на другой символ. У него определяется две формы. Первая позволяет заменить символы, а вторая заменить CharSequence. Полное объявление метода:

String replace (CharSequence *исходная*, CharSequence *заменяемая*)

String replace (char *исходная*, char *заменяемая*)

Ниже представлен пример использования выше описанных методов:

import java.io.\*;

public class Main {

public static void main(String[] args) throws IOException {

BufferedReader reader=

new BufferedReader(

new InputStreamReader(System.in));

String str;

String text = "";

System.out.println("Для остановки введите Stop");

do{

System.out.println("Введите строку: ");

str=reader.readLine().

trim().

replaceAll("String", "Char Sequence");

System.out.println("Строка: "+str);

text= text.concat(str);

}while (!str.equalsIgnoreCase("Stop"));

System.out.println("Окончание чтения.");

System.out.println(text);

}

}

Преобразование данных метод valueOf ().

Статический метод valueOf () преобразует данные в удобные для чтения форме. Он перегружается для Object, int, short и прочие.

Методы toLowerCase () и toUpperCase ().

Метод toLowerCase () преобразует все символы из верхнего регистра на нижний регистр, а метод toUpperCase () делает обратное. У каждого из методов определенно по две формы. Одна – пустая, вторая позволяет указать Locale для регионов.

Ниже представлен пример использования выше описанных классов:

public class Main {

public static void main(String[] args) {

String s = "Изменение регистра в строке.";

System.out.println(s);

System.out.println("Верхний регистр: "+s.toUpperCase());

System.out.println("Нижний регистр: "+s.toLowerCase());

}

}

Соединение символьных строк.

В классе String определяется статический метод join(), который позволяет соединить две и более строк в одну целую через указанный *разделитель*. У него определяется две формы:

static String join (CharSequence *разделитель*, CharSequence … *строки*)

static String join (CharSequence , Iterator <? extends CharSequence> *строки*)

Ниже представлен пример соединения символьных строк:

public class Main {

public static void main(String[] args) {

String s =String.join(", ",

"Car Radio", "Migraine", "Ode To sleep");

System.out.println(s);

}

}

Класс StringBuffer.

Конструкторы.

У класса StringBuffer определенно 4 конструктора. Первый по умолчанию резервирует места для 16 символов. Если размер буфера не достаточно оно увеличивается на 16. Второй конструктор позволяет указать начальный размер буфера, третий начальное содержимое в виде объекта String. Четвертый создает буфер с начальным содержимым с любым содержащий последовательность символов. Конструкторы:

StringBuffer ()

StringBuffer (int *длина\_буфера*)

StringBuffer (String *строка*)

StringBuffer (CharSequence *строка*)

Избранные метода.

Методы length () и capacity ().

Метод length () позволяет узнать количество символов в буфере т.е. длину. Метод capacity () позволяет узнать объем или емкость, которая может содержать буфер. Полные определения символов:

int length ()

int capacity ()

Ниже представлен пример использования выше описанных методов:

public class Main {

public static void main(String[] args) {

StringBuffer buffer=new StringBuffer("Hello");

System.out.println("Буфер: "+buffer);

System.out.println("Длина -> "+buffer.length());

System.out.println("Объем -> "+buffer.capacity());

}

}

Метод ensureCapacity () и setLength ().

Методом ensureCapacity () можно указать емкость буфера. Ниже приведено полное объявление данного метода, где *минимальная емкость* обозначает минимальную емкость для буфера:

void ensureCapacity (int *минимальная\_емкость*)

Метод setLength () можно установить длину буфера. Если указанная длина меньше длины буфера –буфер укорачивается, а если больше дополняется пустыми значениями. Полное объявление:

void setLength (int *длина*)

Методы append () и insert ().

Метод append () добавляет строковое представление чего-либо. Он перегружается для всех примитивов, объект String и Object.

Метод insert () вставляет строку в другую. Он перегружается для примитивов, String, Object и CharSequence. Некоторые полные объявление, где *индекс* обозначает куда вставлять строку:

StringBuffer insert (int *индекс*, Object *объект*)

Ниже представлен пример использования выше описанных методов:

public class Main {

public static void main(String[] args) {

StringBuffer buffer=new StringBuffer("Hello");

buffer.append(" my friend!");

buffer.insert(9, "dear ");

System.out.println(buffer);

}

}

Метод reserve ().

Метод reserve () переворачивает содержимое StringBuffer и возвращает ее копию. Полное объявление данного метода:

StringBuffer reserve ()

Ниже представлен пример использования метода reserve ():

public class Main {

public static void main(String[] args) {

StringBuffer buffer=new StringBuffer("Hello StringBuffer");

System.out.println(buffer);

System.out.println(buffer.reverse());

}

}

Метод delete () и deleteCharAt ().

Метод delete () удаляет часть StringBuffer, а метод deleteCharAt() удаляет символ по указанному индексу. Полные объявления:

StringBuffer delete (int *начало*, int *конец*)

StringBuffer deleteCharAt (int *индекс*)

Ниже представлен пример использования выше описанных методов:

public class Main {

public static void main(String[] args) {

StringBuffer buffer=new StringBuffer("Example using method delete() и deleteCharAt().");

System.out.println("Изначальное содержимое: +"+buffer);

buffer.deleteCharAt(30);

System.out.println("После deleteCharAt(): "+buffer);

buffer.delete(8, 14);

System.out.println("После delete(): "+buffer);

}

}

Метод replace ().

Вызвав метод replace () можно заменить часть строки на другую указанную строку. Полное объявление данного метода.

StringBuffer replace (int *начало*, int *конец*, String *строка*)

Ниже представлен пример метода replace ():

public class Main {

public static void main(String[] args) {

StringBuffer buffer=new StringBuffer("Hello StringBuffer");

buffer.replace(0, 5, "Hi");

System.out.println(buffer);

}

}

Метод substring ()

Вызвав метод substring () можно получить часть строки. У него определено две версии. Первая позволяет указать начало подстроки, а второй еще позволяет указать конец подстроки. Полные объявления:

String substring (int *начало*)

String substring (int *начало*, int *конец*)

Ниже представлен пример использования данного метода:

public class Main {

public static void main(String[] args) {

StringBuffer buffer = new StringBuffer("Hello StringBuffer");

buffer.replace(0, 5, "Hi");

System.out.println(buffer);

}

}

Методы setCharAt () и charAt().

Метод setCharAt () заменяет символ на указанном индексе на новый символ. Узнать какой символ находится по индексу можно метод charAt(). Ниже представлен пример использования выше описанных методов:

public class Main {

public static void main(String[] args) {

StringBuffer buffer=new StringBuffer("text from ...");

buffer.setCharAt(5, 'c');

System.out.println("Символ \'"+buffer.charAt(5)+"\'");

System.out.println("StringBuffer "+buffer);

}

}

Метод getChars ().

Данный метод возвращает массив символов состоящий из части StringBuffer. Полное объявление данного метода:

void getChars (int *начало*, int *конец*, char [] *куда*, int *начало*)

Ниже представлен пример использования данного метода:

public class Main {

public static void main(String[] args) {

StringBuffer buffer=new StringBuffer("I can\'not buy the house of gold");

char [] chars=new char[10];

buffer.getChars(0, 10, chars, 0);

System.out.println(buffer);

System.out.println(Arrays.toString(chars));

}

}

Класс StringBuilder.

Данный класс де-факто ничем не отличается от StringBuilder. За исключением, что он является потоко-безопасным и строку StringBuilder можно одновременно использовать потоками. Также его чаще используют в больших проектах, а StringBuffer в небольших проектах.

Пример использования StringBuilder.

Все конструкторы класса StringBuilder совпадают с конструкторами StringBuffer. Такая же обстановка и стоит с метода. Ниже представлен пример использования StringBuilder:

public class Main {

public static void main(String[] args) {

StringBuilder builder = new StringBuilder();

//Метод append() присоединяет что-либо к концу StringBuilder

builder.append("append()");

//Метод insert() добавляет что-либо на указанный индекс.

builder.insert(7, "insert()");

System.out.println("Содержимое: "+builder);

//lastIndexOf () и indexOf ()

//находит первое и последнее вхождение указанного символа

System.out.println(" lastIndexOf(\"insert\") -> "+builder.lastIndexOf("insert"));

System.out.println(" indexOf(\"()\") -> "+builder.indexOf("()"));

//Метод replace () заменяет одни символы на другие.

builder.replace(6, 7, ", ");

//Методы delete () и deleteCharAt () удаляет часть или символ.

builder.delete(14, 17);

System.out.println("Содержимое: "+builder);

builder.append(" ");

builder.append(" setCharAt");

//Метод setCharAt () устанавливает символ по индексу

//Метод charAt () возвращает символ по индексу

builder.setCharAt(14, ',');

System.out.println(" charAt(6) -> "+builder.charAt(6));

System.out.println("Содержимое: "+builder);

//Метод reserve() переворачивает StringBuilder.

builder.reverse();

System.out.println("Содержимое: "+builder);

System.out.println("\n"+"Количество символов: "+builder.length());

System.out.println("");

}

}

Интерфейс CharSequence.

Данный интрефейс объявляет методы для чтения последовательностей символов. Его реализуют String, StringBuilder и StringBuffer. Ниже перечислены его методы:

char charAt (int *индекс*) – Извлекает символ по индексу.

default IntStream chars () – Возвращает поток типа IntStream.

default IntStream CodePoints () – Возвращает поток типа IntStream для кодовых точек.

int length () – Возвращает длину символьной строки.

CharSequence subSequence (int *начало*, int *конец*) – Возвращает часть строки.

String toString () – возвращает строковое представление символьной строки.

1

Графическое программирование.

События и их обработка.

В основу графического программирования легли события. События управляют любое графическое приложение, и генерируется от действий пользователя.

Модель делегирования событий.

Графические приложение используют *модель делегирования событий*. В нем есть *источник*, который посылает событие и *слушатель*, который генерирует и обрабатывает событие.

Слушатель должен быть зарегистрирован у источника. Получается пользовательский интерфейс и логика обработки две разные сущности.

События.

*Событие* – это объект, который описывает изменения источника. Генерируется после действий пользователя с пользовательским интерфейсом.

Источник событий.

*Источник* – это объект, который генерирует событие. Для того чтоб получить уведомление о событие, слушатель зарегистрироваться у источника.

Для регистрации используется методы-регистрации событий. С формой объявления:

public void add*Тип*Listener(*Тип*Listener *элемент*)

где Тип – имя события, а *элементы –* ссылка на объект слушателя. При наступлении события оповещаются и получают копию событии.

Также предоставляется метод, который удаляет убрать объект из слушателя. Объявление:

public void remove*Тип*Listener(*Тип*Listener *элемент*)

Данные методы предоставляются источником.

Слушатель.

*Слушатель* – объект, который получает известия о событиях. К слушателю предъявляет два требования:

1) Он должен быть зарегрестирован у источника.

2) Реализация методов принимающих и обрабатывающих события.

Все методы, получающие и обрабатывающие события, определяются в пакете java.awt.event .

Классы событий.

Классы, представляющие события легли в основу обработки событий. На вершине иерархии находится класс EventObject, который является суперклассом для всех событий и находящийся в java.util . Класс AWTEvent подкласс EventObject, который косвенно суперкласс для событий в пакете awt, находится в java.awt.event . Именно события awt используется в модели делегирования.

Список событий:

ActionEvent - Формируется после щелчка на кнопке, двойного щелчка на элементе списка или выбора пункта меню.

AdjustmentEvent - Формируется при манипулировании полосой прокрутки.

ComponentEven - Формируется, когда компонент становится видимым или невидимым, а также при изменении его размеров и перемещении.

ContainerEvent - Формируется, когда компонент добавляется в контейнер или удаляется из него.

FocusEvent- Формируется, когда компонент получает или теряет фокус ввода.

InputEvent-Абстрактный суперкласс для всех классов событий, связанных с вводом данных .

ItemEvent - Формируется по щелчку на флажке или элементе списка, а также наступает при выборе или отмене выбора пункта меню.

KeyEvent - Формируется при вводе данных с клавиатуры

MouseEvent - Формируется при перемещении или перетаскивании мыши, а также по нажатию, отпусканию или щелчку кнопкой мыши или же в том случае, когда курсор мыши наводится на элемент интерфейса или перемещается с него

TextEvent - Формируется при изменении содержимого области или поля ввода текста

WindowEvent - Формируется, если окно делается активным или неактивным, сворачивается или разворачивается, открывается или закрывается либо покидается

Интерфейсы слушателей событий.

Слушатель получает уведомление о событии. Слушатели событий awt создаются с помощью реализации определенных интерфейсов. Источник при необходимости может вызвать метод, определенным слушателем и событие передать, как аргумент.

Список основных интерфейсов:

ActionListener - Определяет один метод для получения событий действий. Такого рода события наступают при нажатии кнопки, выборе пункта меню и т.д.

AdjustmentListener - Определяет один метод для получения событий настройки, подобных тем, которые наступают при манипулировании полосой прокрутки.

ComponentListener - Определяет четыре метода для выявления факта сокрытия, показа, изменения размеров или перемещения компонента пользовательского интерфейса

ContainerListener - Определяет два метода для выявления факта добавления компонента в контейнер или удаления из него.

FocusListener - Определяет два метода для выявления факта получения и потери компонентом фокуса ввода.

ItemListener - Определяет один метод для выявления факта изменяющий состояния элемента пользовательского интерфейса. Событие от элемента может формироваться, например, флажком.

KeyListener - Определяет три метода для выявления нажатия, отпускания клавиши ввода с клавиатуры

MouseListener - Определяет пять методов для выявления щелчка кнопкой мыши, ее нажатия и отпускания, а также наведения курсора мыши на элемент пользовательского интерфейса или перемещения с него

MouseMotionListener -Определяет два метода для выявления факта перемещения или перетаскивания мыши

MouseWheelListener - Определяет один метод для выявления факта прокрутки колесика мыши

TextListener - Определяет один метод для выявления факта изменения текстового значения

WindowListener - Определяет семь методов для выявления факта открытия и закрытия, сворачивания и разворачивания, активизации и деактивизации или покидания окна.

Программирование на модели делегирования событий.

По сути, программирование кода обработки событий сводится к 2 этапам:

* Реализация соответствующего интерфейса у слушателя, для получения событий.
* Реализация кода для регистрации и если есть необходимость отмены регистрации слушателей (действия у источника).

Все остальное дело графического интерфейса, который косвенно и является источником событий. Суть графическое программирование можно представить, как панель с кнопочками, которая при нажатии реагирует и делает определенное действие. Но и действие заложено инженером, а в деле программирование источник сам выберет необходимый метод и к соответственному событию реагирует по действиям программиста.

Библиотека JavaFX.

Библиотека javaFX – библиотека, предназначенная для графического программирования. Была создана относительно не давно, но завоевала популярность за удобность, гибкость и не зависимость от платформ.

Все пакеты, связанные с данной библиотекой находятся с префиксом javafx.

Классы Stage и Scene.

В основу пользовательских интерфейсов на основе javafx легли понятия *театральные подмостики* (stage) и *сцены* (scene).

* Stage – это место, где проходит выступление и является контейнером верхнего уровня.
* Scene – это само выступление.

Все графические приложения получают доступ к одному контейнеру Stage, называемым *основным платформой*. Эта платформа предоставляется исполняющей платформой. Но все равно есть возможность завести 2 и более stage .

Узлы и графы сцены.

Отдельные элементы сцены называются *узлами*(nodes). Узлы могут образовывать *группы узлов* и в них состоять.

Узлы могут быть *родительскими*(parent node) , когда имеют дочерние узлы; *листья*(leaves), когда не имеют дочерних узлов.

Совокупность всех узлов сцены называются графами сцены(scene graph), которые образуют дерево(tree).

Важную роль у графа играет *корень*(root). Корень – это идентичный узел, который не имеет родительского узла, получается все остальные узлы косвенно потомки корневого узла.

Панели компоновки.

В JavaFX также существуют панели компоновки, которые представляют возможность управления процесса размещения элементов. Классы компоновки находятся в javafx.scene.layout .Существует несколько панелей.

1)FlowPane – класс обеспечивает плавающую компоновку.

2)GridPane – класс обеспечивает табличную компоновку.

3) BorderPane – разбивает пользовательский интерфейс на пять частей: южную, западную сторону и п.р.

Жизненный цикл JavaFX приложения и класс Application

Любая JavaFX-приложении расширяет класс Application, в нем определено 3 метода, которые управляют жизненным циклом.

* void init () – инициализация необходимых вещей. Нельзя использовать для создания платформы и формирования сцены. Необязателен в использовании.
* abstract void start () – начинает выполнение программы, котором конструируют stage и scene, обязателен в применении.
* void stop () – прекращает работу приложения. Выполняет рутинные операции, со сборкой мусора и т.п.

Запуск приложения JavaFX.

Чтоб запустить автономное приложение с JavaFX , нужно использовать метод launch(), определенный в интерфейсе Application.

public static void launch (String … аргументы)

*где* аргументы – список строк, обычно являющимся аргументами командной строки. Вызов метода launch () , приводит загрузке приложения, со следующим методомstart() . Данная версия метода загружает в приложение класс, который расширяет класс Application.

Каркас приложения JavaFX.

При создании приложений следует использовать *каркас* *приложений на JavaFX*. В ниже программе выводится сообщения о вызове методов.

Код:

package com.company;

import javafx.application.Application;

import javafx.scene.Scene;

import javafx.scene.layout.FlowPane;

import javafx.stage.Stage;

public class Main extends Application {

@Override

public void init() {

System.out.println("В методы init().");

}

@Override

public void start(Stage myStage) {

System.out.println("В методе start().");

//устанавливаем заголовок окна

myStage.setTitle("JavaFX приложение");

FlowPane root = new FlowPane();

Scene myscene = new Scene(root, 400, 350);

myStage.setScene(scene);

myStage.show();

}

@Override

public void stop() {

System.out.println ("В методе stop()");

}

public static void main(String[] args) {

launch(args);

}

}

Вначале программы мы импортируем stage и scene. При помощи них мы будем строить приложение. Позже мы импортируем класс Application. Используя класс FlowPane можно сделать корень.

В классе Main переопределяем метод start(). Аргумент ему передаем театральный подмостик. В строке:

myStage.setTitle("JavaFX приложение");

К окну мы присваиваем заголовок "JavaFX приложение" при помощи метода setTitle().

Следующим шагом создается корневой узел. В данном случае корень создается с помощью объекта FlowPane. Данный класс поддерживает плавающую компоновку.

В создание сцены мы используем:

Scene myscene = new Scene (root, 400, 350);

В параметры мы передаем корень и размеры. Позже мы устанавливаем сцену на платформу:

myStage.setScene(myscene);

Устанавливаем с помощью метода setScene().

Последняя строка отображает платформу:

myStage.show();

В методе main() мы вызываем метод launch() , а в аргумент передаем списочный массив args:

launch(args);

Компоненты и события.

Компонент Label.

Компонент Label - это ярлык или метка. Данный компонент создается с помощью класса Label. Конструктору мы передаем строку, которая будет служить меткой.

Для того чтоб внести компонент в корень мы используем:

root.getChildren().add(label);

Пример программы:

package com.company;

import javafx.application.Application;

import javafx.scene.Scene;

import javafx.scene.layout.FlowPane;

import javafx.stage.Stage;

import javafx.scene.control.Label;

public class Main extends Application {

@Override

public void init() {

System.out.println("В методы init().");

}

@Override

public void start(Stage myStage) {

System.out.println("В методе start().");

//устанавливаем заголовок окна

myStage.setTitle("JavaFX приложение");

FlowPane root = new FlowPane();

Scene scene = new Scene(root, 400, 350);

myStage.setScene(scene);

Label label = new Label("Первый элемент");

root.getChildren().add(label);

myStage.show();

}

@Override

public void stop() {

System.out.println("В методе stop()");

}

public static void main(String[] args) {

launch(args);

}

}

Обработка событий в JavaFX

События в JavaFX наследуются от класса Event, который в свою очередь подкласс EventObject, что говорит, что события ни чем не различаются.

Для обработки событий требуется реализовать интерфейс EventHandler с объявлением:

interface EventHandler<T extends Event>

где *T* – тип события. EventHandler объявляет метод handler () с объявление:

void handler (T *событие*)

где *событие* - генерируемое событие.

Элемент Button

Элемент Button – это обычная кнопка. Находится в javafx.scene.control.В конструктор ему передают текст на кнопке, изображение и т.п.

Для обработки события используют метод setOnAction ().

Пример:

package com.company;

import javafx.application.Application;

import javafx.geometry.Pos;

import javafx.scene.Scene;

import javafx.scene.control.Button;

import javafx.scene.layout.FlowPane;

import javafx.stage.Stage;

import javafx.scene.control.Label;

public class Main extends Application {

Label label;

@Override

public void start(Stage myStage) {

System.out.println("В методе start().");

myStage.setTitle("JavaFX приложение");

FlowPane root = new FlowPane(10, 10);

Scene scene = new Scene(root, 300, 100);

myStage.setScene(scene);

root.setAlignment(Pos.CENTER);

label = new Label("Результат!");

Button button = new Button("Действие, кнопка.");

button.setOnAction( (ae) ->

label.setText("Вы нажали на кнопку"));

root.getChildren().addAll(label, button);

myStage.show();

}

@Override

public void stop() {

System.out.println("В методе stop()");

}

public static void main(String[] args) {

launch(args);

}

}

Элемент CheckBox

Элемент CheckBox –это флажок или галочка, он схож с кнопкой, но еще владеет галочкой или флажком. У этого элемента есть 3 состояния:

* Поднят
* Снят
* Неопределенен

Неопределенное состояние показывается, когда значение не установлено или не имеет значение в конкретной ситуации.

Для создания галочки используется конструктор, которому передается строка с пояснительной надписью. Флажок также генерирует событие ActionEvent.

Нижняя программа спрашивает вас о вашем животном. В ней Создается три флажка с надписям «Кошка», «Собака», «Птица» , а после выводит кого вы выбрали только что и скольких вы выбрали.

Программа:

package com.company;

import javafx.application.Application;

import javafx.geometry.Orientation;

import javafx.geometry.Pos;

import javafx.scene.Scene;

import javafx.scene.control.CheckBox;

import javafx.scene.layout.FlowPane;

import javafx.stage.Stage;

import javafx.scene.control.Label;

import javafx.event.\*;

public class Main extends Application {

Label respons;

Label selected;

CheckBox cat;

CheckBox dog;

CheckBox birt;

String pet;

@Override

public void start(Stage myStage) {

myStage.setTitle("Твой питомец?");

FlowPane root = new FlowPane(Orientation.VERTICAL, 10, 10);

Scene scene = new Scene(root, 400, 200);

myStage.setScene(scene);

root.setAlignment(Pos.CENTER);

respons = new Label("");

selected = new Label("");

cat = new CheckBox("Кошка");

dog = new CheckBox("Собака");

birt = new CheckBox("Птица");

cat.setOnAction(new EventHandler<ActionEvent>() {

@Override

public void handle(ActionEvent event) {

if(cat.isSelected())

respons.setText("Вы владеете кошкой");

else

respons.setText("Вы не владеет кошкой");

showAll();

}

});

dog.setOnAction(new EventHandler<ActionEvent>() {

@Override

public void handle(ActionEvent event) {

if(dog.isSelected())

respons.setText("Вы владеете собакой");

else

respons.setText("Вы не владеете собакой");

showAll();

}

});

birt.setOnAction(new EventHandler<ActionEvent>() {

@Override

public void handle(ActionEvent event) {

if(birt.isSelected())

respons.setText("Вы владеете птицой");

else

respons.setText("Вы не владеете птицой");

showAll();

}

});

root.getChildren().addAll(cat, dog, birt, respons, selected);

myStage.show();

}

void showAll(){

pet="";

if(cat.isSelected()) pet +="Кошка ";

if(dog.isSelected()) pet += "Собака ";

if(birt.isSelected()) pet +="Птица ";

selected.setText("У вас есть: "+pet);

}

@Override

public void stop() {

System.out.println("Программа прекратилась");

}

public static void main(String[] args) {

launch(args);

}

}

Стоит отметить то, что у корня установлена вертикальная ориентация. Ориентация по умолчанию горизонтальная.

В случае необходимости использования неопределенного состояния необходимо его определить явно с помощью метода setAllowIndeterminate() и передаем true, если необходимо неопределенное состояние. Для проверки на неопределенно

cat.setAllowIndeterminate(true);

cat.setOnAction(new EventHandler<ActionEvent>() {

@Override

public void handle(ActionEvent event) {

if(cat.isSelected())

respons.setText("Вы владеете кошкой");

else if(cat.isIndeterminate())

respons.setText("Вы неопределенны на счет кошки");

else

respons.setText("Вы не владеет кошкой");

showAll();

}

});

Компонент ListViev

Компонент ListViev – отображает список элементов с возможностью выбора одного и более. Инкапсулируется классом ListViev. Конструктор ListView:

ListView (Observable <T> список).

где *список* там список элементов, подлежащих к отображению. Класс Observable поддерживает список объектов, которые образуются с помощью статического метода observableList(). Сам метод действует принципом фабрики и находится в javafx.collections . Полное объявление:

static <E> ObservableList<E> observableArrayList(E … элементы).

В данном случае E обозначает тип элементы. У элементов ListView можно настроить размеры с помощью следующих методов:

* setPrefWidth()
* setPrefHeigth()
* setPrefSize()

Первый и второй задают высоту и ширину, а третий сразу ширину и высоту.

Слушатель событий тут реализуется интерфейсом ChangeListener. В ChangeListener определен метод changed():

void changed(ObservableLValue <? extends T> *changed*, T oldVal, T newVal)

где *changed* – экземпляр класса ObservableLValue <T> за изменением, которого состояния можно следить. В oldVal передается прежнее значение, а newVal новое.

Чтоб прослушивать события, прежде необходимо получить модель выбора, используемую компонентом ListView. Это можно сделать с помощью метода getSelectModel():

final MultipleSelectionModel <T> getSelectModel()

Данный метод возвращает ссылку на модель. Класс MultipleSelectionModel определяет модель, используемую для группового выбора. Однако в ListView групповой режим разрешен только в условии, что режим группового выбора активизирован.

Используя эту модель можно получить ссылку на свойства выбранного элемента. Элемент в нашем случае определяет действия элемента. Для этого используется метод selectItemProperty()

Добавить слушателя в свойство осуществляется методом addListener ():

void addListener (ChangeListener<? super T> слушатель)

В данном случае T обозначает тип свойства.

Код:

package com.company;

import javafx.application.Application;

import javafx.beans.value.ChangeListener;

import javafx.beans.value.ObservableValue;

import javafx.collections.FXCollections;

import javafx.collections.ObservableList;

import javafx.geometry.Orientation;

import javafx.geometry.Pos;

import javafx.scene.Scene;

import javafx.scene.control.ListView;

import javafx.scene.control.MultipleSelectionModel;

import javafx.scene.layout.FlowPane;

import javafx.stage.Stage;

import javafx.scene.control.Label;

public class Main extends Application {

@Override

public void start(Stage myStage) {

myStage.setTitle("Выберите свое животное.");

FlowPane root = new FlowPane(Orientation.VERTICAL, 10, 10);

Scene scene = new Scene(root, 400, 200);

myStage.setScene(scene);

root.setAlignment(Pos.CENTER);

Label label = new Label("");

ObservableList<String> myObsLs = FXCollections.observableArrayList(

"Кошка", "Собака", "Попугай", "Хорек");

ListView<String> listView = new ListView<>(myObsLs);

listView.setPrefSize(100, 70);

MultipleSelectionModel<String> msm =listView.getSelectionModel();

msm.selectedItemProperty().addListener(

new ChangeListener<String>() {

@Override

public void changed(ObservableValue<? extends String> observable, String oldValue, String newValue) {

if(newValue.equals("Кошка"))

label.setText("Вы как и я выбрали кошку");

else

label.setText("Вы выбрали: "+newValue);

}

});

root.getChildren().addAll(listView, label);

myStage.show();

}

@Override

public void stop() {

System.out.println("Программа прекратилась");

}

public static void main(String[] args) {

launch(args);

}

}

Для того чтоб поставить режим выбора нескольких элементов, следует вызвать метод setSelectionMode(режим). Где режим нужно указать SelectionMode.MULTIPLE или SelectionMode.SINGLE.

Для получения списка следует использовать метод getSelectedItems().

Компонент ComboBox

Компонент ComboBox – создает список, с выбранным ранее до него значением, но с возможность выбора значения. Конструктор данного класса:

ComboBox (ObservableList<E> *коллекция*)

В данном случае создается список с элементами из *коллекции*. Чтобы получить элемент, который выбран пользователь, следует использовать метод getValue() с полным объявлением:

final T getValue()

Чтоб поставить значение, которое будет доступно по умолчанию, следует вызвать метод setValue() с провиденным ниже объявлением:

final void setValue(T *новое\_значение*)

Также есть возможность поставить режим пользовательского редакции списка и чтоб его поставить следует вызвать метод setEditable() с приведенным ниже объявлением:

final void setEditable(boolean *режим*)

Пример использования ComboBox:

package myPackage;

import javafx.application.Application;

import javafx.collections.FXCollections;

import javafx.collections.ObservableList;

import javafx.geometry.Pos;

import javafx.scene.Scene;

import javafx.scene.control.\*;

import javafx.scene.layout.FlowPane;

import javafx.stage.Stage;

public class Main extends Application{

@Override

public void start(Stage primaryStage) {

primaryStage.setTitle("ComboBox Demo");

FlowPane root = new FlowPane(10, 10);

root.setAlignment(Pos.CENTER);

Scene scene = new Scene(root, 310, 120);

primaryStage.setScene(scene);

Label response=new Label("");

ObservableList<String> list= FXCollections.observableArrayList( "Badger", "Panda", "Birds", "Bear");

ComboBox<String> box=new ComboBox<>(list);

box.setValue("Badger");

box.setEditable(true);

response.setText("You are press on "+box.getValue());

box.setOnAction((ae) -> {

response.setText("You are press on "+box.getValue());

});

root.getChildren().addAll(box, response);

primaryStage.show();

}

@Override

public void stop() {

System.out.println("The program is over");

}

public static void main(String[] args) {

launch(args);

}

}

TreeView.

Данный класс создает список в *древовидной форме*. К примеру можно отнести файловый редактор содержит список, где файлы находятся в директории.

Чтоб создать такой список следует создать объект класса TreeView, а после добавить корень TreeItem, наполненная с объектами TreeItem. Чтобы отреагировать на событие используется такой же способ как в ListView.

Класс TreeView обобщен. Обычно он принимает String. Ниже представлен конструктор:

TreeView<> (TreeItem <T> *корень*)

Класс TreeItem создает в своем роде листья дерева. Листья могут быть оконченными или с корнями. Следует отметить, что он расширяет Node.

Ниже представлен пример использования данного класс:

package myPackage;

import javafx.application.Application;

import javafx.beans.value.ChangeListener;

import javafx.beans.value.ObservableValue;

import javafx.geometry.Pos;

import javafx.scene.Scene;

import javafx.scene.control.Label;

import javafx.scene.control.MultipleSelectionModel;

import javafx.scene.control.TreeItem;

import javafx.scene.control.TreeView;

import javafx.scene.layout.FlowPane;

import javafx.stage.Stage;

public class Main extends Application {

Label result;

@Override

public void start(Stage primaryStage) {

primaryStage.setTitle("Демонстрация TreeView");

FlowPane root=new FlowPane(20, 20);

root.setAlignment(Pos.CENTER);

Scene scene=new Scene(root, 290, 500);

primaryStage.setScene(scene);

TreeItem<String> music=new TreeItem<>("Музыка");

TreeItem<String> pilots21=new TreeItem<>("21 Pilots");

pilots21.getChildren().addAll(

new TreeItem<>("Not today"),

new TreeItem<>("Hometown"),

new TreeItem<>("We Don't Believe What's On TV"),

new TreeItem<>("House Of Gold"));

TreeItem<String> linkin\_park=new TreeItem<>("Linkin Park");

linkin\_park.getChildren().addAll(

new TreeItem<>("From The Inside"),

new TreeItem<>("In The End"),

new TreeItem<>("Shadow Of The Day"));

TreeItem<String> OneRepublic=new TreeItem<>("OneRepublic");

OneRepublic.getChildren().addAll(

new TreeItem<>("All The Right Moves"),

new TreeItem<>("Secrets"));

music.getChildren().addAll(pilots21, linkin\_park, OneRepublic);

TreeView<String> treeMusic=new TreeView<>(music);

result=new Label("");

MultipleSelectionModel<TreeItem<String>> model=treeMusic.getSelectionModel();

model.selectedItemProperty().addListener(new ChangeListener<TreeItem<String>>() {

@Override

public void changed(ObservableValue<? extends TreeItem<String>> observable,

TreeItem<String> oldValue, TreeItem<String> newValue) {

System.out.println(newValue.getValue());

String str=newValue.getValue();

TreeItem<String> itemGroup=newValue.getParent();

result.setText("Вы слушаете "+itemGroup.getValue()+" песню "+str);

}

});

root.getChildren().addAll(treeMusic, result);

primaryStage.show();

}

public static void main(String[] args) {

launch(args);

}

}

TextField

Компонент TextField – это строка ввода.

Для того чтоб поставить промт текст у TextField следует вызвать метод setPromptText() и в аргумент передать строку, которая после станет промт текстом.

Для того чтоб поставить размер следует вызвать метод setPrefColumnCount() и в аргумент передать размер.

Пример:

package com.company;

import javafx.application.Application;

import javafx.geometry.Orientation;

import javafx.geometry.Pos;

import javafx.scene.Scene;

import javafx.scene.control.Button;

import javafx.scene.control.Label;

import javafx.scene.control.TextField;

import javafx.scene.layout.FlowPane;

import javafx.stage.Stage;

public class Main extends Application {

TextField textField;

@Override

public void start(Stage myStage) {

myStage.setTitle("операция над словами.");

FlowPane root = new FlowPane(Orientation.VERTICAL, 10, 10);

Scene scene = new Scene(root, 220, 210);

myStage.setScene(scene);

root.setAlignment(Pos.CENTER);

textField= new TextField();

textField.setPromptText("Введите строку для операции");

textField.setPrefColumnCount(15);

Label label = new Label();

textField.setOnAction((ae) ->

label.setText(textField.getText())

);

Button clearB = new Button("Очистить");

clearB.setOnAction((ae)->

textField.clear()

);

Button copyB = new Button("Копировать");

copyB.setFocusTraversable(false);

copyB.setOnAction((ae) ->

textField.copy()

);

Button pasteB = new Button("Вставить");

pasteB.setFocusTraversable(false);

pasteB.setOnAction((ae) ->

textField.paste()

);

Button cutB = new Button("Вырезать");

cutB.setFocusTraversable(false);

cutB.setOnAction((ae) ->

textField.cut()

);

root.getChildren().addAll(textField, clearB, cutB, copyB, pasteB);

myStage.show();

}

@Override

public void stop(){

System.out.println("Программа прекратилась.");

}

public static void main(String[] args) {

launch(args);

}

}

Строит обратить внимание тут при помощи метода setFocusTraversable() происходит фокусирование клавиатуры на строку и кнопку.

Компонент ToggleButton.

Компонент ToggleButton –это обычный *кнопка переключатель*. Он может находиться в двух состояниях: *поднятом* и *опущенном*. Конструктор данного класса:

ToggleButton (String *текст*)

В таком случае создается кнопка переключатель с *текстом*. Также есть возможность ввести изображение в конструкторе.

Чтоб определить состояние кнопки переключателя следует вызвать метод isSelected(), с приведенным ниже полным объявлением:

final boolean isSelected()

Он возвращает логическое значение true, если кнопка переключатель находится в состоянии поднятой или наоборот.

Пример компонента ToggleButton:

package myPackage;

import javafx.application.Application;

import javafx.geometry.Pos;

import javafx.scene.Scene;

import javafx.scene.control.\*;

import javafx.scene.layout.FlowPane;

import javafx.stage.Stage;

public class Main extends Application{

ToggleButton togBut;

@Override

public void start(Stage primaryStage) {

primaryStage.setTitle("ToggleButton example");

FlowPane root=new FlowPane(10,10);

root.setAlignment(Pos.CENTER);

Scene scene=new Scene(root, 240, 140);

primaryStage.setScene(scene);

Label label=new Label("");

togBut=new ToggleButton("Yes/No");

togBut.setOnAction((ae) -> {

if(togBut.isSelected())

label.setText("Button is on");

else

label.setText("Button is off");

});

root.getChildren().addAll(togBut, label);

primaryStage.show();

}

@Override

public void stop() {

System.out.println("The program is over");

}

public static void main(String[] args) {

launch(args);

}

}

Компонент RadioButton.

Компонент RadioButton - это группа взаимоисключающих кнопок переключателей. Конструктор класса RadioButton:

RadioButton (String *строка*)

В данном случае создается кнопка с текстом *строка*. Чтоб создать группу таких кнопок следует создать объект типа ToggleGroup. И с помощью метода setToggleGroup() можно добавить кнопку в объект ToggleGroup. Полное объявление метода:

final void setToggleGroup(ToggleGroup *группа*)

Вызвав метод setSelected() можно поставить состояние кнопке в группе, а методом fire() поставить кнопку по умолчанию. Полные объявления методов:

final void setSelected(boolean *режим*)

void fire()

Где *режим* можно поставить true и кнопка будет выбрана и наоборот. Различие между setSelected() и fire() в том, что fire() ставится в самом начале (по умолчанию), а setSelected() можно ставить режим косвенно.

Пример использования данного компонента:

package myPackage;

import javafx.application.Application;

import javafx.application.Platform;

import javafx.geometry.Pos;

import javafx.scene.Scene;

import javafx.scene.control.\*;

import javafx.scene.layout.FlowPane;

import javafx.stage.Stage;

public class Main extends Application{

ToggleButton togBut;

@Override

public void start(Stage primaryStage) {

primaryStage.setTitle("What transport you\'re using?");

FlowPane root = new FlowPane(10, 10);

root.setAlignment(Pos.CENTER);

Scene scene = new Scene(root, 310, 120);

primaryStage.setScene(scene);

Label label = new Label("");

RadioButton buttonAir = new RadioButton("Airplane.");

RadioButton buttonTrain = new RadioButton("Train");

RadioButton buttonCar = new RadioButton("Car");

ToggleGroup group = new ToggleGroup();

buttonAir.setToggleGroup(group);

buttonCar.setToggleGroup(group);

buttonTrain.setToggleGroup(group);

buttonAir.setOnAction((ae) ->

label.setText("You are using the Airplane")

);

buttonCar.setOnAction((ae) ->

label.setText("You are using the Car")

);

buttonTrain.setOnAction((ae) ->

label.setText("You are using the Train")

);

buttonCar.fire();

Button exit=new Button("Exit");

togBut=new ToggleButton("Yes/No");

togBut.setOnAction((ae) -> {

if(togBut.isSelected()) {

buttonAir.setDisable(true);

buttonCar.setDisable(true);

label.setDisable(true);

buttonTrain.setDisable(true);

} else {

buttonAir.setDisable(false);

buttonCar.setDisable(false);

label.setDisable(false);

buttonTrain.setDisable(false);

}

});

111 exit.setOnAction((ae) ->

Platform.exit()

);

root.getChildren().addAll(buttonAir, buttonCar, buttonTrain, togBut, label, exit);

primaryStage.show();

}

@Override

public void stop() {

System.out.println("The program is over");

}

public static void main(String[] args) {

launch(args);

}

}

Элемент Text

Компонент Text – это обычный текст, но его отличие от Label в том, что он более функционален. У него можно поставить тип шрифта, вес шрифта и т.п.

Для того чтоб поставить настройки у шрифта следует вызвать метод setFont(). И передать ему объект типа Font и передать методу font() необходимые значения.

К ним относят:

* family – тип шрифта.
* FontWeight - вес шрифта, передаем одно из девяти значений.
* Постановка – поза шрифта, может быть FontPosture.REGULAR(обычный шрифт) или FontPosture.ITALIC(*волнистый шрифт*)
* size – размер

Насчет порядка, следует придерживаться следующего порядка:

text.setFont (Font.font (String *тип\_шрифта*, FontWeight.*вес\_шрифта*, FontPosture.позиция, double *размер*))

Чтоб установить цвет setFill().

Пример:

package com.company;

import javafx.application.Application;

import javafx.geometry.Pos;

import javafx.scene.Scene;

import javafx.scene.layout.FlowPane;

import javafx.scene.paint.Color;

import javafx.scene.text.Font;

import javafx.scene.text.FontPosture;

import javafx.scene.text.FontWeight;

import javafx.scene.text.Text;

import javafx.stage.Stage;

public class Main extends Application {

@Override

public void start(Stage myStage) {

myStage.setTitle("Использование Text");

FlowPane root = new FlowPane(20, 20);

root.setAlignment(Pos.CENTER);

Scene scene = new Scene(root, 350, 100);

myStage.setScene(scene);

Text textf = new Text("Изображение");

textf.setFill(Color.GOLD);

textf.setFont(Font.font("Calibri", FontWeight.MEDIUM, FontPosture.REGULAR, 25));

root.getChildren().addAll(textf);

myStage.show();

}

@Override

public void stop(){

System.out.println("Программа прекратилась.");

}

public static void main(String[] args) {

launch(args);

}

}

Класс Image и ImageView

Для того чтоб задать картинку и провести над ней операции используется классы Image, ImageView. Первый инкапсулирует саму картинку, а второй загружает картинку. У класса Image определено несколько конструкторов:

Image(String *url*)

Image (new FileInputStream (String *ссылка*))

Первый загружает картинку с url, указанный в конструкторе. Во второй поток загружает картинку с *ссылкой*.

После создания объекта инкапсулирующего картинку, его следует передать его в конструктор класса ImageView:

ImageView (Image *объект*)

Все эти два класса расширяют класс Node так, что они могут быть добавлены в граф сцены. Ниже представлен пример использования Image, ImageView:

package myPackage;

import javafx.application.Application;

import javafx.scene.Scene;

import javafx.scene.image.Image;

import javafx.scene.image.ImageView;

import javafx.scene.layout.FlowPane;

import javafx.stage.Stage;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

public class Main extends Application {

Image image;

ImageView view;

@Override

public void start(Stage primaryStage) throws FileNotFoundException {

primaryStage.setTitle("Демонстрация картинок");

FlowPane root=new FlowPane(10, 10);

Scene scene=new Scene(root, 500, 450);

primaryStage.setScene(scene);

image=new Image(new FileInputStream("C:\\Users\\Семья\\Desktop\\маме\\9RmZmFWmWiI.jpg"));

view=new ImageView(image);

root.getChildren().addAll(view);

primaryStage.show();

}

public static void main(String[] args) {

launch(args);

}

}

Эффекты

Эффекты позволяют создать более точное и изысканное представление графического интерфейса. Существует ряд встроенных эффектов. К ним относят:

1. Bloom – увеличивает яркость.
2. BoxBlur – размывает отображении узла.
3. DropShadow – отображает тени от узлов.
4. Glow – создает эффект свечения.
5. InnerShadow – отображает тени внутри узла
6. Lighting – создает эффект теней при наличии света.
7. Reflection – создает эффект отображения.

Чтобы использовать эффекты следует на узел повесить эффект. Это можно сделать с помощью метода setEffect(*эффект*), но перед этим следует создать объект эффекта.

Эффект BoxBlur.

Данный эффект размывает узел. Графический пример:



Эта технология основывается на настройке пикселей и поэтому степень размытия можно контролировать. Условие состоит в том, что размывать можно только прямоугольники. Конструктор:

BoxBlur (double *ширина*, double *высота*, int *итерация*)

где *ширина*, *высота* – определяется размеры области размытия, могут принимать в аргументы 0-255; где *итерация* - уровень кратности применения эффекта в пределах 0-3. В конструкторе по умолчанию *ширина* и *высота* 5.0, а где *итерация* 1.

Также существует ряд методов, которые могут поменять высоту, ширину или итерацию:

* final void setWidth(double *ширина*)
* final void Height(double *высота*)
* final void setIterations(int *итерация*)

Пример:

package com.company;

import javafx.application.Application;

import javafx.geometry.Pos;

import javafx.scene.Scene;

import javafx.scene.control.Label;

import javafx.scene.effect.BoxBlur;

import javafx.scene.control.Button;

import javafx.scene.layout.FlowPane;

import javafx.stage.Stage;

public class Main extends Application {

@Override

public void start(Stage myStage) {

myStage.setTitle("операция над словами.");

FlowPane root = new FlowPane( 20, 20);

Scene scene = new Scene(root, 200, 70);

myStage.setScene(scene);

root.setAlignment(Pos.CENTER);

Label label = new Label("");

BoxBlur boxBlur = new BoxBlur(2.9, 0.5, 3);

Button button = new Button("Размытая кнопка");

button.setEffect(boxBlur);

button.setOnAction((ae) ->

label.setText("Размытие")

);

root.getChildren().addAll(button, label);

myStage.show();

}

@Override

public void stop(){

System.out.println("Программа прекратилась.");

}

public static void main(String[] args) {

launch(args);

}

Эффект Reflection

Эффект Reflection отображает узел по горизонтали. Графический пример:



Для создания отображения применяется следующий конструктор:

Reflection (double *смещение*, double *доля*, double непрозрачность\_вверх, double *непрозрачность\_низ*).

где смещение – определить расстояние между нижней и верхней части;

где *доля* - предусмотрена, когда нужно отразить определенную часть в параметр передается число от 0 до 1.0;

где *непрозрачность\_вверх* устанавливает непрозрачность оригинала, а *непрозрачность\_низ*, установляются от 0 до 1.0;

Получается, чтоб оригинал был непрозрачным то нужно установить 1.0, а отражение было прозрачно нужно от 0. Ели будет 0, то низ будет виден, хотя логически это неправильность.

В конструкторе по умолчанию: *смещение*  0, доля 0.75, *непрозрачность\_вверх* 0.5, *непрозрачность\_низ* 0.

*void setFraction(double доля)*

void setTopOffset (double смещение)

Также существует ряд методов, которые позволяют установить определенное значение:

* final void setTopOpacity(double *непрозрачность*)
* final void setBottomOpacity(double *прозрачность*)
* void setTopOffset (double *смещение*)
* void setFraction(double *доля*)

Пример:

package com.company;

import javafx.application.Application;

import javafx.geometry.Pos;

import javafx.scene.Scene;

import javafx.scene.effect.Reflection;

import javafx.scene.layout.FlowPane;

import javafx.scene.text.Text;

import javafx.stage.Stage;

public class Main extends Application {

@Override

public void start(Stage myStage) {

myStage.setTitle("Использование эффекта Reflection");

FlowPane root = new FlowPane(20, 20);

Scene scene = new Scene(root, 350, 100);

myStage.setScene(scene);

root.setAlignment(Pos.CENTER);

Reflection reflection = new Reflection(1, 0.75, 1, 0);

Text textf = new Text(100, 50, "Это отображение");

textf.setEffect(reflection);

root.getChildren().addAll(textf);

myStage.show();

}

@Override

public void stop(){

System.out.println("Программа прекратилась.");

}

public static void main(String[] args) {

launch(args);

}

}

Эффект DropShadow

Эффект DropShadow – создает эффект тени от узла. Графический пример:



У данного класса три конструктора.

* DropShadow()
* DropShadow(double *радиус*, Color.*цвет*)
* DropShadow(double *радиус*, double *смещение\_x*, double *смещение\_y*, Color.*цвет*)

где *радиус* – радиус смещения.

где *цвет* – цвет тени.

где *смещение\_x –* смещение относительно ширины.

где *смещение\_y* – смещение относительно высоты.

Также у эффекта можно определить алгоритм размытия и чтобы его настроить нужно использовать метод setBlurType() и передать ему необходимый тип. Также существует ряд методов, которые могут назначить отдельное значение.

Пример такого эффекта:

package com.company;

import javafx.application.Application;

import javafx.geometry.Pos;

import javafx.scene.Scene;

import javafx.scene.layout.FlowPane;

import javafx.scene.paint.Color;

import javafx.scene.text.Font;

import javafx.scene.text.FontPosture;

import javafx.scene.text.FontWeight;

import javafx.scene.text.Text;

import javafx.stage.Stage;

import javafx.scene.effect.DropShadow;

import javafx.scene.effect.BlurType;

public class Main extends Application {

@Override

public void start(Stage myStage) {

myStage.setTitle("Использование Text");

FlowPane root = new FlowPane(20, 20);

root.setAlignment(Pos.CENTER);

Scene scene = new Scene(root, 360, 100);

myStage.setScene(scene);

DropShadow dropShadow = new DropShadow( 5,4, 4, Color.GREY);

// ставим алгоритм размытия

dropShadow.setBlurType(BlurType.GAUSSIAN);

Text textf = new Text("Изображение c помощью DropShadow!");

textf.setFill(Color.GOLD);

textf.setEffect(dropShadow);

textf.setFont(Font.font("Calibri", FontWeight.MEDIUM, FontPosture.ITALIC, 21));

root.getChildren().addAll(textf);

myStage.show();

}

@Override

public void stop(){

System.out.println("Программа прекратилась.");

}

public static void main(String[] args) {

launch(args);

}

}

Эффект Grow

Эффект Glow – эффект, который освещает узел. Графический пример:

Без Glow:



С Glow:



У данного эффекта два значения:

* input – представляет вход для эффекта свечения.
* level – уровень свечения, от 0 до 1.0

Конструктор передается level. Существует ряд методов настройки:

* setInput() – данный метод представляет возможность взять еще один эффект.
* setLevel() – задает уровень.

Пример:

package com.company;

import javafx.application.Application;

import javafx.geometry.Pos;

import javafx.scene.Scene;

import javafx.scene.control.Button;

import javafx.scene.control.Label;

import javafx.scene.effect.Glow;

import javafx.scene.effect.Reflection;

import javafx.scene.layout.FlowPane;

import javafx.stage.Stage;

public class Main extends Application{

@Override

public void start(Stage primaryStage) {

primaryStage.setTitle("Задача 15");

FlowPane root = new FlowPane(20, 20);

root.setAlignment(Pos.CENTER);

Scene scene = new Scene(root, 200, 200);

primaryStage.setScene(scene);

Label label = new Label("");

Glow glow = new Glow(0.7);

glow.setInput(new Reflection());

Button button1 = new Button("Кнопка");

button1.setMinSize(30, 25);

button1.setOnAction((ae) ->

button1.setEffect(glow)

);

root.getChildren().addAll(button1, label);

primaryStage.show();

}

public static void main(String[] args) {

launch(args);

}

}

Данном случае методу setInput передают эффект Reflection.

Преобразования.

Преобразование поддерживается абстрактным классом Transform. Существует 4 преобразования:

|  |  |
| --- | --- |
| Rotate | Переворачивает узел. |
| Scale | Увеличивает узел. |
| Translate | Перемещает узел. |
| Share | Сдвигает узел. |

Для того чтоб добавить преобразование следует узлу вернуть список преобразований с помощью метода getTransform() и добавить элемент или элементы с помощью add, addAll.

Существует возможность удалить список преобразований методом clear() или конкретного преобразования remove().

Преобразование Scale.

Данное преобразование масштабирует узел. Конструктор класса Scale:

Scale (double *x\_ширина*, double *y\_высота*)

Для автоматической настройки можно использовать:

* setScaleX()
* setScaleY()

Преобразование Rotate.

Rotate осуществляет переворот узла на заданный угол вокруг заданной точки. Эти значения установлены в экземпляре Rotate:

Rotate (double *угол\_поворота*, double *ширина*, double *высота*)

где угол определяет угол поворота в градусах. Координаты центра поворота или опорной точкой , определяется параметрами x и y.

Можно использовать конструктор по умолчанию. Тогда чтоб устанавливать значения используется методы:

setAngle(double угол)

setPivotX(double x)

setPivotY(double y).

Метод setAngle() присваивает угол. Все вышеприведенные методы final void.

Преобразование Translate.

Данное преобразование переводит узел на определенную позицию. Конструктор класса Translate:

Translate (double *x*, double *y*)

где *x*- ширина на позиции, где *y*- высота на позиции.

Есть ряд методов, которые могут ручную настройку позиции:

* setTranslateX()
* setTranslateY()

Демонстрация преобразований.

package com.company;

import javafx.application.Application;

import javafx.event.ActionEvent;

import javafx.geometry.Pos;

import javafx.scene.Scene;

import javafx.scene.control.Button;

import javafx.scene.control.Label;

import javafx.scene.effect.Glow;

import javafx.scene.effect.Reflection;

import javafx.scene.layout.FlowPane;

import javafx.scene.transform.Rotate;

import javafx.scene.transform.Scale;

import javafx.scene.transform.Translate;

import javafx.stage.Stage;

public class Main extends Application{

double x =0;

double y =0;

double angleRot = 0;

double xSc =2.0;

@Override

public void start(Stage primaryStage) {

primaryStage.setTitle("Задача 15");

FlowPane root = new FlowPane(20, 20);

root.setAlignment(Pos.CENTER);

Scene scene = new Scene(root, 200, 200);

primaryStage.setScene(scene);

Scale scale = new Scale(xSc, xSc);

Button button1 = new Button("Кнопка");

button1.setMinSize(30, 25);

button1.getTransforms().add(scale);

button1.setOnAction((ae) -> {

xSc += 2;

if(xSc > 20.0) xSc = 2;

scale.setX(xSc/2);

scale.setY(xSc/2);

}

);

Translate translate = new Translate();

Button buttonT = new Button("Перенос");

buttonT.getTransforms().add(translate);

buttonT.setOnAction((ae) -> {

x +=0.5;

y +=0.5;

translate.setX(x);

translate.setY(y);

});

Rotate rotate = new Rotate();

Button buttonRot = new Button("Rotate");

buttonRot.getTransforms().add(rotate);

buttonRot.setOnAction((ae)-> {

angleRot += 15.0;

rotate.setAngle(angleRot);

rotate.setPivotY(buttonRot.getHeight()/2);

rotate.setPivotY(buttonRot.getWidth()/2);

});

root.getChildren().addAll(button1, buttonT, buttonRot);

primaryStage.show();

}

public static void main(String[] args) {

launch(args);

}

}

Дополнительные возможности.

Подсказка.

Подсказка – это текст, который выводится на наведение курсора на узел с подсказкой. Для того чтоб поставить подсказку следует с помощью метода setTooptip().

Чтобы поставить подсказку для узла, следует создать объект Tooltip и передать ему текст.

Пример переделанной кнопки верхней программы с подсказкой:

Button buttonRot = new Button ("Rotate");  
buttonRot.setTooltip(new Tooltip("Подсказка для Rotate"));

Отключение элементов управления.

В JavaFX существует возможность отключение элемента управления. Условие: отключать узлы можно только в прикладной программе.

Чтоб отключить узел следует методом setDisable() и передать ему true чтобы отключить узел.

Пример двух кнопок, где первая блокирует, а вторая разблокирует. Достигается это с помощью того, что в первом случае мы передаем true, а во втором false.

Button disable = new Button("Отключить кнопки");

disable.setOnAction((ae) -> {

button1.setDisable(true);

buttonRot.setDisable(true);

buttonT.setDisable(true);

});

Button unblock = new Button("Разблокировать кнопки");

unblock.setOnAction((ae) -> {

button1.setDisable(false);

buttonRot.setDisable(false);

buttonT.setDisable(false);

});

Рисование на холсте.

Для рисования на холсте кругов, квадратов и эллипсов, устанавливать цвет и т.д. в JavaFX внедрен ряд полезных классов.

Чтобы создать холст нам следует создать объект класса Canvas . У этого класса следующий конструктор:

Canvas (double, *ширина* double *высота*)

Все методы для рисования графических объектов следует создать объект класса GraphicsContext. В данном классе определен ряд удобных методов для рисования на холсте. Для того чтоб создать объект данного класса следует вызвать у объекта Canvas метод getGraphicsContext2D(), с полным объявлением метода:

GraphicsContext getGraphicsContext2D ()

Вызвав метод strokeLine, можно нарисовать линию. Полное объявление метода:

void strokeLine(double *начало\_X*, double *начало\_Y*, double *конец\_X*, double *конец\_Y*)

Чтоб нарисовать прямоугольник следует воспользоваться методами strokeRect() или fillRect(). Разница в том, что strokeRect() рисует оправу, а второй зарисовывает фигуру сплошным цветом. Полное объявление методов:

void strokeRect (double *начало\_X*, double *начало\_Y*, double *конец\_X*, double *конец\_Y*)

void fillRect (double *начало\_X*, double *начало\_Y*, double *конец\_X*, double *конец\_Y*)

Чтобы нарисовать эллипс следует воспользоваться методами strokeOval() или fillOval (), с полными объявлениями:

void strokeOval(double *начало\_X*, double *начало\_Y*, double *конец\_X*, double *конец\_Y*)

void fillOval(double *начало\_X*, double *начало\_Y*, double *конец\_X*, double *конец\_Y*)

Чтобы поставить текст на холсте следует вызвать метод fillText (), с полным объявлением:

void fillText(String *строка*, double *начало\_X*, double *конец\_Y*,)

Ниже представлен пример использования холста:

package myPackage;

import javafx.application.Application;

import javafx.geometry.Pos;

import javafx.scene.Scene;

import javafx.scene.canvas.Canvas;

import javafx.scene.canvas.GraphicsContext;

import javafx.scene.layout.FlowPane;

import javafx.scene.text.Font;

import javafx.stage.Stage;

public class Main extends Application {

GraphicsContext graphics;

@Override

public void start(Stage primaryStage) {

primaryStage.setTitle("Демонстрация рисования на холсте");

FlowPane root = new FlowPane(20, 20);

root.setAlignment(Pos.CENTER);

Scene scene=new Scene(root, 400, 400);

primaryStage.setScene(scene);

Canvas canvas=new Canvas(350, 350);

graphics=canvas.getGraphicsContext2D();

graphics.strokeLine(180, 120, 280, 220);

graphics.strokeOval(0, 0, 200, 200);

graphics.strokeRect(30, 30, 130, 130);

graphics.setFont(new Font(20));

graphics.fillText("Рисуем на холсте", 150, 250);

root.getChildren().addAll(canvas);

primaryStage.show();

}

public static void main(String[] args) {

launch(args);

}

}

Программирование меню и подменю.

Программирование меню важная часть программирования графического интерфейса в JavaFX. Именно меню представляет доступ к основным функциям. Меню является ключевым понятием для пользователя, так как пользователь всегда пользуется меню и подменю.

К основным элементам меню относят:

* Строка меню, в котором находится главное меню приложение.
* Стандартное меню с выбираемыми пунктами и т.п.
* Всплывающие окна.

В меню также поддерживаются:

* *Оперативные клавиши* – выбор пункта не активизируя меню.
* *Мнемоника* – выбор пункта с клавиатуры после раскрытия меню.
* *Панель инструментов* – панель, которая предоставляет быстрый доступ к основным возможностям.

Основы меню.

Система меню опирается на часть классов из пакета javafx.scene.control. Также следует сказать, что в основном эти классы не настраивают, а используют в исходном виде, потому что их настройка по умолчанию подходит для многих целей. Таблица основных классов:

|  |  |
| --- | --- |
| CheckMenuItem | Отмечает пункт флажком. |
| ContextMenu | Всплывающее окно, обычно по клике правой мышке. |
| Menu | Меню, состоящие из пунктов типа MenuItem. |
| MenuBar | Объект, содержащий объект главного меню. |
| MenuItem | Объект, наполняющий меню. |
| RadioMenuItem | Пункт меню, который отмечается как кнопка-разделитель. |
| SeparatorMenuItem | Визуальный разделитель пунктов. |

Философия системы меню.

Чтобы сделать главное меню нужно получить экземпляр класса MenuBar. Класс MenuBar служит как контейнер для главного меню. В экземпляр MenuBar вводятся экземпляры класса Menu, которые представляют собой отдельные меню. Каждое меню содержит пункты, которые являются объектами класса MenuItem, который определяет пункты.

Также существует возможность создавать пункты отмечаемые флажком или кнопкой-разделителем. Их особенность в том, что они параллельно действуют с элементами управления флажка или кнопки-разделителя. Служебный класс SeparatorMenuItem создает линию, которая разделяет пункты.

Следует учитывать, что MenuItem не наследует класс Node. Это значит, что экземпляры класса MenuItem используются только для Menu. Но класс MenuBar наследует класс Note и именного его следует вводить в граф сцены.

Также класс Menu наследуется от класса MenuItem, что дает возможность создать подменю – меню, вложенное в меню. Так, что чтоб создать подменю надо создать Menu, наполнить экземплярами MenuItem и ввести в другой объект Menu.

При нажатии на пункт генерируется событие действие. Следует учитывать, что текст на пункте это и его имя. Значит, что при генерации события следует ориентироваться именно на эти имена.

К счастью во время генерация события действия его обработке можно использовать лямбы или анонимные классы. В этом случае имя уже известно и не надо анализировать его имя.

При создании контекстных или всплывающих меню следует создать объект класса ContextMenu и наполнить его MenuItem. В практике они активизируется от щелчка правой кнопки мышки. Для создания панели инструментов следует создать объект ToolBar.

Класс MenuBar.

У данного класса только один конструктор, по умолчанию. Так, что при создании его он будет пустым.

Он определяет несколько методов. Метод getMenus() – возвращает список меню, который управляется из строки меню. Именно в него нужно вводить создаваемые меню.

Для того чтоб ввести экземпляр Menu используют метод add(), addAll(). Вводимые меню располагают слева направо в порядке их ввода. Если требуется ввести меню в определенное место:

void add(int *индекс*, Menu *меню*)

Также существует возможность убрать меню с помощью метода remove():

void remove(Menu *меню*)

void remove(int *индекс*)

Также можно и узнать количество элементов с помощью метода size().

Класс Menu.

У данного класса определенно 2 конструктора. В первом передают имя меню:

Menu (String *имя*)

В нем создают меню с заголовком *имя*. Второй конструктор:

Menu (String *имя*, Node *изображение*)

В нем создают меню с заголовком и *изображение*. Также существует возможность ручной настройки с помощи методов setText() и setGraphic().

Чтобы добавить пункт нужно возвратить список методом getItem() и добавить методами add() или addAll(). Также чтоб узнать количество – size().

В список также можно ввести разделитель. Это удобно когда длинные меню.

Класс MenuItem

Данный класс инкапсулирует пункт меню. У данного класса 3 конструктора:

MenuItem ()

MenuItem (String *текст*)

MenuItem (String *текст*, Node *изображение*)

где *текст* – текст, который будет отображаться на пункте.

Во вовремя нажатия на пункт генерируется событие действие. Для его обработки используется метод setOnAction, как и везде собственно. Полное объявление:

final void setOnAction(EventHandler<ActionEvent> *обработчик*)

где *обработчик* – слушатель, который обрабатывает само событие. Также есть возможность заблокировать пункт методом setDisable().

Создание главного меню.

Процесс создания главного меню делится на несколько стадий.

1. Создаем экземпляр MenuBar.
2. Создаем необходимые меню.
3. Заполняем их.
4. Создаем обработчик событий.
5. Добавляем меню в MenuBar

Программа:

package com.company;

import javafx.application.Application;

import javafx.application.Platform;

import javafx.event.ActionEvent;

import javafx.event.EventHandler;

import javafx.scene.Scene;

import javafx.scene.control.\*;

import javafx.scene.layout.BorderPane;

import javafx.stage.Stage;

public class Main extends Application{

@Override

public void start(Stage primaryStage) {

primaryStage.setTitle("Задача 15");

BorderPane root = new BorderPane();

Scene scene = new Scene(root, 300, 300);

primaryStage.setScene(scene);

Label label = new Label("");

MenuBar menu = new MenuBar();

Menu fileMenu = new Menu("Меню");

MenuItem open = new MenuItem("Файл");

MenuItem close = new MenuItem("Закрыть");

MenuItem save = new MenuItem("Сохранить");

MenuItem exit = new MenuItem("Выход");

fileMenu.getItems().addAll(open, save, close, exit);

menu.getMenus().add(fileMenu);

Menu menuOptional = new Menu("Опции");

Menu color = new Menu("Цвет");

MenuItem green = new MenuItem("Зеленый цвет");

MenuItem yellow = new MenuItem("Желтый цвет");

MenuItem orange = new MenuItem("Оранжевый цвет");

color.getItems().addAll(green, yellow, orange);

menuOptional.getItems().add(color);

menu.getMenus().add(menuOptional);

Menu priory = new Menu("Приоритет");

MenuItem high = new MenuItem("Высокий приоритет");

MenuItem low = new MenuItem("Низкий приоритет");

priory.getItems().addAll(high, low);

menuOptional.getItems().add(priory);

EventHandler<ActionEvent> handler = new EventHandler<ActionEvent>() {

@Override

public void handle(ActionEvent event) {

String name =((MenuItem)event.getTarget()).getText();

if(name.equals("Выход")) Platform.exit();

label.setText("Выбран: "+name);

}

};

open.setOnAction(handler);

close.setOnAction(handler);

save.setOnAction(handler);

exit.setOnAction(handler);

green.setOnAction(handler);

yellow.setOnAction(handler);

orange.setOnAction(handler);

high.setOnAction(handler);

low.setOnAction(handler);

root.setTop(menu);

root.setCenter(label);

primaryStage.show();

}

public static void main(String[] args) {

launch(args);

}

}

Добавление мнемоники и оперативных клавиш.

Чтобы добавить мнемонику нужно предварить символ "\_" перед наименованием меню или пункта. Теперь чтоб вызвать меню или пункт с мнемоникой нужно нажать "Alt+*первая\_буква\_наименоваания*". Но мнемоника активизируется в том случае, если поставлен режим мнемоники. Чтоб его активировать его нужно в метод setMnemonicParsing () передать true.

Оперативная клавиша, по сути, что и мнемоника, но не активирует меню. Чтобы указать комбинацию клавиш нужно использовать метод setAccelerator(). Ему нужно передать объект KeyCombination с применением фабричного метода keyCombination(String *клавиши*).

Ввод изображений.

Для того чтоб рядом с пунктом или меню была картинка (изображение) нужно создать объект типа ImageView и передать ему путь к какой-нибудь картинке. И после объект передать, как второй аргумент.

Класс CheckMenuItem.

Данный класс позволяет создать пункт со значком. Конструктор класса CheckMenuItem:

CheckMenuItem (String *имя*)

Здесь *имя* наименование кнопки. Также есть возможность настроить состояние флажка метод setSelected() и передать ему режим. Если передать ему true, то состояние поднятое и наоборот.

Пример:

Menu menuOptional = new Menu("Опции");

Menu color = new Menu ("Цвет");

CheckMenuItem green = new CheckMenuItem ("Зеленый цвет");

CheckMenuItem yellow = new CheckMenuItem ("Желтый цвет");

CheckMenuItem orange = new CheckMenuItem ("Оранжевый цвет");

green.setSelected (true);

color.getItems().addAll(green, yellow, orange);

Чтобы создать кнопку-переключатель нужно создать класс RadioMenuButton. Конструктор нужно передать текст, который будет наименованием нашего пункта. Также у него есть возможность использовать метод setSelected().

Всплывающее меню.

Чтоб создать такое меню следует создать объект класса ContextMenu и передать ему объекты класса MenuItem. Также после нужно внедрить всплывающие меню в какой-то узел. Пример:

package com.company;

import javafx.application.Application;

import javafx.application.Platform;

import javafx.event.ActionEvent;

import javafx.event.EventHandler;

import javafx.geometry.Pos;

import javafx.scene.Scene;

import javafx.scene.control.\*;

import javafx.scene.input.KeyCombination;

import javafx.scene.layout.BorderPane;

import javafx.scene.layout.FlowPane;

import javafx.stage.Stage;

public class Main extends Application{

@Override

public void start(Stage primaryStage) {

primaryStage.setTitle("Задача 15");

BorderPane root = new BorderPane();

Scene scene = new Scene(root, 300, 300);

primaryStage.setScene(scene);

Label label = new Label("");

MenuBar menu = new MenuBar();

Menu fileMenu = new Menu("\_Меню");

fileMenu.setMnemonicParsing(true);

MenuItem open = new MenuItem("Файл");

open.setAccelerator(KeyCombination.keyCombination("Ctrl-O"));

MenuItem close = new MenuItem("Закрыть");

MenuItem save = new MenuItem("Сохранить");

MenuItem exit = new MenuItem("Выход");

fileMenu.getItems().addAll(open, save, close, exit);

menu.getMenus().add(fileMenu);

Menu menuOptional = new Menu("Опции");

Menu color = new Menu("Цвет");

CheckMenuItem green = new CheckMenuItem("Зеленый цвет");

CheckMenuItem yellow = new CheckMenuItem("Желтый цвет");

CheckMenuItem orange = new CheckMenuItem("Оранжевый цвет");

green.setSelected(true);

color.getItems().addAll(green, yellow, orange);

menuOptional.getItems().add(color);

menu.getMenus().add(menuOptional);

Menu priory = new Menu("Приоритет");

RadioMenuItem high = new RadioMenuItem("Высокий приоритет");

RadioMenuItem low = new RadioMenuItem("Низкий приоритет");

high.setSelected(true);

priory.getItems().addAll(high, low);

menuOptional.getItems().add(priory);

TextField textField = new TextField();

textField.setFocusTraversable(true);

textField.setPrefColumnCount(15);

MenuItem copy= new MenuItem("Копировать");

copy.setOnAction((ae) -> textField.copy());

MenuItem paste = new MenuItem("Вставить");

paste.setOnAction((ae) -> textField.paste());

MenuItem cut = new MenuItem("Вырезать");

cut.setOnAction((ae) -> textField.cut());

final ContextMenu contextMenu = new ContextMenu(copy, paste, cut);

textField.setContextMenu(contextMenu);

EventHandler<ActionEvent> handler = new EventHandler<ActionEvent>() {

@Override

public void handle(ActionEvent event) {

String name =((MenuItem)event.getTarget()).getText();

if(name.equals("Выход")) Platform.exit();

label.setText("Выбран: "+name);

}

};

open.setOnAction(handler);

close.setOnAction(handler);

save.setOnAction(handler);

exit.setOnAction(handler);

green.setOnAction(handler);

yellow.setOnAction(handler);

orange.setOnAction(handler);

high.setOnAction(handler);

low.setOnAction(handler);

FlowPane roitFp = new FlowPane(10, 10);

roitFp.setAlignment(Pos.CENTER);

roitFp.getChildren().addAll(label, textField);

root.setTop(menu);

root.setCenter(roitFp);

primaryStage.show();

}

public static void main(String[] args) {

launch(args);

}

}

В JavaFX также можно привязать контекстное меню к самой сцене. Это осуществляется вызовом метода setOnContextMenuRequested() у сцены предварительно передав, слушатель. В том случае, когда уже есть какое-то всплывающие меню можно использовать метод show() для его отображения. У метода show() следующее полное объявление:

final void show(Node *узел*, double *вверх*, double *низ*)

где *узел* элемент управления, с которым связано контекстное меню. Где *вверх*, *низ* координаты местоположения контекстного меню.

Пример кода, который осуществляет такое меню:

MenuItem copy= new MenuItem("Копировать");

copy.setOnAction((ae) -> textField.copy());

MenuItem paste = new MenuItem("Вставить");

paste.setOnAction((ae) -> textField.paste());

MenuItem cut = new MenuItem("Вырезать");

cut.setOnAction((ae) -> textField.cut());

final ContextMenu contextMenu = new ContextMenu(copy, paste, cut);

textField.setContextMenu(contextMenu);

root.setOnContextMenuRequested((ae) ->{

contextMenu.show(root, ae.getScreenX(), ae.getScreenY());

});

Пример создания меню.

Ниже представлен пример по созданию меню с использованием выше описанным материалом:

package com.company;

import javafx.application.Application;

import javafx.application.Platform;

import javafx.event.ActionEvent;

import javafx.event.EventHandler;

import javafx.geometry.Pos;

import javafx.scene.Scene;

import javafx.scene.control.\*;

import javafx.scene.image.Image;

import javafx.scene.image.ImageView;

import javafx.scene.input.KeyCombination;

import javafx.scene.layout.BorderPane;

import javafx.scene.layout.FlowPane;

import javafx.stage.Stage;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

public class Main extends Application {

Image image;

@Override

public void start(Stage primaryStage) throws FileNotFoundException {

primaryStage.setTitle("Задача 15");

BorderPane root = new BorderPane();

Scene scene = new Scene(root, 300, 300);

primaryStage.setScene(scene);

Image image = new Image(new FileInputStream("C:\\Users\\Семья\\Downloads\\delete.png"));

Image imageCut = new Image(

new FileInputStream("C:\\Users\\Семья\\Downloads\\cut-content-button\_icon-icons.com\_72787.png"));

Label label = new Label("");

MenuBar menu = new MenuBar();

Menu fileMenu = new Menu("\_Меню");

fileMenu.setMnemonicParsing(true);

MenuItem open = new MenuItem("Файл");

open.setAccelerator(KeyCombination.keyCombination("Ctrl-O"));

MenuItem close = new MenuItem("Закрыть");

MenuItem save = new MenuItem("Сохранить");

MenuItem exit = new MenuItem("Выход");

fileMenu.getItems().addAll(open, save, close, exit);

menu.getMenus().add(fileMenu);

Menu menuOptional = new Menu("Опции");

Menu color = new Menu("Цвет");

CheckMenuItem green = new CheckMenuItem("Зеленый цвет");

CheckMenuItem yellow = new CheckMenuItem("Желтый цвет");

CheckMenuItem orange = new CheckMenuItem("Оранжевый цвет");

green.setSelected(true);

color.getItems().addAll(green, yellow, orange);

menuOptional.getItems().add(color);

menu.getMenus().add(menuOptional);

Menu priory = new Menu("Приоритет");

RadioMenuItem high = new RadioMenuItem("Высокий приоритет");

RadioMenuItem low = new RadioMenuItem("Низкий приоритет");

high.setSelected(true);

priory.getItems().addAll(high, low);

menuOptional.getItems().add(priory);

TextField textField = new TextField();

textField.setPrefColumnCount(15);

Button copy= new Button("Копировать");

copy.setFocusTraversable(false);

copy.setOnAction((ae) -> textField.copy());

Button paste = new Button("Вставить");

paste.setFocusTraversable(false);

paste.setOnAction((ae) -> textField.paste());

ImageView imageViewCut = new ImageView(imageCut);

imageViewCut.setFitHeight(15);

imageViewCut.setFitWidth(15);

Button cut = new Button("Вырезать", imageViewCut);

cut.setContentDisplay(ContentDisplay.GRAPHIC\_ONLY);

cut.setFocusTraversable(false);

cut.setOnAction((ae) -> textField.cut());

ImageView imageView = new ImageView(image);

imageView.setFitWidth(15);

imageView.setFitHeight(15);

Button clear = new Button("Очистить", imageView);

clear.setContentDisplay(ContentDisplay.GRAPHIC\_ONLY);

clear.setOnAction((ae) -> textField.clear());

ToolBar tool = new ToolBar();

tool.getItems().addAll(copy, paste, cut, clear);

EventHandler<ActionEvent> handler = new EventHandler<ActionEvent>() {

@Override

public void handle(ActionEvent event) {

String name =((MenuItem)event.getTarget()).getText();

if(name.equals("Выход")) Platform.exit();

label.setText("Выбран: "+name);

}

};

open.setOnAction(handler);

close.setOnAction(handler);

save.setOnAction(handler);

exit.setOnAction(handler);

green.setOnAction(handler);

yellow.setOnAction(handler);

orange.setOnAction(handler);

high.setOnAction(handler);

low.setOnAction(handler);

FlowPane roitFp = new FlowPane(10, 10);

roitFp.setAlignment(Pos.CENTER);

roitFp.getChildren().addAll(label, textField);

root.setTop(menu);

root.setCenter(roitFp);

root.setBottom(tool);

primaryStage.show();

}

public static void main(String[] args) {

launch(args);

}

}

Параллельное программирование или Утилита параллелизма.

Введение.

*Параллельное программирование* является в своем роде дополнением к многопоточному программированию. Утилита параллелизма открывает объекты синхронизаторы вроде семафоры, пулы потоков исполнения, диспетчеры, самоблокировка, новые параллельные коллекции.

Все вышесказанное сильно увеличивает производительность программ. Для утилиты параллелизма выделено три пакета. Первый и самый главный java.util. concurrent. В нем определяется основная функциональность параллелизма. К ним относят:

• Синхронизаторы.

• Исполнители.

• Параллельные коллекции.

• Каркас Fork/Join Framework.

*Синхронизаторы* предоставляют высокоуровневые способы синхронизации взаимодействия нескольких потоков. В пакете java.util.concurrent определен ряд классов синхронизаторов, перечисленных ниже:

Semaphore - Реализует классический семафор

CountDownLatch - Ожидает до тех пор, пока не произойдет определенное количество событий

CyclicВarrir - Позволяет группе потоков исполнения войти в режим ожидания в предварительно заданной точке выполнения

Xchanqer – Осуществляет обмен между двумя потоками исполнения.

Phaser - Синхронизирует потоки исполнения, проходящие через несколько фаз операции.

Следует иметь в виду, что каждый синхронизатор предоставляет конкретное решение задачи синхронизации. Благодаря этому можно оптимизировать работу каждого синхронизатора.

*Исполнители управляют исполнением потоков*. На вершине иерархии исполнителей находится интерфейс Executor , предназначенный для запуска потока исполнения. Интерфейс расширяет интерфейс Executor и предоставляет методы, управляющие исполнением.

В пакете java.util .concurrent определяется также служебный класс E x e c u t o r s , содержащий несколько статических методов, упрощающих создание различных исполнителей.

С исполнителями связаны *также интерфейсы Future и Callable* . Интерфейс Future содержит значение, возвращаемое потоком после исполнения. Таким образом, это значение определяется "на будущее", когда поток завершит свое исполнение. Интерфейс Callable определяет поток исполнения, возвращающий значение.

В пакете java.util.concurrent определяется ряд классов *параллельных коллекций*, в том числе ConcurrentHashMap , ConcurrentLinkedQueue и CopyOnWriteArrayList . Они предоставляют параллельные альтернативные варианты для связанных с ними классов, определенных в каркасе коллекций Collections Framework. В каркасе Fork/Join Framework поддерживается параллельное программирование.

К числу основных классов в этом каркасе относятся классы ForkJoinTask , ForkJoinPool , RecursiveTask и RecursiveAction . И наконец, для усовершенствованной синхронизации потоков исполнения в пакете java.util . concurrent определяется перечисление TimeUnit.

Классы, альтернативны синхронизации.

Класс CountDownLatch.

Класс CountDownLatch решает проблемы самоблокировки. Идея состоит в том, что он обладает, определен количеством событий, который должны произойти. Конструктор данного класса:

CountDownLatch(int *событие*)

Для того чтоб самоблокировать поток используется метод await() с двумя формами объявления:

void await( ) throws InterruptedException

boolean await (int *события*, TimeUnit *значение*) throws InterruptedException

Для того чтоб оповестить о события используется метод countDown(). Пример использования самоблокировки.

package com.company;

import java.util.concurrent.CountDownLatch;

import java.util.concurrent.Semaphore;

public class Main {

public static void main(String[] args){

CountDownLatch latch=new CountDownLatch(5);

CountClockExample clockExample=new CountClockExample(latch, "Дядя Страхуев");

try{

latch.await();

} catch (InterruptedException exc){

System.out.println("Ошибка использования самоблокировки: "+exc);

}

System.out.println("Вышел из самоблокировки");

}

}

class CountClockExample implements Runnable{

CountDownLatch latch;

final String s;

CountClockExample(CountDownLatch latch, String name){

this.latch=latch;

s=name;

new Thread(this).start();

}

@Override

public void run() {

try {

System.out.println(s + " Начал свое выполнение");

for (int i = 0; i < 5; i++) {

System.out.println(s + " событие: " + i);

latch.countDown();

Thread.sleep(150);

}

System.out.println(s + " прекратил выполнение");

} catch (InterruptedException exc){

System.out.println("Ошибка использования времени: "+exc);

}

}

}

Класс Semaphore

Идея *семафора* состоит в том, что он содержит значение, значение. Если оно больше нуля, то поток, который требует ресурс, проходит, а если равно нуля, то потоку отказывают. Когда поток входит в какой-то ресурс значение уменьшается на единицу, а если выходит, поднимается на единицу. Класс Semaphore реализует семафор.

Для того чтоб получить разрешение семафору следует вызвать метод acquire(), а для того чтоб освободить release(). Только следует учитывать, что семафор управляет необходимым ресурсом не прямо и не косвенно. Он создает в своем роде сетку, где один поток находящийся внутри может спокойно использовать ресурс, а другой в то время ждет. Преимущество сепаратора в том, что он может содержать два и более потока сразу, в то время обычная синхронизация один. Пример использования Semaphore:

import java.util.concurrent.Semaphore;

public class Main {

public static void main(String[] args){

Semaphore sem=new Semaphore(1);

new StreamOne(sem, "А");

new ComeBackStream(sem, "B");

}

}

// Общий ресурс

class Shared {

static int count = 0;

}

class StreamOne implements Runnable{

String name;

Semaphore sem;

StreamOne(Semaphore sem, String name){

this.name=name;

this.sem=sem;

new Thread(this).start();

}

@Override

public void run() {

try {

System.out.println(name + " ожидает разрешение");

sem.acquire();

System.out.println(name+" получил разрешение");

for(int i=0; i<5; i++) {

Shared.count++;

System.out.println(name + ": " + Shared.count);

Thread.sleep(10);

}

System.out.println("Ручное оповещение");

} catch (InterruptedException exc){

System.out.println("Ошибка использования семафора: «+exc);

}

sem.release();

System.out.println(name+” освободил семафор”);

}

}

class ComeBackStream implements Runnable{

Semaphore sem;

String name;

ComeBackStream(Semaphore sem, String name){

this.sem=sem;

this.name=name;

new Thread(this).start();

}

@Override

public void run() {

try{

System.out.println(name +” ожидает разрешение”);

sem.acquire();

System.out.println(name+” получил разрешение”);

for(int i=0; i<5; i++){

Shared.count--;

System.out.println(name+”: “+Shared.count);

Thread.sleep(10);

}

}catch (InterruptedException exc){

System.out.println(«Ошибка использования семафора»);

}

System.out.println(name+» освобождает разрешение»);

sem.release();

}

}

Класс CyclicBarrier.

Данный класс позволяет в определенный момент перевести несколько потоков в ожидание состояния. Он позволяет определить объект синхронизации, который приостанавливается до тех пор, пока определенное количество потоков исполнения не достигнет некоторой барьерной точки. Конструкторы:

CyclicBarrier (int *количество\_потоков*)

CyclicBarrier (int *количество\_потоков*, Runnable *действие*)

Где *количество\_потоков* - количество потоков, которые должны достигать своего барьера, где *действие* поток, который вызывается после достижений барьера. Для того чтоб обозначить границу барьера у потока следует использовать метод await(). Пример использования данного класса:

package com.company;

import java.util.concurrent.BrokenBarrierException;

import java.util.concurrent.CyclicBarrier;

public class Main {

public static void main(String[] args){

Runnable BarAction= () ->

System.out.println("Барьер достигнут");

CyclicBarrier barrier=new CyclicBarrier(3, BarAction);

System.out.println("Начало выполнения потоков");

new MyThread(barrier, "A");

new MyThread(barrier, "B");

new MyThread(barrier, "C");

}

}

class MyThread implements Runnable{

CyclicBarrier barrier;

String name;

MyThread(CyclicBarrier barrier, String name){

this.barrier=barrier;

this.name=name;

new Thread(this).start();

}

@Override

public void run() {

System.out.println(name);

try {

barrier.await();

} catch (InterruptedException exc){

System.out.println("Ошибка: "+exc);

} catch (BrokenBarrierException exc) {

System.out.println("Ошибка: "+exc);

}

}

}

Класс Exchanger.

Реализация обмена сообщение между потока в классе Exchanger происходит так: объект типа Exchanger ждет, когда потока вызовут метод exchange(). Класс Exchanger является обобщенным и объявляется так:

Exchanger<V>

где V тип передаваемых сообщений. В этом классе определяется метод exchanger(), который служит в своем роде мессенджером.

Пример:

package com.company;

import java.util.concurrent.Exchanger;

public class Main {

public static void main(String[] args){

Exchanger<String> exchanger=new Exchanger<>();

new MakeString(exchanger);

new UseString(exchanger);

}

}

class MakeString implements Runnable{

Exchanger<String> ex=new Exchanger<>();

String str;

MakeString(Exchanger<String> ex){

this.ex=ex;

str =new String();

new Thread(this).start();

}

@Override

public void run() {

char ch='A';

for(int i=0; i<3; i++){

for (int j=0; j<8; j++){

str += (char) ch++;

}

try{

str=ex.exchange(str);

}catch (InterruptedException exc) {

System.out.println(exc);

}

}

}

}

class UseString implements Runnable{

Exchanger<String> ex;

String str;

UseString(Exchanger<String> ex){

this.ex=ex;

new Thread(this).start();

}

@Override

public void run() {

for (int i = 0; i < 3; i++) {

try{

str = ex.exchange(new String());

System.out.println("Получено: "+str);

}catch (InterruptedException exc){

System.out.println(exc);

}

}

}

}

Класс Phaser.

Класс Phaser реализует синхронизацию потов с фазами. Конструктор:

Phaser ()

Phaser (int *количество\_сторон*)

Чтобы зарегистрировать синхронизатор используется метод register(), который возвращает номер фазы. Полное объявление:

int register()

Если нужно известить о конце фазы используются методы arrive() и arriveAndAwaitAdvance(). Их разница в том, что первый не блокирует поток, когда второй блокирует поток.

int arrive()

int arriveAndAwaitAdvance()

Также возможно снять регистр. Для этого следует вызвать метод arriveAndDeregister(). Полное объявление данного метода:

int arriveAndDeregister()

Чтоб получить номер текущей фазы, который начинается с нуля, следует вызвать метод getPhase(), с полным объявлением:

final int getPhase()

Пример такого синхронизатора:

package myPackage;

import java.util.concurrent.Phaser;

public class Main{

public static void main(String[] args) {

Phaser phaser=new Phaser(1);

int curPhaser;

System.out.println("Threads was starting");

new MyThread(phaser, "A");

new MyThread(phaser, "B");

new MyThread(phaser, "C");

curPhaser=phaser.getPhase();

System.out.println("Phase "+curPhaser+" is over");

phaser.arriveAndAwaitAdvance();

curPhaser=phaser.getPhase();

System.out.println("Phase "+curPhaser+" is over");

phaser.arriveAndAwaitAdvance();

curPhaser=phaser.getPhase();

System.out.println("Phase "+curPhaser+" is over");

phaser.arriveAndDeregister();

if(phaser.isTerminated())

System.out.println("Phase is terminated");

}

}

class MyThread implements Runnable{

Phaser phaser;

String name;

MyThread(Phaser phaser, String name){

this.phaser=phaser;

this.name=name;

new Thread(this).start();

phaser.register();

}

@Override

public void run() {

int curPhase;

curPhase =phaser.getPhase();

System.out.println("The thread "+name+" begins phase: "+curPhase);

phaser.arriveAndAwaitAdvance();

try{

Thread.sleep(100);

}catch (InterruptedException exc){

System.out.println(exc);

}

curPhase=phaser.getPhase();

System.out.println("The thread "+name+" begins phase: "+curPhase);

phaser.arriveAndAwaitAdvance();

try{

Thread.sleep(100);

}catch (InterruptedException exc){

System.out.println(exc);

}

curPhase=phaser.getPhase();

System.out.println("The thread "+name+" begins phase: "+curPhase);

phaser.arriveAndDeregister();

try{

Thread.sleep(100);

}catch (InterruptedException exc){

System.out.println(exc);

}

if(phaser.isTerminated())

System.out.println("Phase is terminated");

}

}

Для нормально функционирования этапщика следует расширять класс Phaser и переопределить метод onAdvance(). Он вызывается системой во время перехода от одной фазы к другой фазе. Пример переделанной верхней программой:

package myPackage;

import java.util.concurrent.Phaser;

public class Main{

public static void main(String[] args) {

MyPhaser phaser=new MyPhaser(1, 4);

System.out.println("Thread\'s begin started.");

new MyThread(phaser, "A");

new MyThread(phaser, "B");

new MyThread(phaser, "C");

while (!phaser.isTerminated()) {

phaser.arriveAndAwaitAdvance();

}

System.out.println("Phase synchronized complete");

}

}

class MyPhaser extends Phaser{

int numPhase;

MyPhaser(int parties, int phaseCount){

super(parties);

numPhase=phaseCount-1;

}

@Override

protected boolean onAdvance(int phase, int registeredParties) {

System.out.println("The phase "+phase+" is complete");

if(phase ==numPhase || registeredParties==0) return true;

return false;

}

}

class MyThread implements Runnable{

Phaser phaser;

String name;

MyThread(Phaser phaser, String name){

this.phaser=phaser;

this.name=name;

new Thread(this).start();

phaser.register();

}

@Override

public void run() {

while (!phaser.isTerminated()) {

System.out.println("The thread "+name+" begin phase: "+phaser.getPhase());

phaser.arriveAndAwaitAdvance();

try{

Thread.sleep(10);

}catch (InterruptedException exc){

System.out.println(exc);

}

}

}

}

В данном случае numPhase данная фаза; parties количество сторон; phaseCount общее количество фаз.

Исполнители.

*Исполнитель* поддерживает управление и создает потоки исполнения. В основу лег интерфейс Executors, с основным методом execute() с полным объявлением:

void execute(Runnable *поток*)

Он запускает *поток* исполнения. Интерфейс ExecutoreService расширяет Executors и вводит ряд методов, которые помогают помогать исполнителями. Метод shutdown() останавливает все подконтрольные интерфейсу ExecutoreService потоки.

*Пул потоков* ряд потоков, которые решают разные проблемы. Чтобы получить пул потоков нужно получить объект типа ThreadPoolExecutor или ScheduledThreadPoolExecutor с помощью фабричных методов.

В приведенном ниже коде можно увидеть, как пул потоков из 2 потоков может решать сразу 4 операции (в нашем случае инкрементирование). Потоки будут самоблокироваться. Код:

package com.company;

import java.util.concurrent.\*;

public class Main {

public static void main(String[] args) {

CountDownLatch countDownLatch=new CountDownLatch(5);

CountDownLatch countDownLatch1=new CountDownLatch(5);

CountDownLatch countDownLatch2=new CountDownLatch(5);

CountDownLatch countDownLatch3=new CountDownLatch(50);

ExecutorService service=Executors.newFixedThreadPool(2);

service.execute(new MyThread(countDownLatch, "A"));

service.execute(new MyThread(countDownLatch1, "B"));

service.execute(new MyThread(countDownLatch2, "C"));

service.execute(new MyThread(countDownLatch3, "D"));

try{

countDownLatch.await();

countDownLatch1.await();

countDownLatch2.await();

countDownLatch3.await();

}catch (InterruptedException exc){

System.out.println(exc);

}

service.shutdown();

}

}

class MyThread implements Runnable{

CountDownLatch latch;

String name;

MyThread(CountDownLatch latch, String name){

this.latch=latch;

this.name=name;

new Thread(this);

}

@Override

public void run() {

for(int i=0; i<5; i++) {

System.out.println(name+": "+i);

latch.countDown();

try {

Thread.sleep(10);

}catch (InterruptedException exc){

System.out.println(exc);

}

}

System.out.println("Поток "+name+" закончился");

}

}

Интерфейсы Callable и Future.

Интерфейс Callable реализует поток исполнения, который возвращает результат. Полное объявление:

interface Callable<V>

Где V тип возвращаемых данных. В нем определяется метод call(), который в теле содержится сама задача, полное объявление:

V call() throws Exception

Но при этом выполнение самой задачи от Callable накладывается на интерфейс Future. Чтобы выполнить задачу следует вызвать метод submit().

Пример использования Callable и Future:

package com.company;

import java.util.concurrent.\*;

public class Main {

public static void main(String[] args) {

int [] arr ={1, 6, 3, 7, 2};

ExecutorService service=Executors.newFixedThreadPool(2);

Future<Integer> futureFact;

Future<Double> futureDiv;

Future<int []> futureArr;

System.out.println("Запуск потоков");

futureFact=service.submit(new Factorial(18));

futureDiv=service.submit(new Division(7, 2));

futureArr=service.submit(new Sort(arr));

try{

System.out.println("Факториал: "+futureFact.get());

System.out.println("Частное: "+futureDiv.get());

}catch (InterruptedException exc){

System.out.println(exc);

}catch (ExecutionException exc){

System.out.println(exc);

}

for(int v=0; v<arr.length; v++) {

if(v==0)

System.out.print("["+arr[v]+" ");

else if (v == 4)

System.out.println(arr[v]+"]\n");

else System.out.print(arr[v]+" ");

}

service.shutdown();

System.out.println("Завершение");

}

}

class Factorial implements Callable<Integer>{

int num;

Factorial(int num){

this.num=num;

}

@Override

public Integer call() {

if(num==0) return 1;

return num\*(num-1);

}

}

class Division implements Callable<Double>{

int num1;

int num2;

Division(int num1, int num2){

this.num1=num1;

this.num2=num2;

}

@Override

public Double call() {

if(num2 ==0)

System.out.println("Деление на ноль не возможно");

if(num1 ==1 ||num2==1) return 1.0;

return Double.valueOf(num1 % num2);

}

}

class Sort implements Callable<int[]> {

int [] array;

Sort(int [] array) {

this.array = array;

}

@Override

public int[] call() {

int i = 0;

int goodPairsCounter=0;

while (true) {

if (array[i] > array[i + 1]) {

int q = array[i];

array[i] = array[i + 1];

array[i + 1] = q;

goodPairsCounter=0;

}else {

goodPairsCounter++;

}

i++;

if (i == array.length - 1) {

i = 0;

}

if (goodPairsCounter == array.length - 1) break;

}

return array;

}

}

Блокировка.

*Блокировка* это альтернатива блокам кода synchronized. Блокировка особа полезна, когда двум потокам требуется получить доступ к ресурсам. Блокировку реализует интерфейс Lock. Метод lock() просто ожидает получение блокировки, а метод unlock() действует наоборот; tryLock() смотрит можно ли получить блокировку; newCondition() создает объект типа Condition(данный класс связан с блокировкой, также он определяет методы await() и signal()).

Также существует возможность *рентабельной блокировки*, в которой может исполняющий поток, может повторно использовать блокировку и именно ее использует, как по умолчанию. Эту блокировку реализует интерфейс ReentrantLock. Разница между обычной блокировки в том, что она явна и ей, можно детально управлять. Пример блокировки:

package com.company;

import java.util.concurrent.locks.ReentrantLock;

public class Main {

public static void main(String[] args) {

ReentrantLock lock=new ReentrantLock();

new LockThread(lock, "A");

new LockThread(lock, "B");

}

}

class Shared{

static int count;

}

class LockThread implements Runnable{

ReentrantLock lock;

String name;

LockThread(ReentrantLock lock, String name){

this.lock=lock;

this.name=name;

new Thread(this).start();

}

@Override

public void run() {

try{

System.out.println("Поток "+name+" ожидает блокировку счетчика");

lock.lock();

System.out.println("Поток "+name+" блокирует счетчик");

Shared.count++;

System.out.println(name+": "+Shared.count);

System.out.println("Поток "+name+" ожидает");

Thread.sleep(100);

}catch (InterruptedException exc){

System.out.println(exc);

}finally {

System.out.println("Поток "+name+" разблокирует счетчик");

lock.unlock();

}

}

}

Атомарные операции.

В пакете java.util.concurrent.atomic предоставляется альтернатива другим средствам синхронизации для чтения или записи значений переменных некоторых типов. В этом пакете доступны методы, которые получают, устанавливают или сравнивают значение переменной во время одной непрерывной, т.е. *атомарной*, операции. Это означает, что для выполнения такой операции не требуется ни блокировка, ни любой другой механизм синхронизации.

Атомарные операции выполняются с помощью классов Atomicinteger и AtomicLong , а также методов get( ) , set( ) , compareAndSet ( ) , decrementAndGet ( ) и getAndSet ( ) , которые реализуют соответственно следующие действия: получение, установку, сравнение и установку, декремент и получение, получение и установку. Пример применения атомарных операций:

package com.company;

import java.util.concurrent.atomic.AtomicInteger;

public class Main {

public static void main(String[] args) {

new AtomThread("A");

new AtomThread("B");

new AtomThread("C");

}

}

class AtomShared{

static AtomicInteger count=new AtomicInteger();

}

class AtomThread implements Runnable{

String name;

AtomThread(String name){

this.name=name;

new Thread(this).start();

}

@Override

public void run() {

for(int i=0; i<3; i++)

System.out.println("Поток "+name+": "+AtomShared.count.getAndSet(i));

}

}

Каркас Fork/Join Framework.

*Параллельное программирование* – это использование методик по многоядерному вычислению. Его реализуют классы и интерфейсы из каркаса Fork/Join Framework из пакета java.concurrent .

Во время одноядерных машин многопоточное программирование использовало лишь для использования времени простоя, а теперь во время многоядерных машин можно наращивать вычислительные процессы.

Ядро основных классов данного каркаса:

* ForkJoinTask <V> - абстрактный класс, определяющий задачу.
* ForkJoinPool – управляет задачей ForkJoinTask.
* RecursiveAction – производные от ForkJoinPool, для задач, которые не возвращают результат.
* RecursiveTask<V> - от ForkJoinPool, для задач, которые возвращают результат.

Класс ForkJoinPool, получается, управляет задачей абстрактного класса ForkJoinTask<V>. ForkJoinTask расширяет RecursiveAction, RecursiveTask<V>. Эти классы обычно расширяются в прикладной программе ради решения проблемы.

Класс ForkJoinTask<V>

Данный класс определяет задачу, которая будет управляться ForkJoinPool. Параметр V обозначает тип задачи.

Класс ForkJoinTask отличается от Thread в том, что он предоставляет абстракцию, а не поток. Из методов следует отметить:

final ForkJoinTask<V> fork()

final V join()

Метод fork() вызывает задачу для асинхронного выполнения, как только задача запланирована на выполнение, тогда возвращается this ссылка на объект задачи. Метод join() ожидает завершение задачи и возвращает результат.

Метод invoke() соединяет операцию вилочного соединения в единый вызов или запускает задачу на выполнение и ожидает выполнение, возвращая результат выполнения программы. Формы:

final V invoke()

final void invokeAll(ForkJoinTask<V> *задача*, ForkJoinTask<V> *задача*)

final void invokeAll(ForkJoinTask<V> … *задачи*)

Класс RecursiveAction

Он является производным от ForkJoinTask и инкапсулирует задачу, которая не возвращает результат. Обычно его расширяют в прикладном коде и в нем задачу. Само же задачу кладут в метод compute():

protect abstract void compute()

Этот метод можно вызвать в методах этого класса или в производственных классах. Метод compute() следует реализовать. Также RecursiveAction используется в стратегии «Разделяй и властвуй».

Класс RecursiveTask<V>

Класс RecursiveTask<V> схож с RecursiveAction, но возвращает результат выполнения. Где V тип возвращаемого результата. Метод compute() действует также, но возвращает результат. Полное объявление:

protect V compute()

Класс RecursiveTask служит для реализации рекурсивной стратегии выполнения задачи. Он также применяется в стратегии «разделяй и властвуй».

ForkJoinPool

Класс ForkJoinPool выполняет задачи в виде пула. Используется, как управляющий над ForkJoinTask. Конструкторы:

ForkJoinPool ()

ForkJoinPool (int *уровень\_параллелизма*)

В первом случае пул создается по умолчанию, во втором случае с указанным количеством потоков, которые могут выполняться одновременно. Нужно учитывать, что *уровень\_параллелизма* - цель, а не средства. Также есть создать *общий\_пул,* т.е. статистический объект ForkJoinPool для использования. Это можно сделать с помощью метода commonPool(), с полным объявлением:

static ForkJoinPool commonPool()

Пул управляет потоками по принципу *перехват работы*. У каждого потока есть очередь задач, которые он должен выполнить, а если у него пустая очередь он берет задачу другого потока.

Самый простой вызвать задачу – использовать метод invoke(), с полным объявлением:

<T> T invoke(ForkJoinPool<T> *задача*)

Чтобы вызвать и не ждать конца, добиш синхронно выполнения задачи используется метод execute():

void execute(ForkJoinPool<?> *задача*)

Также следует учитывать, что потоки пула являются *потоками доменами* и это значит, что они автоматически закроются, хотя можно сделать и явно с помощью метода shutdown().

Тактика «Разделяй и властвуй»

Данная тактика предлагает разделение задачи на подзадачи, а эти подзадачи и далее дедуктивным способом и до тех пор, пока не появится минимальное значение операций, т.е. *порог*. Данная стратегия используется для классов RecursiveAction и RecursiveTask.

Плюс такого способа – параллельное выполнение. Для успешного использования «Разделяй и властвуй» нужно правильно объявить порог. Порог следует ставить в пределах 100-10000, и если значение поставлено не правильно, то лучше чтоб порог был выше нужного значения, чем ниже.

Пример простого вилочного соединения.

В приведенной ниже программе значения типа double преобразуют в их квадратный корень:

package com.company;

import java.util.concurrent.ForkJoinPool;

import java.util.concurrent.RecursiveAction;

public class Main {

public static void main(String[] args) {

double [] data=new double[1000];

ForkJoinPool joinPool=ForkJoinPool.commonPool();

for(int i=0; i<1000; i++)

data[i] =(double) i;

System.out.println("Сотая часть массива:");

for(int i=0; i<10; i++)

System.out.print(data[i]+" ");

SqrtTransform sqrt=new SqrtTransform(data, 0, data.length);

sqrt.invoke();

System.out.println("\nКорни сотой части массива: ");

for(int i=0; i<10; i++)

System.out.printf("%.4f ", data[i]);

System.out.println();

}

}

class SqrtTransform extends RecursiveAction{

final int MAXVAL=1000;

double [] arr;

int start, end;

SqrtTransform(double [] arr, int start, int end){

this.arr=arr;

this.start=start;

this.end=end;

}

@Override

protected void compute() {

if((end-start) < MAXVAL){

for(int i=start; i<end; i++){

arr[i] =Math.sqrt(arr[i]);

}

}else{

int miblle=(start+end) /2;

invokeAll(new SqrtTransform(arr, start, miblle),

new SqrtTransform(arr, miblle, end));

}

}

}

Простой пример RecursiveTask

В приведенной ниже программе значения из массива считывают и их возвращают:

package myPackage;

import java.util.concurrent.ForkJoinPool;

import java.util.concurrent.RecursiveTask;

public class Main {

public static void main(String[] args) {

int [] data=new int[10000];

int result;

for(int i=0; i<data.length; i++)

data[i]=((i%2)==0 ? i : -i);

ForkJoinPool joinPool=new ForkJoinPool();

Sum tesk=new Sum(data, 0, data.length);

result=joinPool.invoke(tesk);

System.out.println("Суммирование: "+result);

}

}

class Sum extends RecursiveTask<Integer>{

int sum=0;

int [] arr;

int start, end;

int thershold=500;

Sum(int [] array, int start, int end){

arr=array;

this.end=end;

this.start=start;

}

@Override

protected Integer compute() {

if((end-start) <thershold){

for(int i=0; i<arr.length; i++) {

sum += arr[i];

}

}else {

int q=(end+start) /2;

Sum task1=new Sum(arr, start, q);

Sum task2=new Sum(arr, q, end);

task1.fork();

task2.fork();

sum=task1.join()+task2.join();

}

return sum;

}

}

Следует обратить внимание на выполнение обоих подзадач:

task1.fork();

task2.fork();

Используется метод fork() так, как он вызывает задачу асинхронно. Также заслуживает внимание следующая строка:

sum=task1.join()+task2.join();

В данной строке результаты выполнений задач присваиваются в переменную sum. Хотя есть альтернативы:

sum=task1.join()+task2.invoke()

sum=task1.join()+task2.compute()

Еще один пример параллельной программы.

Третий пример параллельной программы:

import java.util.concurrent.ForkJoinPool;

import java.util.concurrent.RecursiveTask;

public class Main {

public static void main(String[] args) {

char [] array = new char[600];

for(char c = 0; c<array.length; c++)

array[c] = c;

ForkJoinPool pool=new ForkJoinPool();

char[] invoke\_res = pool.invoke(new MyThread(new String(array), 0, array.length));

for(char c : invoke\_res)

System.out.print(c+" ");

}

}

class MyThread extends RecursiveTask<char []> {

String text;

int start, end;

final int THRESHOLD= 1000;

MyThread (String str, int start, int end){

text=str;

this.start=start;

this.end=end;

}

@Override

protected char [] compute() {

char [] array=new char[text.length() / 2];

if((end-start) < THRESHOLD){

for(char c ='a', i =0; c<array.length && i<array.length; c+= 2, i++)

array[i] = c;

}else {

int q = (end - start) / 2;

MyThread t = new MyThread(text, start, q);

MyThread t1 = new MyThread(text, q, end);

t.fork();

t1.fork();

array = (t.join().toString() + t1.join().toString()).toCharArray();

}

return array;

}

}

Другие избранные возможности.

На вышесказанном не заканчивается каркас Fork/Join Framework и есть еще много возможностей, благодаря которым можно удобно управлять и писать программу.

*Асинхронное выполнение задач*: Когда нужно не ожидать выполнения задачи можно вызвать ее асинхронно, т.е. не дожидаться его выполнения. Это можно сделать, если вызвать метод execute() с двумя версиями выполнения:

void execute(ForkJoinTask<V> *задача*)

void execute(Runnable *задача*)

Тут можно отметить то, что можно использовать уже стандартные потоки. Это в своем роде мост между многопоточностью и параллельным программированием. Нужно учитывать, что потоки в пуле потоки демоны и для их полноценного выполнения нужно поддерживать жизнь пула.

*Отмена задачи*: Чтоб отменить задачу можно вызвать метод cancel() из класса ForkJoinPool. Общая форма данного метода:

boolean cancel(boolean *режим*)

Метод cancel() возвращает логическое значение true, если поток успешно отменен или false. Чтоб узнать, что отменен, ли поток можно вызвать метод isCancelled().

*Определение состояния завершения задачи:* Также есть другие методы, который возвращают состояние. К ним относят:

* isCompletedNormally(), который проверяет, завершилась ли программа, т.е. не случилось ли отмены или генерации исключений.
* isCompletedAdNormally() противоположность isCompletedNormally()

Ниже приведены полные объявления методов:

final boolean isCompletedNormally()

final boolean isCompletedAdNormally()

*Перезапуск задачи*: Обычно перезапустить задачу во время ее выполнения нельзя ее перезапустить, но после выполнения задачу можно вернуть к прошлому состоянию задачи. Для этого можно вызвать метод reinitialize() с полным объявлением:

void reinitialize()

Избранные способности из класса ForkJoinTask

С помощью метода adapt() можно адаптировать объекты типа Runnable и Callable в объекты типа ForkJoinPool .

Вызвав метод getQueuedTaskCount() можно определить количество задач на поток, а getSurplusQueuedTaskCount () узнать количество потоков, чем потоков в пуле, которые могут перехватить задачу.

Методом tryUnfork() можно отменить вызов задачи. Иными слова исключить задачу из плана выполнения.

Упрощенными аналогами join() и invoke(), которые не кидают исключения и не возвращают, являются quietlyJoin() и quietlyInvoke(), с приведенными ниже полными объявлениями:

final void quietlyJoin ()

final void quietlyInvoke ()

Избранные средства из ForkJoinPool

Вызвав метод isQuiescent() можно узнать, не бездействует ли поток. Вызвав метод getPoolSize() можно получить количество потоков в пуле. Методом getActiveThreadCount () - приблизительное количество активных потоков исполнения в пуле.

Чтоб закрыть пул, но продолжать выполнение задач и не брать новые задачи следует вызвать shutdown(). Если нужно даже прекратить выполнение задач у пула нужно вызвать shutdownNow(). Чтоб проверить пул на активность нужно вызвать isShutDown(). Вызвав метод isTerminate() модно узнать закрыт ли пул и задачи выполнены.

Рекомендации.

Чтоб правильно и эффективно использовать параллельное программирование нужно следовать следующими рекомендациями:

* Не следует ставить уровень параллелизма.
* Чем выше порог, тем и лучше.
* Не использовать синхронизированные методы и блоки.
* Использовать другие средства синхронизации
* Не использовать средства ввода-вывода, которые могут привести к долгой блокировке.
* Не делать предположения относительно среды исполнения.

Пакет java.lang.

Оболочки примитивных типов.

Класс Number

Абстрактный класс Number реализует классы оболочки, к ним относят классы: Integer, Double, Float, Long, Character и Byte. Он содержит методы, которые возвращают методы в разных числовых форматах. Полные объявления этих методов:

int intValue()

double doubleValue()

float floatValue()

byte byteValue()

short shortValue()

long longValue()

Классы Double и Float

Классы Double и Float служат оболочками для примитивных типов с плавающей точкой. Конструкторы класса Float:

Float (float *значение*)

Float (double *значение*)

Float (String *строка*) throws NumberFormatException

Объекты типа Float можно создать из значения типа float или double. Также есть возможность *строку* преобразовать в значение. Конструкторы, определенные в классе Double:

Double (double *значение*)

Double (String *строка*) throws NumberFormatException

Значения данных классов:

BYTES - Длина типа float или double в байтах.

МАХ\_EXPONENT - Максимальный показатель степени.

МАХ\_VALUE - Максимальное положительное значение.

MIN\_ EXPONENT- Минимальный показатель степени.

MIN\_NORMAL - Минимальное положительное нормальное значение.

MIN\_ VALUE - Минимальное положительное значение.

NaN - Не число.

POSITIVE\_INFINITY - Положительная бесконечность.

NEGATIVE \_ INFINITY - Отрицательная бесконечность.

SIZE - Размер заключенного в оболочку значения в битах.

ТУРЕ - Объект типа Class для типов float и double .

Пример использования данной оболочки:

package myPackage;

public class Main{

public static void main(String[] args) throws IOException{

Double double1 = new Double(3.14);

Double double2 =new Double("314E-2");

System.out.println(double1+" и "+double2+" равны "+double1.equals(double2));

Double endless=new Double(1/0.);

Double nan=new Double(0/0.);

System.out.println("Число\t"+" Бесконечное число\t"+"Не число");

System.out.println(endless+"\t "+endless.isInfinite()+"\t "+endless.isNaN());

System.out.println(nan+"\t "+nan.isInfinite()+"\t "+nan.isNaN());

}

}

Несколько слов об дробных литералы: их обозначает буквой E и через дефис пишут положение точки относительно конца.

Оболочки для Byte, Short, Integer, Long.

Классы Byte, Short, Integer, Long служат оболочками для byte, short, int, long соответственно. Конструкторы:

Byte (byte *значение*)

Byte (String *значение*) throws NumberFormatException

Short (short *значение*)

Short (String *значение*) throws NumberFormatException

Integer (int *значение*)

Integer (String *значение*) throws NumberFormatException

Long (long *значение*)

Long (String *значение*) throws NumberFormatException

В этих классах определяют ряд значений:

MIN\_VALUE - Минимальное значение

МАХ\_VALUE - Максимальное значение

SIZE - Длина заключенного в оболочку значения в битах

ТУРЕ - Объект типа Class для типов byte, short, int или long

Ниже представлен пример использования данных оболочек. В них реализуется способ преобразования строки в число для суммирования и возвращения в его бинарной, восьмеричной и шестнадцатеричной форме:

package myPackage;

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

public class Main{

public static void main(String[] args) throws IOException {

BufferedReader buffer=new BufferedReader(new InputStreamReader(System.in));

String str;

int sum=0;

int i;

System.out.println("Введите число, 0 чтоб выйти");

do{

str=buffer.readLine();

try{

i=Integer.parseInt(str);

}catch (NumberFormatException exc){

System.out.println(exc);

i=0;

}

sum+=i;

System.out.println("Текущая сумма: "+sum);

}while (i !=0);

System.out.println(sum+" в двоичном представлении "+

Integer.toBinaryString(sum));

System.out.println(sum+" в восьмеричном представлении "+

Integer.toOctalString(sum));

System.out.println(sum+" в шестнадцатеричном представлении "+

Integer.toHexString(sum));

}

}

Оболочка Character

Данный класс служит оболочкой для переменных типа char. Конструктор класса:

Character (char *значение*)

Чтобы получить значения следует вызвать метод charValue() c полным объявлением:

char charValue()

В данном классе также определен ряд значений:

BYTES - Длина типа char в байтах

МАХ\_RADIX - Максимальное основание системы счисления

MIN\_RADIX- Минимальное основание системы счисления

МАХ\_VALUE - Максимальное значение

MIN\_ VALUE - Минимальное значение

ТУРЕ - Объект типа Class для типа char

В приведенной ниже программе демонстрируется использование Character:

package myPackage;

public class Main{

public static void main(String[] args) {

char a[] ={'A', ' ', 'a', '5'};

for(char i = 0; i < a.length; i++){

System.out.println("Символ: "+a[i]);

if(Character.isLetter(a[i])) {

System.out.println("Это знак");

if (Character.isUpperCase(a[i]))

System.out.println("Это заглавная буква");

if (Character.isLowerCase(a[i]))

System.out.println("Это прописная буква");

}

if(Character.isDigit(a[i]))

System.out.println("Это цифра");

if(Character.isWhitespace(a[i]))

System.out.println("Это знак пробела");

}

}

}

Класс Boolean

Класс Boolean служит очень тонкой оболочкой для логических значений типа boolean , что удобно в тех случаях, когда логические значения требуется передавать по ссылке. Этот класс содержит константы TRUE и FALSE. В классе Boolean определяется также поле TYPE , являющееся объектом типа Class для типа boolean . В класс Boolean определены следующие конструкторы:

Вoolean (boolean лor•vecxoe знаvеюrе) Вoolean (String лоГJ11vесхая\_ с !l'рОха)

Boolean(boolean *значение*)

Boolean(String *строка*)

Класс Void

Класс Void содержит значение TYPE, в котором хранится ссылка на класс Class для типа void. У данного класса не создают экземпляры.

Класс Process.

Абстрактный класс Process инкапсулирует *процесс,* т.е. выполнение программы. Используется, как суперкласс объектам, которые создаются при помощи метода exec() класса Runtime или start() из класса ProcessBuilder. Методы из данного класса:

void destroy() – прерывает процесс.

boolean isAlive() – возвращает true если процесс еще выполняется.

Int waitFor() throws InterruptedException - возвращает код завершения процесса. Не возвращает его до тех пор, пока процесс, для которых он вызван, не завершится.

int extValue() – возвращает код завершения процесса.

…

Класс Runtime

Класс Runtime инкапсулирует исполняющую среду. Чтоб получить ссылку на данный класс следует использовать статистический метод getRuntime(). JVM не принимает апплеты и прочие недостоверные источники. Методы данного класса:

static Runtime getRuntime() – возвращает ссылку на Runtime

void load(String *файл\_библиотека*) - загружает файл с ссылкой *файл\_библиотеки*.

void addShutdownHook(Thread *thread*) – запускает *thread* на выполнении после остановки JVM

int availableProcessors() - возвращает количество процессоров, доступных для JVM.

long maxMemory() –возвращает максимальное количество памяти доступной виртуальной машине.

void halt(int *код*) –прерывает работу JVM. Компилятору возвращает *код*.

long totalMemory() – возвращает общий размер памяти.

void runFinalization() – запускает финализацию.

long freeMemory() - возвращает количество пустых байтов.

void gc() – запускает сборщик мусора.

boolean removeShutdownHook(Thread *поток*) – удаляет *поток*.

Управление памятью с помощью Runtime

Хоть управление памятью в Java автоматическая, но есть возможность самостоятельно ее чистить. Для этих целей в классе Runtime определенны методы gc() и runFinalization(), также объявлены методы, которые осведомляют о самой памяти: freeMemory(), maxMemory() и totalMemory(). В приведенной ниже программе сначала наполняется память, а позже очищается, ради простоты количество памяти возвращается. Код:

package myPackage;

public class Main{

public static void main(String[] args) {

Runtime runtime=Runtime.getRuntime();

long mem1, mem2;

Integer [] arr=new Integer[1000];

System.out.println("Количество памяти: "+runtime.totalMemory());

runtime.gc();

mem1=runtime.freeMemory();

System.out.println("Исходное количество памяти "+mem1);

for(int i=0; i<arr.length; i++)

arr[i] =i;

mem2=runtime.freeMemory();

System.out.println("Память после очищения "+mem2);

System.out.println("Использовано памяти "+(mem1-mem2));

for(int i=0; i<arr.length; i++)

arr[i]=null;

runtime.gc();// begin Cleanup garbage

System.out.println("Конечное количество памяти "+runtime.totalMemory());

}

}

Выполнение других программ.

В безопасных средах можно в программе выполнять другие программы. Для этого следует объекты типа Process вызвать метод exec() и передать ему имя программы, хотя у данного метода есть много разных форм, но обычно используется имеено первая. В приведенном ниже примеры открывается программа Notepad, ожидается его прекращения и возвращается результат :

package myPackage;

public class Main{

public static void main(String[] args) {

Runtime runtime=Runtime.getRuntime();

Process process=null;

try{

process=runtime.exec("Notepad");

process.waitFor();

} catch (Exception e) {

System.out.println(e);

}

System.out.println("Программа блокнот вернула: "+process.exitValue());

}

}

Класс ProcessBuilder

Как упоминалось ранние есть второй способ использования программы в программе. Он заключается в том, что использовать класс ProcessBuilder вызвав у него метод start. В приведенной ниже программе приведен пример его использования:

package myPackage;

public class Main{

public static void main(String[] args) {

try {

ProcessBuilder builder=

new ProcessBuilder("notepad.exe", "testBilder");

builder.start();

}catch (Exception exc){

System.out.println(exc);

}

}

}

Класс System

Класс System содержит ряд статических методов удобных использования ряд важных переменных. К переменным относят in, out, err.

Измерение времени

В классе System есть методы позволяющие измерять время выполнения. К ним относят System.nanoTime() и System.currentTimeMillis(). Полные объявления:

static long currentTimeMillis()

static long nanoTime()

Ниже приведен пример использования метода nanoTime():

package myPackage;

public class Main{

public static void main(String[] args) {

long start, end;

System.out.println("Измерение время переборки от 1 до 10000000");

start=System.nanoTime();

for(int i=0; i<10000000l; i++);

end=System.nanoTime();

System.out.println("Время перебор: "+(end-start));

}

}

Копирование массива

В классе System есть статический метод arraycopy(), который позволяет удобно копировать массивы. Полное объявление методы:

static void arraycopy(Object *откуда*, int *начало\_источника*,

Object *адресат*, int *начало\_адресата*, int *размер*)

Ниже представлен вариант его использования:

package myPackage;

public class Main{

static byte [] a={65, 66, 67, 68, 69, 70, 71, 72, 73, 74};

static byte [] b={77, 77, 77, 77, 77, 77, 77, 77, 77, 77};

public static void main(String[] args) {

System.out.println("a -> "+new String(a));

System.out.println("b -> "+new String(b));

System.arraycopy(a, 0, b, 0, a.length);

System.out.println("Массивы после копирования\n"+"a -> "+new String(a));

System.out.println("b -> "+new String(b));

System.arraycopy(a, 0, a, 1, a.length-1);

System.arraycopy(b, 0, b, 1, b.length-1);

System.out.println("Массивы после смещения на одну позицию:\n"+"a -> "+new String(a));

System.out.println("b -> "+new String(b));

}

}

Класс Object.

Данный класс уже упоминался, но здесь будет подробно рассмотрена операция клонирования. Вообще клонировать объекта следует только в очень редких случаях на свой страх и риск. Минусы клонирования:

* Неопределенность ввода-вывода. Это подразумевает то, что поток ввода-вывода не знает, в какой из клонированных объектов читать/писать.
* Если оба клонированных объекта ссылаются ссылку объекта и если из клонированного объекта изменить объект, то упадет сама система.

Для операции клонирования определен метод clone(), ниже приведено его полное объявление:

Object clone() throws CloneNotSupportedException

Для того чтоб использовать метод clone() класс должен реализовывать интерфейс Cloneable. Если класс клонируемого объекта реализует Cloneable, тогда генерируется исключение CloneNotSupportedException. Его использования его следует писать свой вариант метода clone(). Ниже представлен пример использования клонирования:

package myPackage;

public class Main{

public static void main(String[] args) {

Share share=new Share();

share.a=34;

Share share1;

share.b=32;

share1=(Share)share.clone();

System.out.println("Клонируемый ресурс Share:"+

share1.a+" "+share1.b);

}

}

class CloneObject implements Cloneable{

public Object clone() {

try {

return super.clone();

} catch (CloneNotSupportedException exc) {

System.out.println(exc);

return this;

}

}

}

class Share extends CloneObject implements Cloneable{

int a;

int b;

}

Класс Class

Класс Class инкапсулирует состояние время выполнения класса или интерфейса. Полное объявление данного класса:

class Class<A>

где A класс объекта или интерфейса. Данный класс нельзя получить явно, поэтому используется метод getClass() из класса Object. Его (класс) следует использовать, когда нужно получить информацию о классе или интерфейсе. Ниже приведен пример использования данного класса:

package myPackage;

public class Main{

public static void main(String[] args) {

Class<?> clasObj;

A a=new A();

B b=new B();

clasObj=a.getClass();

System.out.println("Объект a типа: "+clasObj.getName());

clasObj=b.getClass();

System.out.println("Объект b типа: "+clasObj.getName());

clasObj=clasObj.getSuperclass();

System.out.println("Суперкласс для класса B "+clasObj.getName());

}

}

class A{

char a;

}

class B extends A{

char b;

}

Класс ClassLoader.

Абстрактный класс ClassLoader определяет порядок загрузок классов. Благодаря его можно указать свой порядок загрузки классов, но этого никогда не принято делать.

Класс Compiler.

Данный класс поддерживает создание сред, где байт-код компилируется в исполнительный код вместо интерпретируемый. Он объявлен с модификатором final и не может иметь подклассов.

Классы Thread, ThreadGroup и интерфейс Runnable

Это в своем роде дополнение уже существующими знаниями об многопоточности.

Интерфейс Runnable

Этот интерфейс должен реализовать любой класс инициируемый поток. Он содержит один метод run(), ниже приведен полное объявление в прикладном коде:

void run()

Класс Thread

Класс Thread создает новый поток исполнения. В нем реализуется интерфейс Runnable. Конструкторы:

Thread ()

Thread (Runnable *объект\_потока*)

Thread (Runnable *объект\_потока*, String *имя\_потока*)

Thread (String *имя\_потока*)

Thread (ThreadGroup *группа*, Runnable *объект\_потока*)

Thread (ThreadGroup *группа*, Runnable *объект\_потока*, String *имя\_потока*)

Thread (ThreadGroup *группа*, String *имя\_потока*)

В классе объявлены следующие значения:

MAX\_PRIORIITY

NORM\_PRIORITY

MIN\_PRIORITY

Все они обозначают приоритет потока.

Класс ThreadGroup

В этом классе определяется группа потоков. Конструктор:

ThreadGroup(String *имя*)

ThreadGroup(ThreadGroup *родительская\_группа*, String *имя*)

Ниже представлен пример его использования:

package myPackage;

public class Main {

public static void main(String[] args) {

ThreadGroup group=new ThreadGroup("Группа A");

ThreadGroup group1=new ThreadGroup("Группа B");

MyThread thread=new MyThread("A", group);

MyThread thread1=new MyThread("B", group);

MyThread thread2=new MyThread("C", group1);

MyThread thread3=new MyThread("D", group1);

System.out.println("\nВывод из метода list(): ");

group.list();

group1.list();

System.out.println();

System.out.println("Прерывается группа A");

Thread [] tga=new Thread[group.activeCount()];

group.enumerate(tga);

for(int i=0; i< tga.length; i++)

((MyThread)tga[i]).mySuspend();

try{

Thread.sleep(4000);

}catch (InterruptedException exc){

System.out.println(exc);

}

System.out.println("Возобновление группы A");

for(int i=0; i<tga.length; i++)

((MyThread)tga[i]).myResum();

try{

thread.join();

thread1.join();

thread2.join();

thread3.join();

}catch (InterruptedException exc){

System.out.println(exc);

}

System.out.println("Главный поток прерван");

}

}

class MyThread extends Thread{

boolean suspendFlag;

String name;

MyThread(String name, ThreadGroup group){

super(group, name);

this.name=name;

System.out.println("New thread "+name);

suspendFlag=false;

start();

}

@Override

public void run() {

for(int i=0; i<5; i++){

System.out.println(name+": "+i);

synchronized (this){

while (suspendFlag)

try {

wait();

}catch (InterruptedException exc){

System.out.println(exc);

}

}

}

System.out.println(name+" закончил выполнение");

}

synchronized void mySuspend(){

suspendFlag=true;

}

synchronized void myResum(){

suspendFlag=false;

notify();

}

}

Класс RuntimePermission.

Класс RuntimePermission относится к механизму защиты Java. Он не может иметь подклассов и объявляет два конструктора. Кроме того он не объявляет никаких методов.

Класс SecurityManager.

Класс SecurityManager служит суперклассом для создания диспетчера защиты. Получить текущий SecurityManager можно статическим метод System.getSecurityManager().

Класс StackTraceElement.

В классе описывается единственный стековый фрейм - отдельный элемент трассировки стека при возникновении исключения . Каждый стековый фрейм представляет тачку выполнения, которая включает имя класса, метода и файла, а также номер строки исходного кода. Массив элементов типа StackTraceElement возвращается при вызове метода getStackTrace ( ) из класса Throwable .

У класса StackTraceElement имеется следующий единственный конструктор:

StackTraceElement ( String *имя\_класса*, String *имя\_метода*,

String *имя*\_*файла* , int *строка*)

Где параметр *имя*\_ *класса* обозначает имя конкретного класса; параметр имя\_метода - имя конкретного метода; параметр *имя*\_ *файла* - имя конкретного файла; параметр *строка* - номер строки кода. В отсутствие достоверного номера строки кода в качестве параметра строка следует указать отрицательное значение. Но если стековый фрейм ссылается на платформа – ориентированный параметр строка должен принять -2.

Он определяет геттеры для класса, файла, метода и строки и isNativeMethod . Еще он определяет hashCode (), equals () и toString ().

Класс ClassValue <T>.

Класс ClassValue применяется для связи с переменными. Он используется для разработки Java и не используется для обычных программистов.

Интерфейс Thread.UncaughtExceptionHandler

Данный статический интерфейс следует реализовывать, когда требуется обработать необрабатываемое исключение. Он определяет uncaughtException (), с полным объявлением:

void uncaughtException (Thread *поток*, Throwable *исключение*)

Здесь *поток* ссылка на поток исполнения, сгенерировавший *исключение*.

Регулярные выражения

*Регулярное выражение* или *шаблон* – строка описывающая последовательность символов. Их использует при *сопоставлении* со строками. Пример регулярного выражения обычный адрес где-либо. Все классы, связанные с регулярными выражениями находятся в java.util.regex.

Класс Pattern

Класс Pattern определяет шаблон. У данного класса нет конструктора, и для создания объекта используют фабричный метод compile(), с приведенным ниже объявлением:

static Pattern compile(String *шаблон*)

Также в нем определен фабричный метод, который создает объект Matcher, который сопоставляет шаблон. Полное объявление:

Matcher matcher (CharSequence *строка*)

Класс Matcher

В нем определен ряд методов для сопоставления с шаблоном. Самым простым способом определить совпадает ли строка с шаблоном – использовать метод matches(). Полное объявление данного метода:

boolean matches()

Чтоб определить совпадает, ли шаблон с выходящей последовательности следует вызвать метод find(), с полным объявлением:

boolean find()

Символьную строку, которая содержит последнюю совпавшую последовательность можно получить, вызвав метод group(), с полным объявлением:

String group()

Получить индекс текущего совпадения, можно вызвав метод start(), полное объявление:

int start()

Получить индекс следующего совпадения, можно вызвав метод end(), полное объявление:

int end()

Каждую совпавшую последовательность можно заменить, вызвав метод replaceAll(), с полным объявлением:

String replaceAll (String *новая\_строка*)

Синтаксис регулярных выражений.

В общем, регулярное выражение состоит из обычных символов, кванторов, метасимволов и классов символов.

Обычные символы называют *литералами*. Обычный символ сопоставляется в исходном виде. Символы вроде \n обозначаю косой чертой флеша(\).

Классы символы обозначает набор символов. Они заключают в квадратные скобочки []. Например класс символов [adc] может совпасть с символом a b или c. Также есть возможность создать исключающий класс символов, если в начале класса поставить ^, пример [^abc].

Метасимволом служит (.). Он обозначает, что в этом символе может быть любой символ. Квантор определяет, сколько раз будет совпадать символ. Список кванторов:

+ Обозначает совпадение один раз и более

\* Обозначает совпадение нуль и более

? Обозначает совпадение нуль и раз

Пример сопоставления.

Ниже представлен пример использования с литералами:

package myPackage;

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class Main{

public static void main(String[] args) {

// Пример просто регулярного выражения

Pattern pattern=Pattern.compile("Java");

Matcher matcher=pattern.matcher("Java");

if(matcher.matches())

System.out.println("Первая строка совпадает");

else

System.out.println("Первая строка не совпадает");

matcher=pattern.matcher("Java SE");

if(matcher.matches())

System.out.println("Вторая строка совпадает");

else

System.out.println("Вторая строка не совпадает");

}

}

Ниже представлен пример использования классов символов и метасимвола:

package myPackage;

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class Main{

public static void main(String[] args) {

boolean found;

Pattern pattern=Pattern.compile("Java [1-8].");

Matcher matcher=pattern.matcher("Java 8!");

found=matcher.matches();

if(found)

System.out.println("Совпадает");

else

System.out.println("Не совпадает ");

}

}

Ниже приведен пример использования метода group():

package myPackage;

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class Main{

public static void main(String[] args) {

Pattern pat2=Pattern.compile("e.+?d");

Matcher matcher=pat2.matcher("extend cut end result");

while (matcher.find())

System.out.println(": "+matcher.group());

}

}

Вывод из программы:

Последовательность: extend

Последовательность: end

Следует отметить, если не указать «?», тогда будет использоваться *строгое* совпадение и в примере будет:

Последовательность: extend cut end

Ниже приведен пример использования replaceAll():

package myPackage;

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class Main{

public static void main(String[] args) {

boolean res;

String str="John Alex John";

Pattern pattern=Pattern.compile("John.+? John");

Matcher matcher2=pattern.matcher(str);

System.out.println(": "+str);

str=matcher2.replaceAll("Eric");

System.out.println("Измененная строка: "+str);

}

}

Ниже приведен пример использования метода slip():

package myPackage;

import java.util.regex.Pattern;

public class Main{

public static void main(String[] args) {

boolean res;

Pattern pattern1=Pattern.compile("[,.!]");

String [] array=pattern1.split("one\_two, alpha9! beach.") ;

for(int i=0; i<array.length; i++)

System.out.println("Лексема "+array[i]);

}

}

Альтернативные способы поиска.

Метод matches() удобно использовать один раз для сопоставления, полное объявление:

static boolean matches(String *шаблон*, CharSequence *последовательность*)

Также в классе String определяется свой вариант метода matches(), с полным объявлением:

boolean matches(String *шаблон*)

Ниже приведен пример их использования:

package myPackage;

import java.util.regex.Pattern;

public class Main{

public static void main(String[] args) {

boolean res;

String s=" Java 8";

res=s.matches(" Jav. [2-8]");

if(res)

System.out.println("Совпадает");

else

System.out.println("Не совпадает");

res=Pattern.matches(".+ Java", "Случайная строка");

if (res)

System.out.println("Совпадают");

else

System.out.println("Не совпадают");

}

}

Рефлексия

*Рефлексия* – способность программного обеспечения к самоанализу во время выполнения, а не компиляции. Такая способность обеспечивается пакетом java.lang.reflect. Рефлексии является важной частью Java Beans.

В пакете java.lang.reflect входит интерфейс Member, поскольку он определяет ряд методов, которые получают сведения о полях класса, конструкторах класса и его методах. В пакете еще есть восемь классов:

AccessibleObject – Позволяет обходить стандартные проверки управления доступом

Array – Позволяет динамически создавать массивы и ими управлять

Constructor – Содержит сведения о конструкторе

Field – Содержит сведения о полях

Method – Содержит сведения о методах

Modifier - Предоставляет сведения о модификаторах класса и его членов

Proxy – Поддерживает динамически прокси-классы.

ReflectPermission – Разрешает рефлексию закрытых или защищенных членов класса

Ниже приведен пример рефлексии:

package myPackage;

import java.lang.reflect.Constructor;

import java.lang.reflect.Field;

import java.lang.reflect.Method;

public class Main{

public static void main(String[] args) throws ClassNotFoundException {

Class<?> cl=Class.forName("java.lang.String");

System.out.println("Конструкторы: ");

Constructor [] constructors=cl.getConstructors();

for(int i=0; i<constructors.length; i++)

System.out.println(constructors[i]);

System.out.println("Методы: ");

Method [] methods=cl.getMethods();

for(int i=0; i<methods.length; i++)

System.out.println(methods[i]);

System.out.println("Поля: ");

Field [] fields=cl.getFields();

for(int i=0; i<fields.length; i++)

System.out.println(fields[i]);

}

}

Прежде всего, стоит отметить создание ссылки на класс. Это осуществляется метод forName(), которому передаем имя класса. Далее конструкторы получаются методом getConstructors(), методы getMethods() и поля getFields().

Класс Modifier

Класс Modifier содержит информацию о модификаторах к чему-либо. В нем определяется ряд статических методов типа isX(). Ниже приведен пример использования Modifier:

package myPackage;

import java.lang.reflect.Method;

import java.lang.reflect.Modifier;

public class Main{

public static void main(String[] args){

A a=new A();

Class<?> cl=a.getClass();

Method [] methods=cl.getDeclaredMethods();

System.out.println("Отрытые метода : ");

for(int i=0; i<methods.length; i++){

if(Modifier.isPublic(methods[i].getModifiers()))

System.out.println(" "+methods[i].getName());

}

}

}

class A {

public void a1(){

}

private void a2(){

}

protected void a3(){

}

}

Вызов закрытого метода.

Вызвать закрытый метод вроде не реализуемая возможность, но благодаря рефлексии эта способность обеспечена. Это идет как в плюс и как в минус рефлексии и общем Java API . Ниже представлен пример вызова закрытого метода:

package myPackage;

import java.lang.reflect.Constructor;

import java.lang.reflect.InvocationTargetException;

import java.lang.reflect.Method;

public class Main {

public static void main(String[] args)

throws NoSuchMethodException, IllegalAccessException,

InvocationTargetException, InstantiationException

{

Constructor<Hello> constructor=Hello.class.getConstructor(String.class);

Hello hello=constructor.newInstance("Саша");

Method method=Hello.class.getDeclaredMethod("greet");

method.setAccessible(true);

method.invoke(hello );

}

}

class Hello{

final String name;

public Hello(String name){

this.name=name;

}

private void greet(){

System.out.println("Hello, "+name);

}

}

Пакет java.util

Класс StringTokenizer

Класс StringTokenizer разбивает строку на отдельные части – *лексемы*. Это называют *синтаксический анализ*. У класса определенно три конструктора:

StringTokenizer (String *строка*)

StringTokenizer (String *строка*, String *разделители*)

StringTokenizer (String *строка*, String *разделители*, boolean *разделитель\_как\_лексема*)

Где *строка* обозначает строку, которую следует разделить. В первой форме разделители используются по умолчанию, а во второй указываются свои разделители. В третий форме определяется, возвращать ли разделитель вместе с лексемой.

Для того чтоб получить лексему следует воспользоваться методом nextToken(), а чтоб проверить на наличие лексем следует воспользоваться методом hasMoreTokens(). Также следует отметить, что StringTokenizer реализует Enumeration то можно и воспользоваться его методами.

Ниже программа разделяет строку на 3 отдельных лексем:

package myPackage;

import java.util.StringTokenizer;

public class Main {

public static void main(String[] args) {

String [] strings= new String[]{};

String str="Это строка,"+"будет;"+"разделена!";

StringTokenizer tokenizer=

new StringTokenizer(str, ",;");

while (tokenizer.hasMoreTokens()){

strings=

new String[]{tokenizer.nextToken(),

tokenizer.nextToken(),

tokenizer.nextToken()};

}

for(String v : strings){

System.out.println(v);

}

}

}

Класс BitSet.

Класс BetSet служит в своем роде вектором для битов. Биты представляются в виде логических значений true 1 или false 0. В классе BetSet определено два конструктора:

BitSet()

BitSet (int *размер*)

В первом создается изначально пустой массив, который в последствии наращивается. Во второй массив с начальным *размером*.

Ниже представлен пример использования битового вектора:

package myPackage;

import java.util.BitSet;

public class Main {

public static void main(String[] args) {

BitSet set1=new BitSet(16);

BitSet set2=new BitSet(16);

for(int i=0; i<16; i++){

if((i%2)==0) set1.set(i);

if((i%5)!=0) set2.set(i);

}

System.out.println(set1);

System.out.println(set2);

set2.and(set1);

System.out.println("После and():\n"+set2);

set2.or(set1);

System.out.println("После or():\n"+set2);

set2.xor(set1);

System.out.println("После xor():\n"+set2);

}

}

Здесь методы and(), or() и xor() выполняют логические операции И, ИЛИ и исключающие ИЛИ.\

Класс Date.

Класс Date инкапсулирует время в миллисекундах с первого января 1970 года. В нем определен ряд устаревших методов. Ниже представлены два конструктора:

Date ()

Date (long *миллисекунды*)

В первом конструкторе создается дату сегодняшнего дня, а в второй время указывается в *миллисекунды*. Ниже представлены методы:

boolean after(Date *дата*) – Проверяет вызывающею дату на содержание более поздней *дата*.

boolean before(Date *дата*) - Проверяет вызывающею дату на содержание более ранней *дата*.

long getTime() – Возвращает время в миллисекундах после 1 января 1970 года.

void setTime(long *время*) – Устанавливает на дату *время*.

Object clone () – Дублирует вызывающую дату.

int compareTo (Date *дата*) – Сравнивает вызывающую дату с *дата*.

boolean equals(Date *дата*) – Сравнивает дату с *дата* .

int hashCode() – Возвращает хэш-код даты.

String toString() – Возвращает дату в виде строки.

Ниже представлен пример использования класса Date:

package myPackage;

import java.util.Date;

public class Main {

public static void main(String[] args) {

Date today=new Date();

System.out.println(today);

System.out.println("Время в миллисекундах с 1.01.70: "+today.getTime());

}

}

Класс Calendar.

Абстрактный класс Calendar позволяет задавать дату в понятном для нас виде. Для того чтоб получить объект следует вызвать статический метод getInstance(), а для того чтоб поставить следует вызвать метод set().

Ниже представлен пример использования класса Calendar:

package myPackage;

import java.util.Calendar;

public class Main {

public static void main(String[] args) {

Calendar today=Calendar.getInstance();

Calendar birthday=Calendar.getInstance();

birthday.set(2018, 2, 5);

if(today.equals(birthday))

System.out.println("C днем рождения");

else

System.out.println("Сегодня будний день");

}

}

Класс Observable.

Данный класс позволяет создавать подклассы, за которые могут наблюдать отдельные части программы. Для создания подкласса следует расширить класс Observable. А наблюдающие классы должны реализовывать метод update() в интерфейс Observer.

Наблюдаемый объект должен вызвать два метода. Когда у него происходит изменение он должен вызвать метод setChanged(). И когда он готов известить наблюдателя он должен вызвать метод notifyObservers(). Также для того чтоб поставить наблюдатель следует вызвать метод addObservers().

Интерфейс Observers

Чтоб создать наблюдателя, следует реализовать интерфейс Observers с методом update(). Полное объявление метода:

void update (Observable *наблюдаемый*, Object *аргумент*)

где *наблюдаемый* объект за которым будет вести наблюдение, а *аргумент* должен быть аргумент, передаваемый в метод notifyObservers().

Пример наблюдения за объектом.

Ниже представлен пример использования для наблюдения за объектом:

package myPackage;

import java.util.Observable;

import java.util.Observer;

public class Main {

public static void main(String[] args) {

Watcher watch=new Watcher();

Begin beginWatch=new Begin();

beginWatch.addObserver(watch);

beginWatch.counter(10);

}

}

class Watcher implements Observer{

@Override

public void update(Observable o, Object arg) {

System.out.println("В методе update "+((Integer)arg).intValue());

}

}

class Begin extends Observable{

void counter(int period){

for(; period>=0; period--){

setChanged();

notifyObservers(new Integer(period));

try{

Thread.sleep(100);

}catch (InterruptedException exc){

System.out.println(exc);

}

}

}

}

Классы TimerTask и Timer.

Данные классы обеспечивают возможность выполнение задачи, через определенное время. Класс TimerTask задает саму задачу, а класс Timer выполняет задачу через определенное время.

Класс TimerTask реализует интерфейс Runnable. Он содержит абстрактный метод run(), который содержит задачу. Также в нем есть метод cancel(), который останавливает поток.

Класс Timer выполняет саму задачу. Ниже представлены конструкторы:

Timer ()

Timer (boolean *поток\_демон*)

Timer (String *имя*)

Timer (String *имя*, boolean *поток\_демон*)

Где *поток\_демон* обозначает будет ли поток демоном. Для того чтоб указать, что должен выполнять объект Timer следует воспользоваться методом schedule().

Ниже представлен пример использования данных классов:

package myPackage;

import java.util.Timer;

import java.util.TimerTask;

public class Main {

public static void main(String[] args) {

MyTimerTask task=new MyTimerTask();

Timer time=new Timer();

time.schedule(task, 500);

}

}

class MyTimerTask extends TimerTask {

@Override

public void run() {

for(int i=0; i<12; i++){

System.out.println("Цикл задачи"+i);

try{

Thread.sleep(100);

}catch (InterruptedException exc){

System.out.println(exc);

}

if(i==10) {

System.out.println("Конец выполнения программы");

this.cancel();

}

}

}

}

Класс Currency.

Класс Currency инкапсулирует сведения о денежной системы. У него нет конструкторов, для этого существует статический метод getInstance (). Ниже представлен пример использования класс Currency:

package myPackage;

import java.util.Currency;

import java.util.Locale;

import java.util.Set;

public class Main {

public static void main(String[] args) {

Currency money;

// Получаем сведение о денежной системы в США

money=Currency.getInstance(Locale.US);

// Получаем имя денежной валюты.

System.out.println("Название денежной валюты в США: "+money.getDisplayName());

// Создаем множество валют.

Set<Currency> currencies=Currency.getAvailableCurrencies();

for (Currency currency : currencies) {

System.out.println("Валюта: "+currency.getSymbol()+", "+currency.getDisplayName());

}

}

}

Форматирование средствами класс Formatter.

Данный класс позволяет форматировать выходящие данные. Он преобразует двоичные данные в форматированный текст.

В нем определенно много конструкторов, ниже они перечислены:

Formatter ()

Formatter (Appendable *буфер*)

Formatter (Appendable *буфер*, Locale *региональные\_настройки*)

Formatter (String *имя\_файла*) throws FileNotFoundException

Formatter (File *файл\_ввода*) throws FileNotFoundException

Здесь *буфер* будет хранить отформатированные данные. Где *региональные\_настройки* обозначает региональные настройки. Где *имя\_файла* файл, который будет содержать форматированный текст. А *файл\_ввода* ссылка на открытый файл, принимающий форматирующий текст.

Форматирование методом format().

Основным достоинство класса Formatter является метод format(). Он форматирует передаваемый текст с определенными аргументами. Ниже представлено полное объявление данного метода:

void format ( String *строка*, Object … *аргументы*)

Здесь *строка* состоит из двух частей: символ – простой текст, который копируется в буфер; *спецификатор доступа* – это способ, который должен выводится *аргументы*.

Спецификатор доступ обозначается символом % и послеидущей буквой. К примеру %f обозначает, что здесь аргумент должен быть отформатирован как число с плавающей точкой. Ниже перечислены основные спецификаторы доступов:

%а %А - Шестнадцатеричное значение с плавающей точкой.

%b %B - Логическое значение.

%c – Символ.

%d – Десятичное целое число.

%h %H – Хэш-код аргумента.

%е %E – Экспоненциальное представление числа.

%f – Десятичное число.

%g %G – Использует %e или %f в зависимости от случая.

%o – Для восьмеричных чисел.

%n – Знак перевода строки.

%s %S – Строка.

%t %Т – Дата.

%x %Х – Шестнадцатеричное представление.

%% - Знак %

Ниже представлен пример форматирования:

package myPackage;

import java.util.Formatter;

public class Main {

public static void main(String[] args) {

Formatter format=new Formatter();

format.format("Это форматирование %s: %d %f", "средствами Formatter", 109, 0.3);

System.out.println(format.toString());

}

}

Форматирование дат.

Наиболее эффективным средством является форматирование дат. Их форматируют с помощью спецификатора доступа %t, но для него определен ряд суффиксов, которые могут выводить часы, эру, часовой пояс и т.д. Ниже они перечислены:

a – Сокращенное название дня недели

A – Полное название дня недели

b – Сокращенное название месяца

B – Полное название месяца

c – Стандартная строка времени *день месяц дата чч:мм:сс часовой пояс год*

C – Первые две цифры года

d – Номер числа в месяце

F – Формат «год-число-месяц»

H – Часы (0-23)

I – Часы (1-12)

j – День города в десятичном представлении

k – Часы (0-23)

l – Часы (1-12)

L – Миллисекунды

m – Месяц в десятичном представлении (1-13)

M – Минуты в десятичном представлении (1-59)

N – Наносекунды

P – Представляет am и pm

Q – миллисекунды с 1.1.1970

r – Представляет время в формате *чч*:*мм*:*сс*

R – Представляет время в формате *чч*:*мм*

S – секунды

s – Миллисекунды с 01.01.70

T – Представляет время в формате *чч:мм:сс* (24 часовой формат)

y – Год десятичном представлении (0-99)

Y- Год в десятичном представлении, включая столетия (1-9999)

z – Смещение относительно UTC

Z – Наименование часового пояса

Ниже представлен пример форматирования дат:

package myPackage;

import java.util.Calendar;

import java.util.Formatter;

import java.util.Locale;

public class Main {

public static void main(String[] args) {

Formatter format=new Formatter(Locale.ENGLISH);

Calendar today=Calendar.getInstance();

format.format("%tr ", today);

System.out.println(format);

format=new Formatter(Locale.ENGLISH);

format.format("%tc ", today);

System.out.println(format);

format=new Formatter(Locale.ENGLISH);

format.format("%tl:%tm", today, today);

System.out.println(format);

format=new Formatter(Locale.ENGLISH);

format.format("%tb %td %tm ", today, today, today);

System.out.println(format);

}

}

Указание минимальной ширины поля.

Для того чтоб указать ширину поля следует между % вставить целое число, которое обозначает минимальную ширину. Если нужно чтоб в начала был ноль или ноли можно перед целым числом указать необходимые ноли. К примеру: %012f. Ниже представлен пример указания ширины поля:

package myPackage;

import java.util.Formatter;

public class Main {

public static void main(String[] args) {

Formatter format=new Formatter();

format.format("|%f|%n|%12f|%n|%012f|", 10.12345, 10.12345, 10.12345);

System.out.println(format);

format.close();

System.out.println();

for (int i = 1; i <=10 ; i++) {

format=new Formatter();

format.format("%4d %4d %4d ", i, i\*i, i\*i\*i);

System.out.println(format);

format.close();

}

}

}

Признаки форматов.

Класс Formatter распознает ряд *признаков форматов*. Они позволяют определять аспекты форматирования. Ниже перечислены признаки форматов:

|  |  |
| --- | --- |
| - | Выравнивает текст по левому краю |
| + | Ставит плюс перед положительным числом |
| ( | Ставить отрицательное число в скобки |
| пробел | Положительные числа выводятся с пробелом |
| , | Числовое значение дополняется «,» |
| # | Альтернативный способ преобразования |
| 0 | У выводимых данных пробелы заменяют на 0 |

Ниже представлен пример использования признаков форматов:

package myPackage;

import java.util.Formatter;

public class Main {

public static void main(String[] args) {

Formatter format=new Formatter();

format.format("%(.2f", -12.395);

System.out.println(format);

format.close();

format=new Formatter();

format.format("%+.3f", 12.452);

System.out.println(format);

format.close();

format=new Formatter();

format.format("|%-10.2f|", 123.123);

System.out.println(format);

format.close();

format=new Formatter();

format.format("% d", 10);

System.out.println(format);

format.close();

format=new Formatter();

format.format("%,.2f", 4321567.891);

System.out.println(format);

format.close();

}

}

Оперирование индексами аргументов.

Как правило все аргументы идут слева, но порой удобно указать где, какой аргумент должен быть. Индекс аргумента указывается после % со знаком $. Если же нужно чтоб все модификаторы доступов использовании один и тот же аргумент. Ниже представлен пример использования индексов аргументов:

package myPackage;

import java.util.Formatter;

public class Main {

public static void main(String[] args) {

Formatter format=new Formatter();

format.format("%3$d %1$d %2$d", 12, 120, 68);

System.out.println(format);

format.close();

format=new Formatter();

format.format("%d %<d", 12);

System.out.println(format);

}

}

Автоматическое закрытие класса Formatter

Начиная с JDK 7 класс Formatter реализует интерфейс AutoCloseable. И благодаря этому можно использовать try с ресурсами. Ниже представлен пример использования автоматического закрытия:

package myPackage;

import java.util.Formatter;

public class Main {

public static void main(String[] args) {

try(Formatter format=new Formatter()){

format.format("Форматировать %s просто %d %.1f", "средствами Java", 12, 12.5);

System.out.println(format);

}

}

}

Класс Scanner.

Класс Scanner читает отформатированные вводные данные и преобразует их в двоичном представлении. Он служит чтения данных из консоли, файлов, объектов класса String, иных источников реализующих интерфейс Readable или ReadableByteChannel.

Конструкторы класса Scanner.

Класс Scanner определяет много конструкторов. Ниже они перечислены:

Scanner(File *source*) throws FileNotFoundException –

создает сканнер, производящий данные из *source*.

Scanner (File *source*, String *chartset*) throws FileNotFoundException –

создает сканнер, производящий данные из *source*, сопоставляющая *chartset*.

Scanner (InputStream *stream*) –

читает данные с консоли, т.е. System.in.

Scanner (InputStream *stream*, String *chartset*) –

читает данные с консоли, т.е. System.in, сопоставляющая *chartset*.

Scanner (Path *путь*) throws IOException –

создает сканнер, производящие из *путь*.

Scanner (Path *путь*, String *chartset*) throws IOException –

создает сканнер, производящие из *путь*, сопоставляющая *chartset*.

Scanner (Readable *source*) –

создает сканнер, производящий из *source*.

Scanner (ReadableByteChannel *source*) –

создает сканнер, производящий из *source*.

Scanner (ReadableByteChannel *source*, String *chartset*) –

создает сканнер, производящий из *source*, сопоставляющая *chartset*.

Scanner (String *строка*) –

создает сканнер, производящий из *строка*.

Основы сканирования

Сканнер входящие данные разделяет на *токены*, которые разделяются (если это нужно) с помощью *разделителей*. По умолчанию разделителями служат пробелы и клавиша Enter. Таким образом для использования сканирования используется следующий план:

* Удостоверится на наличие данных.
* Получить данные методами типа next()
* Если нужно получить ряд токенов повторить верхние планы.
* Закрыть с помощью метода close()

Стоит отметить, что класс реализует AutoCloseable и поэтому для них можно использовать try с ресурсами.

Ниже представлен пример использования скана в Java:

package myPackage;

import java.util.Scanner;

public class Main {

public static void main(String[] args) {

int i=0;

try (Scanner scanner = new Scanner(System.in)) {

System.out.println("Введите число: ");

while (scanner.hasNextInt()) {

i = scanner.nextInt();

if (i == 0) {

scanner.close();

}

System.out.println("Вы ввели: "+i);

}

}catch (IllegalStateException exc){

System.out.println("Сканнер заблокирован");

i=-1;

}

}

}

Здесь в цикле while проверяется на наличие данных типа int. Есть такие же методы для double, float и т.п., а для полной строки hasNextLine().

В случае строке i = scanner.nextInt(); к переменной присваивается переменная типа int. Такие методы есть для long, short и т.п., а для полной строки nextLine (), а просто следующий токен next ().

Установка разделителей.

Токены разделяются разделителями. По умолчанию разделителями являются пробелы и клавиша Enter. Но в классе Scanner определен метод useDelimiter(), позволяющий вносить разделители. Ниже приведены две формы данного метода:

Scanner useDelimiter (String *шаблон*)

Scanner useDelimiter (Pattern *паттерн*)

Для того чтоб получить все разделители можно воспользоваться методом delimiter(). Ниже представлен пример внедрения разделителей:

package myPackage;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

import java.util.Scanner;

public class Main {

public static void main(String[] args) throws IOException {

int count=0;

int res=0;

FileWriter fileWriter=new FileWriter("text.txt");

fileWriter.write("12, 3, 45, готово");

fileWriter.close();

FileReader reader=new FileReader("Text.txt");

Scanner scanner=new Scanner(reader);

scanner.useDelimiter(", \*");

while (scanner.hasNext()){

if(scanner.hasNextInt()){

count++;

res += scanner.nextInt();

}else {

String str=scanner.next();

if(str.equals("готово"))

break;

else {

System.out.println("Ошибка формата");

return;

}

}

}

scanner.close();

System.out.println("Среднеарифметическое чисел: "+(res/count));

System.out.println("Разделители: "+scanner.delimiter());

}

}

Пакет java.text.

В пакете java.text предназначен для манипулирования , форматирования, поиска и синтаксического анализа текста. Он также определяет современные средства для работы с времен.

Класс DateFormat.

Абстрактный класс DateFormat и задает средства форматирования, анализировать дату и/или время. В нем определенно пять констант, которые обозначают стиль:

* SHORT
* DEFAULT
* MEDIUM
* LONG
* FULL

Для того чтоб создать объект DateFormat, следует вызвать один из трех методов. Первый из них getDateInstance(), он возвращает объект, инкапсулирующий данные о дате. У него определенно три формы:

static final DateFormat getDateInstance()

static final DateFormat getDateInstance(int *стиль*)

static final DateFormat getDateInstance(int *стиль*, Locale *локация*)

Чтобы получить объект, инкапсулирующий данные о времени можно воспользоваться методом getTimeInstance(). У него также определенно три формы:

static final DateFormat getTimeInstance()

static final DateFormat getTimeInstance(int *стиль*)

static final DateFormat getTimeInstance(int *стиль*, Locale *локация*)

Чтобы получить объект, инкапсулирующий данные о времени и дате можно воспользоваться методом getDateTimeInstance(). У него также определенно три формы:

static final DateFormat getDateTimeInstance()

static final DateFormat getDateTimeInstance(int *стиль\_дня*, int *стиль\_времени*)

static final DateFormat getDateTimeInstance(int *стиль\_дня*, int *стиль\_времени*, Locale *локация*)

Для того чтоб воспользоваться объектом для отображения даты следует воспользоваться методом format(). У него определяется перегружаемые формы, но используется одна версия:

final String format (Date *дата*)

Ниже представлен пример использования данного класса:

package myPackage;

import java.text.DateFormat;

import java.util.Date;

import java.util.Locale;

public class Main {

public static void main(String[] args) {

Date today=new Date();

DateFormat fd=DateFormat.getDateInstance(DateFormat.SHORT, Locale.JAPAN);

DateFormat ft = DateFormat.getTimeInstance(DateFormat.LONG, Locale.JAPAN);

DateFormat fdt = DateFormat.getDateTimeInstance(

DateFormat.MEDIUM, DateFormat.MEDIUM, Locale.JAPAN);

System.out.println("Дата по японскому варианту: "+fd.format(today));

System.out.println("Время по японскому варианту: "+ft.format(today));

System.out.println("Дата ив время по японскому варианту: "+fdt.format(today));

fd=DateFormat.getDateInstance(DateFormat.SHORT, Locale.ENGLISH);

ft=DateFormat.getTimeInstance(DateFormat.LONG, Locale.ENGLISH);

fdt=DateFormat.getDateTimeInstance(DateFormat.MEDIUM, DateFormat.MEDIUM, Locale.ENGLISH);

System.out.println("\n"+"Дата по английскому варианту: "+fd.format(today));

System.out.println("Время по английскому варианту: "+ft.format(today));

System.out.println("Дата и время по английскому варианту: "+fdt.format(today));

}

}

Класс SimpleDateFormat

Данный класс позволяет задать свою форму даты. Он расширяет абстрактный класс DateFormat. Конструктор данного класса:

SimpleDateFormat (String *шаблон*)

Здесь дату будет выводится по заданному *шаблону*. Ниже перечислены шаблонные буквы:

Письмо Компонент даты или времени презентация Примеры

G Эра-указатель Текст AD

y Год Число 1996; 96

Y Недельный год Число 2009; 09

M Месяц в году (с учетом контекста) Число July; Jul;07

L Месяц в году (отдельная форма) Число July; Jul;07

w Неделя в году Число 27

W Неделя месяца Число 2

D День в году Число 189

d День месяца Число 10

F День недели в месяц Число 2

E Название дня в неделю Текст Tuesday; Tue

u День недели Число 1

a Маркер am / pm Текст PM

H Час в день (0-23) Число 0

k Час в день (1-24) Число 24

K Час в am / pm (0-11) Число 0

h Час в am / pm (1-12) Число 12

m Минута в час Число 30

s Второй в минуту Число 55

S миллисекунды Число 978

Ниже представлен пример использования SimpleDateFormat:

package myPackage;

import java.text.SimpleDateFormat;

import java.util.Date;

public class Main {

public static void main(String[] args) {

Date date=new Date();

SimpleDateFormat formatDate =new SimpleDateFormat("Сегодня: WW EEE \n HH:MM");

System.out.println(formatDate.format(date));

}

}

Форматирование чисел.

В java.text определяется три класса для форматирования чисел. Первый это абстрактный класс NumberFormat в нем определяется основные методы: format() и parse(). Подклассами NumberFormat являются DecimalFormat, определяющий средства для работы с десятичными числами и ChoseFormat, определяющий формат по диапазону чисел.

NumberFormat

Экземпляр данного класса можно получить статическим методом getInstance(). У него два объявления. Первый создает экземпляр, который будет форматировать числа по локации по умолчанию, а второй в параметры получает *локацию*:

static NumberFormat getInstance()

static NumberFormat getInstance(Locale *локализация*)

Методы format() и parse() действуют схожим образом как и в остальных случаях. Ниже представлен пример использования NumberFormat:

package myPackage;

import java.math.BigDecimal;

import java.text.NumberFormat;

import java.text.ParseException;

import java.util.Locale;

public class Main {

public static void main(String[] args) {

String res=NumberFormat.getInstance().format(1000\_000\_000f);

String res1=NumberFormat.getInstance().format(8000);

String res2=NumberFormat.getInstance(Locale.ENGLISH).format(new BigDecimal("370000.45"));

System.out.println("Число по-российскому формату: "+res);

System.out.println("Число по-корейскому формату: "+res1);

System.out.println("Число по-английскому формату: "+res2);

try {

Number val=NumberFormat.getInstance().parse(res);

System.out.println("Отпарсированное res: "+val);

Number val1=NumberFormat.getInstance().parse(res1);

System.out.println("Отпарсированное res1: "+val1);

Number val2=NumberFormat.getInstance().parse(res2);

System.out.println("Отпарсированное res2: "+val2);

} catch (ParseException e) {

System.out.println(e);

}

}

}

Класс DecimalFormat

У него определяется два открытых конструктора. В первый передают String *шаблон*, а во второй еще передают объект DecimalFormatSymbolic. Конструкторы:

DecimalFormat (String *шаблон*)

DecimalFormat (String *шаблон*, DecimalFormatSymbolic)

Также можно получить экземпляр методом getInstance(), обеими формами метода. В общем он схож с NumberFormat.

Ниже представлен пример его использования:

package myPackage;

import java.text.DecimalFormat;

import java.text.ParseException;

import java.util.Locale;

public class Main {

public static void main(String[] args) {

DecimalFormat format=new DecimalFormat("00.0");

String res = format.format(10.4);

String res1= DecimalFormat.getInstance(Locale.ENGLISH).format(10.4);

System.out.println("СНГ: "+res+" | Великобритания: "+res1);

try {

Number val= DecimalFormat.getInstance().parse(res);

Number val1= DecimalFormat.getInstance(Locale.ENGLISH).parse(res1);

System.out.println(val+" | "+val1);

} catch (ParseException e) {

System.out.println(e);

}

}

}

Пакет java.time

В JDK 8 появились новые средства для работы с даты и времен. Все они находятся в пакете java.time. В отличие от старых версий в новом API все они предлагают новый способ обработки и анализа дат.

Классы LocalDate, LocalTime, LocalDateTime.

Пакета java.time определяется несколько классов верхнего уровня для работы с классами. Класс LocalDate инкапсулирует информацию о дате, класс LocalTime инкапсулирует информацию о дате, а LocalDateTime инкапсулирует информацию о дате и времени.

В данных классах открытых конструкторов отсутствуют, но для их создания служит фабричный метод now(). Полное объявление метода для каждого класса:

static LocalDate now()

static LocalTime now()

static LocalDateTime now()

Каждый метод возвращает вариант объекта. Он возвращает объект, который легко отображается методом println(). В приведенное ниже программе демонстрируется применение метода now():

package myPackage;

import java.time.LocalDate;

import java.time.LocalDateTime;

import java.time.LocalTime;

public class Main {

public static void main(String[] args) {

LocalDate ld=LocalDate.now();

LocalTime lt=LocalTime.now();

LocalDateTime ldt=LocalDateTime.now();

System.out.println(ld);

System.out.println(lt);

System.out.println(ldt);

}

}

Также у каждого класса определяется ряд геттеров. Ниже представлен пример их использования:

package myPackage;

import java.time.LocalDateTime;

public class Main {

public static void main(String[] args) {

LocalDateTime ldt=LocalDateTime.now();

System.out.println(ldt);

System.out.println("Минуты: "+ldt.getMinute());

System.out.println("День: "+ldt.getMonth());

System.out.println("Год: "+ldt.getYear());

System.out.println("Час: "+ldt.getHour());

System.out.println("День недели: "+ldt.getDayOfWeek());

}

}

Форматирование даты и времени.

Класс DateTimeFormatter доставляет необходимые форматы для времени и даты. Он входит в состав пакета java.time.format, и для получение его следует воспользоваться фабричными методами:

static DateTimeFormatter ofLocalizedDate (FormatStyle *формат\_даты*)

static DateTimeFormatter ofLocalizedTime (FormatStyle *формат\_времени*)

static DateTimeFormatter ofLocalizedDateTime (

FormatStyle *формат\_даты*, FormatStyle *форма\_времени*)

Перечисление FormatStyle входит в пакет java.text.format и содержит следующие константы:

* FULL
* LONG
* SHORT
* MEDIUM

Для форматирования дат и времени в классах LocalDate, LocalTime и LocalDateTime определяется метод format(). Ему в параметры передает объект DateTimeFormatter. Полное объявление метода:

String format (DateTimeFormatter *формат*)

Также есть возможность задать свой формат для даты и времени. Для этого можно вызвать фабричный метод ofPattern() у DateTimeFormatter. Полное объявление:

static DateTimeFormatter ofPattern (String *шаблон*)

Для *шаблона* предназначаются ряд букв. Ниже перечисляются избранные буквы:

a – Обозначает полдень

d – День месяца

E – День недели

h – Час в 12-часовом формате

H – Час в 24 часовом формате

M - Месяц

m - Минуты

s - Секунды

y - Год

Ниже представлен пример использования format():

package myPackage;

import java.time.LocalDate;

import java.time.LocalDateTime;

import java.time.LocalTime;

import java.time.format.DateTimeFormatter;

import java.time.format.FormatStyle;

import java.util.Locale;

public class Main {

public static void main(String[] args) {

LocalDate ld=LocalDate.now();

System.out.println(ld.format(

DateTimeFormatter.ofLocalizedDate(

FormatStyle.FULL)));

LocalTime lt=LocalTime.now();

System.out.println(lt.format(

DateTimeFormatter.ofLocalizedTime(

FormatStyle.SHORT)));

LocalDateTime ldt=LocalDateTime.now();

System.out.println(ldt.format(

DateTimeFormatter.ofPattern(

“MMMM d’,’ yyyy h’:’mm a”,

Locale.ENGLISH)));

}

}

Синтаксический анализ текста даты и времени.

В классах LocalDate, LocalTime и LocalDateTime определяются средства для анализа текста даты и времени. Для этого предназначен метод parse(), который следует вызвать у экземпляра выше перечисленных классов. У него определенно два формы объявления. В первой форме в параметры передается обычная строка, к которой применятся шаблон по умолчанию – формат ISO:

static LocalDateTime parse (CharSequence *текст*)

static LocalDate parse (CharSequence *текст*)

static LocalTime parse (CharSequence *текст*)

Во второй форме метода parse() в нем также можно указать и шаблон, по которому будет проводится анализ:

static LocalDateTime parse (CharSequence *текст*, DateTimeFormatter *шаблон*)

static LocalDate parse (CharSequence *текст*, DateTimeFormatter *шаблон*)

static LocalTime parse (CharSequence *текст*, DateTimeFormatter *шаблон*)

Ниже представлен пример использования анализа текста:

package myPackage;

import java.time.LocalDateTime;

import java.time.format.DateTimeFormatter;

import java.util.Locale;

public class Main {

public static void main(String[] args) {

DateTimeFormatter formatter = DateTimeFormatter.ofPattern("MMMM d, yyyy hh:mm a", Locale.ENGLISH);

LocalDateTime ldt=LocalDateTime.parse(

"June 21, 2014 11:12 AM", formatter);

System.out.println(ldt.format(formatter));

System.out.println("Минуты: "+ldt.getMinute());

System.out.println("День: "+ldt.getMonth());

System.out.println("Год: "+ldt.getYear());

System.out.println("Час: "+ldt.getHour());

System.out.println("День недели: "+ldt.getDayOfWeek());

}

}

Единицы времени и дат.

Перечисление DayOfWeek.

Перечисление DayOfWeek содержит семь констант, т.е. дней недели. Все они номеруются с понедельника по воскресенья.

Методом getDisplayName() можно получить название недели с указанной *локацией* и *стилем*. Полное объявление метода:

String getDisplayName(TextStyle *стиль*, Locale *локация*)

Методами minus() и plus() можно узнать, какой день недели будет после и было до указанной *даты*. Полные объявления методов:

DayOfWeek minus(long *день*)

DayOfWeek plus(long *день*)

Метод getValue() возвращает номер дня недели. Полное объявление метода:

int getValue()

Метод values() можно получить массив с константами DayOfWeek. Полное объявление метода:

static DayOfWeek [] values()

Ниже представлен пример работы с DayOfWeek:

package myPackage;

import java.time.DayOfWeek;

import java.time.format.TextStyle;

import java.util.Locale;

public class Main {

public static void main(String[] args) {

DayOfWeek day=DayOfWeek.FRIDAY;

Locale thisLocal=Locale.ENGLISH;

System.out.println(day.getDisplayName(TextStyle.FULL, thisLocal));

System.out.println(day.getDisplayName(TextStyle.NARROW, thisLocal));

System.out.println(day.getDisplayName(TextStyle.SHORT, thisLocal));

System.out.println("День недели 8 дней назад: "+day.minus(8));

System.out.println("День недели после 8 дней: "+day.plus(8));

System.out.println("Номер дня недели: "+day.getValue());

DayOfWeek [] weeks=DayOfWeek.values();

for (DayOfWeek d : weeks) {

System.out.println(d);

}

}

}

Перечисление Month

Перечисление Month содержит 12 констант, т.е. месяцев. Все они номеруются с января (1) по декабрь (12).

Методом getDisplayName() можно получить название месяца с указанной *локацией* и *стилем*. Полное объявление метода:

String getDisplayName(TextStyle *стиль*, Locale *локация*)

Методами maxLength() и minLength() можно узнать максимальное и минимальное количество дней в месяце. Полные объявления методов:

int maxLength()

int minLength()

Методами minus() и plus() можно узнать месяц, который будет до или после указанного количества *месяцев*. Полные объявления методов:

Month minus(long *месяц*)

Month plus(long *месяц*)

Метод getValue() возвращает номер месяца. Полное объявление метода:

int getValue()

Метод values() можно получить массив с константами Month. Полное объявление метода:

static Month [] values()

Ниже представлен пример использования Month:

package myPackage;

import java.time.Month;

import java.time.format.TextStyle;

import java.util.Locale;

public class Main {

public static void main(String[] args) {

Month may=Month.MAY;

Locale thisLocal=Locale.ENGLISH;

System.out.println(may.getDisplayName(TextStyle.FULL, thisLocal));

System.out.println("Максимальное количество дней в мае: "+may.maxLength());

System.out.println("Минимальное количество дней в мае: "+may.minLength());

System.out.println("Месяц года 13 месяц назад: "+may.minus(13));

System.out.println("Месяц года после 13 месяцев: "+may.plus(13));

System.out.println("\n"+"Месяца: ");

Month [] months=Month.values();

for (Month m : months) {

System.out.println(m);

}

}

}

Класс Year.

Класс Year хранит год. Для того чтоб получить объект класса Year следует вызвать метод now() или методом of(), которому надо передать год.

Методом isLeap() можно узнать является год ли високосным. У него определенно две формы объявления. Первый вызывается у экземпляра, а второй статический, которому передают год. Полные объявления методов:

boolean isLeap()

static boolean isLeap (Year *год*)

Метод isAfter() проверяет, прошел ли этот год после указанного *года*, а методом isBefore() сделать наоборот. Полные объявления:

boolean isAfter (Year *год*)

boolean isBefore (Year *год*)

Ниже представлен пример работы с классом Year:

package myPackage;

import java.time.Year;

import java.util.Arrays;

public class Main {

public static void main(String[] args) {

Year thisYear=Year.now();

System.out.println("Этот год високосный: "+thisYear.isLeap());

System.out.println("Этот год был после 1998: "+thisYear.isAfter(Year.of(1998)));

System.out.println("Этот год был до начала болотной революции(1911): "+

thisYear.isBefore(Year.of(2011)));

System.out.println("Количество дней в этом году: "+thisYear.length());

}

}

Класс MonthDay

Класс MonthDay инкапсулирует месяц и дату. Для получения ее можно вызвать фабричный метод of() и передать месяц и день или получить сегодняшний месяц и дату методом now(). Методом with() можно получить объект с указанным месяцем или датой. Ниже представлен пример работы с MonthDay:

package myPackage;

import java.time.Month;

import java.time.MonthDay;

public class Main {

public static void main(String[] args) {

MonthDay day=MonthDay.of(Month.MAY, 30);

System.out.printf("%s, %d %n", day.getMonth(), day.getDayOfMonth());

System.out.println("Эта дата корректна для 2014 года: "+day.isValidYear(2014));

MonthDay day1=day.with(Month.FEBRUARY);

System.out.printf("$s, $d $n");

MonthDay day2=day.withDayOfMonth(12);

}

}

Моменты и периоды.

Класс Instant.

Класс Instant служит альтернативный классом Date в пакете java.time. Он содержит наносекунды с 1 января 1970 года, но также может содержать отрицательные числа, т.е. время до 1 января 1970.

Чтобы получить сегодня следует вызвать статический метод now(). Полное объявление метода:

static Instance now()

Методом plusSecond() и minusSecond() и им подобные методы позволяет отнять или прибавить секунды, миллисекунды и наносекунды.

Методом getNano() можно получить текущее время в виде наносекунд, а методом getEpochSecond() миллисекунды с 1970.

Методами isAfter() или isBefore() можно узнать была ли вызывающая дата раньше или позднее *указанной даты*.

Работать с объектом Instant трудно и поэтому ее принято привязывать к LocalDateTime. Для этого следует вызвать статический метод ofInstant() и передать ему конкретный *объект* Instant и *зону*. Полное объявление метода:

static LocalDateTime ofInstant( Instant *момент*, ZoneId *зона*)

Ниже представлен пример использования данного класса:

package myPackage;

import java.time.Instant;

import java.time.LocalDateTime;

import java.time.ZoneId;

import java.time.format.DateTimeFormatter;

public class Main {

public static void main(String[] args) {

Instant instant=Instant.now();

System.out.println(“Изначальное время: “+instant);

Instant instant1= instant.plusSeconds(12467);

System.out.println(“Время со вмещением вперед 12467 секунд: “+instant1);

Instant instant2Minus= instant.minusNanos(12350);

System.out.println(“Время с вмещением назад 12350 наносекунд: “+instant2Minus);

System.out.println(“instance был раньше instance1: “+instant.isAfter(instant1));

System.out.println(«Время в наносекундах с 1970: «+instant.getEpochSecond());

LocalDateTime ldt=LocalDateTime.ofInstant(instant2Minus, ZoneId.systemDefault());

System.out.println(«Отформатированное время: «+

ldt.format(DateTimeFormatter.ofPattern(“HH:mm:ss”)));

}

}

Класс Period.

Класс Period инкапсулирует протяженность даты. Класс Period определяет период в понятном для человека образе: год, месяц и день месяца.

Чтобы получить период можно вызвать статический метод between(), в параметры передают *стартовую* и *конечную* дату, после вычитается время периода. Полное объявление метода:

static Period between (LocalDate *начало*, LocalDate *конец*)

В классе Period определяется ряд геттеров: getDays(), getMounts() и getYears() . Также чтоб получить количество всех дней, годов и месяцев можно вызвать метод between() у перечисления ChronoUnit.

Методом isNegative() можно узнать есть ли отрицательные числа. Методом isZero() можно узнать, равен ли период нулю.

Ниже представлен пример использования данного класса:

package myPackage;

import java.time.LocalDate;

import java.time.Month;

import java.time.Period;

import java.time.temporal.ChronoUnit;

public class Main {

public static void main(String[] args) {

LocalDate birthday=LocalDate.of(2005, Month.FEBRUARY, 5);

Period p=Period.between(birthday, LocalDate.now());

long days= ChronoUnit.DAYS.between(birthday, LocalDate.now());

System.out.println(“Возраст: “+ p.getYears()+” лет, “+

p.getMonths()+» месяца и «+

p.getDays()+» дней»+». Всего «+days+» дней»);

System.out.println(«Есть ли в периоде отрицательные числа: «+p.isNegative());

System.out.println(«Все цифры периода равны: «+p.isZero());

}

}

Класс Duration.

Класс Duration инкапсулирует протяженность времени. Класс Duration определяет период в понятном для нас виде: час, минуты, секунды, миллисекунды и наносекунды.

Чтоб его получить можно вызвать метод between(), передавая *начальную* дату и *стартовую* дату, а после вычитывает период. Еще есть ofSeconds(), ofHouses() и им подобные методы задает сам период.

Геттерами можно вернуть ту или иную единицу времени, а методами toNanos() и т.д. можно узнать все время в наносекундах и т.д.

Ниже представлен пример использования Duration:

package myPackage;

import java.time.Duration;

import java.time.LocalTime;

public class Main {

public static void main(String[] args) {

Duration d=Duration.ofSeconds(12, 23);

System.out.println(“Секунды: “+d.getSeconds());

System.out.println(“Наносекунды: “+d.toNanos());

Duration dur=Duration.between(

LocalTime.of(1, 3, 5),

LocalTime.of(2, 3, 5));

System.out.printf(«Час %d, минуты %d, секунды %d, миллисекунды %d, наносекунды %d»,

dur.toHours(),

dur.toMinutes(),

dur.getSeconds(),

dur.toMillis(),

dur.toNanos());

}

}

Библиотеки от сторонних разработчиков.

Java SE и Java EE предлагают многие средства и библиотек, но все равно их не хватает в многих целях. Для этого многие разработчики создают свои библиотеки и выкладывают в сеть для других разработчиков.

Встает вопрос как их использовать. Существует два способа. Первый заключается самостоятельно скачать фреймворк и вставить в ее проект. Это довольно трудно ведь даже jar файл вставить в другой jar займет время, когда никто не отменил других файлов.

Второй и более практичный способ заключается в использования *сборщиках проекта*. Они представляют удобный способ использовании. При использовании сборщика следует указать только, какая библиотека будет использоваться она установится и настроена.

Чтобы узнать какие библиотеки есть программе следует просто в ExternalLibraries и кроме стандартных библиотек можно найти и скаченные библиотеки.

Сборщик проектов – Maven.

Maven – это фреймворк автоматизированной сборки проектов в Java. Кроме него есть множество сборщиков проектов такие как Garble и прочие, но Maven является самым популярным сборщиком проектов.

Кроме сборки проектов он управляет зависимостями внутри проекта. Также он позволяет сгенерировать каркас проекта.

Maven проект

У любого проекта созданного с Maven должна быть указана *группа объекта* и *артефакт* группы. Сам же Maven представляет собой файл pom.xml, в который указываются группа и артефакт.

Кроме pom.xml Maven проект содержит файл для тестов JUnit – фреймворк для теста программ. Следует сказать, что из всего xml используется в основном парный тег dependencies.

У любого стороннего фреймворка / библиотеки на сайте разработчика есть xml описание библиотеки для Maven. Именно его следует вставить в выше описанный тег. После времени Maven скачает библиотеку.

Преимущества.

Многие IDEA также могут собирать проекты. Но сравнивать Maven и среды разработки некорректно так как Maven можно свободно использовать.

К минусам IDEA можно отнести то, что разные среды используют разные кодировки, версии языка и библиотек. Также чтоб пользователь приложения мог им пользоваться у него должны быть скачены библиотеки, когда Maven сам скачает и установит.

В общем все плюсы Maven перечислены ниже:

* Можно указать версию библиотеки
* Устоявшая структура, которая позволяет избежать путаницы.
* Набор шагов - компиляция, тестирование, упаковка и т.д.
* Кастомизации процесса сборки - добавления дополнительных шагов.
* Сборка не зависит от IDE, операционной системы и прочие, т.к. можно указать версию компилятора, кодировку и пр.

Pom

POM — эта общая, унифицированная модель описания программных модулей. Она поддерживает хранение не только атрибутов отдельных модулей, но и высокоуровневых связей между ними.

Также она несёт информацию об инструментах, необходимых для поддержки жизненного цикла модуля.

Репозитории артефактов

*Репозитории* *артефактов* являются выделенными хранилищами результатов сборки, устроенными так, чтобы упростить поиск нужного артефакта (по имени, версии, типу артефакта).

Они позволяют группе разработчиков пользоваться результатами работы друг друга без необходимости иметь копии исходных кодов их модулей и выполнять сборку дерева зависимостей с 0.

Maven-совместимые репозитарии стали стандартом де факто публикации результатов различных открытых проектов.

Нужно отметить pom файлы также являются артефактами и могут хранится в репозиториях как и другие. В итоге репозитории является еще источником информации о информации.

Утилита управления жизненным циклом.

*Жизненный цикл* называется совокупность операций от инициализации сборки и развертывания. В процессе жизненного цикла формируются некоторые артефакты

*Утилита* отвечает за реализацию жизненного цикла на основе модели проекта. Сама утилита состоит двух частей:

* Маленькое ядро содержит базовые принципы устройства проектов, цели для фаз программы и рабочая среда для плагинов.
* Плагины, которые содержат код и отвечают за генерацию, компиляцию и т.д.

Пример Maven

Ниже представлен пример pom файла и скачивания библиотеки Jsoup, которая предоставляет средства для парсинга сайтов т.е. получение ее HTML файла.

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>Example\_Maven</groupId>

<artifactId>Examples</artifactId>

<version>1.0-SNAPSHOT</version>

<dependencies>

<!-- https://mvnrepository.com/artifact/org.jsoup/jsoup -->

<dependency>

<groupId>org.jsoup</groupId>

<artifactId>jsoup</artifactId>

<version>1.9.1</version>

</dependency>

</dependencies>

</project>

Здесь из кода написано лишь тег <dependencies>, которому мы передали описание библиотеки jsoup на их сайте. Теперь можно свободно использовать средства загруженной библиотеки.



Тесты с JUnit 5.

Самая популярная библиотека для теста программ это JUnit. Она позволяет автоматизировать тестирование программ. Действующая версия JUnit 5. Описание для Maven:

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-api</artifactId>

<version>RELEASE</version>

</dependency>

*JUnit тест с технической стороны* – это класс который лежит в тестовом ресурсе и который предназначен только для тестирование логики, а не для использования в production коде.

Все тестируемые методы предполагают то, что они могут тестироваться по отдельности и не зависят от друг друга.

Использование JUnit в IntelliJ IDEA.

Если в IntelliJ IDEA выбрать Maven Project то обязательно появится папка test, сама же иконка будет зеленого цвета. Но его можно полноценно использовать, когда ввести JUnit.

Чтобы создать тест для класса нужно в объявлении класса нажать Alt + Enter. После следует выбрать «Create Test». Здесь будут варианты библиотек для теста и следует выбрать JUnit 5.

Также там еще можно указать имя класса с тестами, но лучше сохранить имя по умолчанию; суперкласс и пакет. В Member можно выбрать какие методы будут тестироваться. Еще есть вторичная Generate, но заполнять ее не обязательно.

Аннотация @Test.

Ключевой вещью в JUnit является аннотация @Test. В 5 версии она стала маркером – аннотацией. Метод, который будет аннотирован @Test будет тестироваться.

@RepeatedTest позволяет протестировать метод *n*-количество раз. @Disable указывает на то, что метод не следует тестировать, есть необязательное поле value, которое содержит причину отказа от тестирования.

Если метод аннотирован @BeforeAll тогда он будет выполнен раньше всех тестов. Схожим является @BeforeEach, за исключением того, что он выполняется перед каждым тестом. Также есть @AfterAll и @AfterEach.

Также класс, который будет содержать тесты может иметь внутренние классы, которые также используются тестирования. Для создания таких внутренних классов следует аннотировать его @Nested.

Аннотация @DisplayName обычно используются для тестирования отчетов в среде IDE и создания инструментов и могут содержать пробелы, специальные символы и даже эмоджи. Его значение value обозначает пользовательское отображаемое имя.

Класс Assertions.

Данный утилит уже прямо относится к тестам. Он определяет методы типа assertEquals (), assertNotNull() и им подобные. Все методы void и если хоть один из методов данного класса провалится в тесте, тогда JUnit выведет ошибку.

Такие методы, как assertFalse () и assertTrue () проверяет *значение* на ложность или истинность. Также есть еще версия, которые позволяют передавать *сообщение* (либо Supplier <String>), которое будет передано в противном случае исключению.

Метод assertEquals () сравнивает *фактический* и *предполагаемый* Object либо другой примитив. Еще можно передать *сообщение*. Схожими является assertEqualsArray () и assertIterableEquals ().Схожим assertEquals () является assertSame (), который будет сравнивать по ссылке.

Для проверки того, что объект пуст (null) используется метод assertNull (). Ему следует передать объект, который будет проверятся, дополнительно можно передать *сообщение*. Обратным ему является assertNotNull ().

Пример теста.

Пускай у нас имеется простой класс, который нужно протестировать:

public class Calculator {

int add (int val, int val1) {

return val + val1;

}

boolean isNegative (int number) {

if(number < 0)

return true;

return false;

}

double sqrt (double number) {

return Math.sqrt(number);

}

}

А вот так может выглядеть его тест:

import org.junit.jupiter.api.\*;

import static org.junit.jupiter.api.Assertions.\*;

class CalculatorTest {

@BeforeEach

void beforeEach(){

System.out.println("\nПеред каждым тестов.");

}

@AfterEach

void afterEach(){

System.out.println("После каждого теста.\n");

}

@Test

void add() {

Calculator c = new Calculator();

int result = c.add(12, 12);

assertEquals(24, result, "12 + 12 должно быть 24");

}

@Test

void isNegative() {

Calculator c = new Calculator();

boolean negative = c.isNegative(12);

assertFalse(negative, "Число положительно");

}

@Test

void sqrt() {

Calculator c = new Calculator();

double result = c.sqrt(13.4);

assertNotNull(result);

}

}

Также можно применить и внутренний класс:

@Nested

class InternalClass{

@BeforeEach

void beforeEach(){

System.out.println("\nПеред каждым тестов.");

}

@AfterEach

void afterEach(){

System.out.println("После каждого теста.\n");

}

@Test

void add() {

Calculator c = new Calculator();

int result = c.add(12, 12);

assertEquals(24, result, "12 + 12 должно быть 24");

}

@Test

void isNegative() {

Calculator c = new Calculator();

boolean negative = c.isNegative(12);

assertFalse(negative, "Число положительно");

}

@Test

void sqrt() {

Calculator c = new Calculator();

double result = c.sqrt(13.4);

assertNotNull(result);

}

}

Работа с исключениями.

Определяется два метода для работы с исключениями. Это assertDoesNotThrow (), который проверяет не будет ли сгенерировано исключение и обратное ему assertThrows().

Метод assertThrows () проверяет будет ли *исключение* генерироваться в *коде*. Он имеет следующие формы:

static <T extends Throwable> T assertThrows​(Class<T> *исключение*, ThrowingSupplier<T> *код*, String *сообщение*)

static <T extends Throwable> T assertThrows​(Class<T> *исключение*, ThrowingSupplier<T> *код*)

Метод assertDoesNotThrow () проверяет не будет ли генерироваться исключение в *коде*. Он имеет следующие формы:

static <T> T assertDoesNotThrow​(ThrowingSupplier<T> *код*)

static <T> T assertDoesNotThrow​(ThrowingSupplier<T> *код*, String *сообщение*)

Пример использования выше описанных методов:

@Test

void test(){

String message = "что то пошло не так";

assertDoesNotThrow(() -> { }, message);

assertThrows(

IllegalArgumentException.class, () -> {

throw new IllegalArgumentException(message);});

}

Тестирование интерфейсов.

JUnit также предлагает средства для тестирования интерфейса. Можно тестировать методы по умолчанию, статические методы и обычные.

К, примеру есть такой интерфейс:

public interface Interface {

default void method1() {

System.out.println("from method1 ()");

}

default void method2() {

System.out.println("from method2 ()");

}

void method();

static void staticMethod(){

System.out.println("from staticMethod ()");

}

}

 Библиотека Guava

Guava – это библиотека с свободным исходным кодом, разработанная Google. Она облегчает программирование на Java и существенно уменьшает количество ошибок.

В ней заложены: утилиты коллекций, хеширования, поддержка примитивов, работа с символьными строками, ввод/вывод, параллелизм, аннотации и валидации.

К преимуществам можно отнести следующие:

* Стандартизованная библиотека.
* Эффективное расширение Java.
* Оптимизация.
* Множество полезных классов.
* Добавляет функциональные возможности.
* Проверка.

Описание для Maven:

<dependency>

<groupId>com.google.guava</groupId>

<artifactId>guava</artifactId>

<version>25.1-jre</version>

</dependency>

Класс Optional.

Абстрактный класс Optional является аналогом из java.util.Optional, за исключением того, что он добавляет ряд возможностей и удобных методом. Он обобщен и где T обозначает тип содержимого.

Данный класс расширяет Object и реализует интерфейс Serializable. Его можно найти по пути com.google.common.base.Optional.

|  |  |
| --- | --- |
| Объявление | Описание |
| static <T> Optional <T> absent () | Возвращает пустой Optional. |
| abstract Set<T> asSet () | Возвращает множество с содержимым Optional. |
| abstract boolean equals ( Object *объект*) | Сравнивает вызывающий объект с *объектом*. |
| static <T> Optional<T> fromNullable (T *значение*) | Возвращает пустой Optional если *значение* равно null и наоборот. |
| abstract T get () | Возвращает значение. |
| abstract int hashCode () | Возвращает хэш-код объекта. |
| abstract boolean isPresent () | Вернет true если Optional не пуст. |
| static <T> Optional<T> of(T *значение*) | Возвращает Optional с *значением*. |
| abstract Optional <T> or(Optional <? extends T> *заменяемый*) | Если вызывающий Optional пуст возвращает *заменяемый* , а если вызывающий не пуст возвратится вызывающий. |
| abstract T or(Supplier <? extend T *возвратитель*) | Если вызывающий Optional пуст возвращает *возвратитель*.get (), а если вызывающий не пуст возвратится вызывающий. |
| abstract T or(T *значение*) | Если вызывающий Optional пуст возвращает *значение*, а если вызывающий не пуст возвратится вызывающий. |
| abstract T orNull() | Возвращает содержащийся экземпляр, если он присутствует; null в противном случае. |
| static <T> Iterable <T>  presentInstances (Iterable <? extends Optional <? extends T >> *итератор*) | Возвращает Iterator. |
| abstract String toString () | Возвращает строковое представление Optional |
| abstract <V> Optional<V> transform(Function<? super T,V> *функция*) | Если экземпляр присутствует, он преобразуется с заданной функцией; в противном случае возвращается absent (). |
| abstract java.util.Optional toJavaUtil () | Возвращает классический Optional. |

Ниже представлен пример использования выше описанного Optional. Здесь также используется Function из Guava.

package Example;

import com.google.common.base.\*;

public class GOptionalExample {

public static void main(String[] args) {

Optional<String> optional = Optional.of("Guava Optional");

System.out.println("Содержимое optional: "+optional.get());

Optional<String> optional1 = Optional.fromNullable("Guava Optional.fromNullable");

if(optional1.isPresent())

System.out.println("Содержимое optional1: "+optional1.get());

else

System.out.println("optional1 пуст");

System.out.println("Они равны: "+optional.equals(optional1));

Function<String, String> function = s -> s.toUpperCase();

optional = optional.transform(function);

System.out.println("Новое значение optional: "+optional.get());

}

}

Класс Ordering.

Абстрактный класс Ordering можно рассматривать как обогащенный компаратор. Он расширяет Object и реализует Comparator. Он является обобщенным и где T обозначает тип сортируемых элементов. Ниже представлены методы данного класса:

|  |  |
| --- | --- |
| Объявление | Описание |
| static Ordering <Object> allEqual () | Возвращает пустой компаратор. |
| static Ordering <Object> arbitrary () | Возвращает компаратор, который сортирует в произвольном порядке. |
| abstract int compare (T *левый*, T *правый*) | Сравнивает *левый* и *правый*. |
| int binarySearch (List <? extends T> *список*, T *значение*) | Ищет индекс *значения* в *списке*. Если индекс не находится возвращается -1. |
| static <T> Ordering<T> from(Comparator<T> *компаратор*) | Создает Ordering основанный на *компараторе*. |
| static <C extends Comparable> Ordering<C> natural() | Возвращает Ordering с натуральным порядком. |
| <S extends T> Ordering<S> reverse() | Возвращает Ordering с обратным порядком. |
| static Ordering <Object> usingToString() | Сравнивает строковые представления элементов, возвращаемые метод toString () по натуральному порядку. |
| boolean isOrdered (Iterable <? extend T> *итератор*) | Проверяет отсортирован ли заданный *итератор*. |
| <E extends T> E min (Iterator *итератор*) | Возвращает минимальный элемент *итератора*. |
| <E extends T> E max (Iterator *итератор*) | Возвращает максимальный элемента *итератора*. |
| <S extends T> Ordering <S> nullsFirst () | Возвращает Ordering, который сравнивает пустые значения как меньшие. |
| <S extends T> Ordering <S> nullsLast () | Возвращает Ordering, который сравнивает пустые значение как большие. |
| <F> Ordering<F> onResultOf(Function<F,? extends T> *функция*) | Сначала применяет *функцию*, а после сортирует. |
| <E extends T> List<E> sortedCopy(Iterable<E> *итератор*) | Создает копию сортируемой коллекции |
| <E extends T> List <E> lessOf (Iterable <E> *итерация*, int *количество*) | Возвращает *количество* меньших элементов *итерация*. |

Ниже представлен пример сортировки компаратором Ordering:

package Example;

import com.google.common.collect.\*;

import java.util.\*;

public class GOrderingExample {

public static void main(String[] args) {

ArrayList<String> list = new ArrayList<>();

list.add("Guava");

list.add("Jsoup");

list.add("Maven");

list.add("Apache Commons");

list.add("google-gson");

list.add("Mockito");

System.out.println("Исходный список: "+list);

Ordering<String> ordering = Ordering.natural();

List<String> list1 = ordering.leastOf(list, 2);

System.out.println("Два наименьших элементов: "+list1);

Collections.sort(list, ordering);

System.out.println("Натуральный список: "+list);

Collections.sort(list, Ordering.arbitrary());

System.out.println("Случайный порядок: "+list);

System.out.println("\n========================");

System.out.println("Коллекция отсортирована: "+ordering.isOrdered(list));

System.out.println("Минимум: "+ordering.min(list));

System.out.println("Максимум: "+ordering.max(list));

list.add(null);

Collections.sort(list, ordering.nullsFirst());

System.out.println("\nПустые значение как меньшие: "+list);

}

}

Класс Preconditions.

Класс Preconditions предоставляет статические методы для проверки аргументов метода или конструктора. Все его методы генерируют IllegalArgumentException в сбое. Ниже представлены методы данного класса:

|  |  |
| --- | --- |
| Объявление. | Описание. |
| static void checkArgument (boolean *выражение*) | Если *выражение* ложно кинется исключение IllegalArgumentException. |
| static void checkArgument (boolean *выражение*, Object *сообщение* ) | Если *выражение* ложно кинется исключение IllegalArgumentException с *сообщением*. |
| static <T> T checkNotNull (T *объект*, String *сообщение*, Object … *аргументы*) | Если *объект* пуст кинется исключение IllegalArgumentException с переданному ему *аргументами* и *сообщение*. |
| static <T> T checkNotNull (T *объект*) | Если *объект* пуст кинется исключение IllegalArgumentException. |
| static <T> T checkNotNull (T *объект*, Object *сообщение*) | Если *объект* пуст кинется исключение IllegalArgumentException с *сообщением*. |
| static int checkPositionIndex (int *индекс*, int *размер*) | Проверяет на выходит ли *индекс* за границы *размера* массива, строки и т.д. |
| static int checkPositionIndex (int *начало*, int *конец*, int *размер*) | Проверят ли не выходит ли часть с *начало* до *конца* за границы *размера* массива, строки и т.д. |

Ниже представлен пример использования выше описанного класса:

package Example;

import com.google.common.base.Preconditions;

public class GPreconditionsExample {

public static void main(String[] args) {

try {

sqrt(-23.56);

}catch (IllegalArgumentException exc){

System.out.println("Exception "+exc.getMessage());

}

try {

sum(null, 24);

}catch (NullPointerException exc){

System.out.println("Exception "+exc.getMessage());

}

try{

getValue(12);

}catch (IndexOutOfBoundsException exc){

System.out.println("Exception "+exc.getMessage());

}

}

static int sum (Integer one, Integer two){

one = Preconditions.checkNotNull(one,

"Illegal Argument passed: First parameter is Null.");

two = Preconditions.checkNotNull(two,

"Illegal Argument passed: Second parameter is Null.");

return one + two;

}

static int getValue (int input){

int [] array = {2, 3, 345, 23, 34, 35, 45};

Preconditions.checkPositionIndex(input, array.length,

"Illegal Argument passed: Invalid index.");

return 0;

}

static double sqrt (double value){

Preconditions.checkArgument(value > 0,

"Illegal Argument passed: Negative value %s.");

return Math.sqrt(value);

}

}

Класс Range.

Класс Range инкапсулирует интервал или последовательность. Он так или иначе относится к коллекция. Он является обобщенным <C extends Comparable>, где С тип элементов диапазона.

|  |  |
| --- | --- |
| Объявление | Описание |
| static <C extends Comparable<?>> Range<C> all() | Возвращает Range по умолчанию. |
| static <C extends Comparable<?>> Range<C> atLeast(C *начало*) | Возвращает диапазон, которые содержит большие или равные *началу*. |
| static <C extends Comparable<?>> Range<C> atMost(C *конец*) | Возвращает диапазон, который содержит меньшие или равные *конец*. |
| static <C extends Comparable<?>> Range<C> closed(C *начало*, C *конец*) | Возвращает диапазон, который содержит значения больше или равны *началу* и меньше или равны *конец*. |
| static <C extends Comparable<?>> Range<C> encloseAll(Iterable<C> *итератор*) | Возвращает диапазон, который содержит *итератор*. |
| static <C extends Comparable<?>> Range<C> open(C *начало*, C *конец*) | Возвращает диапазон, содержащий все значения, строго превышающие *начало*, чем *конец*. |
| static <C extends Comparable<?>> Range<C> openClosed(C *начало*, C *конец*) | Возвращает диапазон, содержащий все значения, строго превышающий *начало* или равный *конец*. |
| static <C extends Comparable<?>> Range<C> singleton(C *значение*) | Возвращает диапазон содержащий *значение*. |
| boolean contains (C *значение*) | Проверяет в наличие *значения*. |
| boolean containsAll (Iterable <? extends C> *значения*) | Проверят совпадают *значения* с диапазоном. |
| boolean hasLowerBound () | Проверяет диапазон на наличие более низкой начальной точки. |
| boolean hasUpperBound () | Проверяет диапазон на наличие более высокой конечной точки. |
| C lowerEndpoint() | Возвращает начало. |
| C upperEndpoint() | Возвращает конец. |
| boolean isEmpty () | Проверят диапазон на пустоту. |
| int hashCode () | Возвращает хэш-код. |

Ниже представлен пример создания и использования диапазона:

package Example;

import com.google.common.collect.\*;

public class GRangeExample {

public static void main(String[] args) {

Range<Integer> ri = Range.atLeast(23);

System.out.println("atLeast (): "+ri);

ri = Range.atMost(24);

System.out.println("atMost (): "+ri);

ri = Range.all();

System.out.println("all (): "+ri);

ri = Range.open(-12, 12);

System.out.print("open (): ");

printRange(ri);

System.out.println("\nНачало: "+ri.lowerEndpoint());

System.out.println("Конец: "+ri.upperEndpoint());

System.out.println("\nhasLowerBound () -> "+ri.hasLowerBound());

System.out.println("hasUpperBound () -> "+ri.hasUpperBound());

}

static void printRange (Range<Integer> range){

System.out.print("[ ");

for(int grade : ContiguousSet.create(range, DiscreteDomain.integers())) {

System.out.print(grade +" ");

}

System.out.println("]");

}

}

Аннотации.

Гуава определяет ряд удобных и востребованных аннотаций. Все они представлены в пакете com.google.common.annotations.

@Beta

Retention = Class; Применим: аннотации, конструктор, класс, поле и метод.

Он указывает, что API может быть не совестим с изменениями и/или удалениями. Носитель освобождается от гарантий совместимости предоставляемой библиотекой. @Beta является маркером.

@ VisibleForTesting

Аннотация маркер VisibleForTesting указывает, что элемент программы может использоваться в тестировании и нигде кроме.

@ GwtCompatible

Аннотация @ GwtCompatible указывает, что носитель может использоваться в проекте Google Wed Toolkit (GWT) <http://www.gwtproject.org/?csw=1>. Применим: Type, Method. Поля:

* boolean emulated – указывает будет носитель эмулироваться в GWT. По умолчанию false.
* boolean [serializable](https://google.github.io/guava/releases/19.0/api/docs/com/google/common/annotations/GwtCompatible.html#serializable()) – указывает аннотированный тип или тип возвращаемого значения метода является сериализуемым GWT.

@ GwtIncompatible

Аннотация @GwtIncompatible является противоположным @ GwtCompatible. Он применим к полям, методам и внутренних классов. Поля:

* String value – Описает почему элемент не может использоваться в GWT.

String утилиты.

Библиотека Guava предоставляет множество классов для работы с строками основанные на опыте работы. Ниже представлены 4 основных:

* Joiner – Объединяет строки.
* Splitter – Разделяет строку.
* CharMatcher – Для операций со символами.
* CaseFormat – Для изменения строковых форматов.

Класс Joiner.

Как ранее говорилось класс Joiner отвечает за объединение строк. Он предоставляет ряд методов для склеивания строк. Ниже они перечислены:

|  |  |
| --- | --- |
| Объявление | Описание |
| <A extends Appendable> A appendTo (A *исходное*, Iterator <?> *итератор*) | Присоединяет к *исходной* все элементы *итератор*. |
| <A extends Appendable> A appendTo (A *исходное*, Object [] *массив*) | Присоединяет к *исходной* строковые представления элементов *массива*. |
| <A extends Appendable> A appendTo (A *исходное*, Object *первый*, Object *второй*, Object … *остальные*) | Присоединяет к *исходной* строковые представления *первого*, *второго* и *остальных* объектов. |
| StringBuilder appendTo (StringBuilder *строитель*, Iterator <?> *итератор*) | Использует *строитель* , присоединяя строковый представления элементов *итератора*, возвращаемые метод toString () |
| StringBuilder appendTo (StringBuilder *строитель*, Object [] *массив*) | Использует *строитель* , присоединяя строковый представления элементов *массив*, возвращаемые метод toString () |
| Joiner skipNulls () | Возвращает Joiner пропуская все null. |
| static Joiner on(char *символ\_разделитель*) | Возвращает Joiner, который соединяет строки через *символ\_разделитель*. |
| static Joiner on(String *разделитель*) | Возвращает Joiner, который соединяет строки через *разделитель*. |
| Joiner useForNull (String *строка*) | Возвращает Joiner, который все null заменяет на *строку*. |
| Joiner.MapJoiner withKeyValueSeparator (String *разделитель*) | Возвращает Joiner.MapJoiner присоединяя через *разделитель*. |

Здесь стоит отметить внутренний класс MapJoiner. Он работает схоже как и Joiner, за исключением то что он соединяет ключ – значение.

Ниже представлен пример использования Joiner:

import com.google.common.base.\*;

import java.util.ArrayList;

public class GJoinerExample {

public static void main(String[] args) {

StringBuilder builder = new StringBuilder();

ArrayList<String> list = new ArrayList<>();

list.add("Joiner");

list.add("Optional<T>");

list.add(null);

list.add("Ordering");

list.add(" Guava ");

Joiner join = Joiner.on(", ").useForNull("without null");

builder = join.appendTo(builder, list);

System.out.println("Результат appendTo(): "+builder);

String text = join.join(list);

System.out.println("Результат join(): "+text);

}

}

Класс Splitter

Как выше уже говорилось класс Splitter служит для разделения строки на части. Разделить можно по длине, символе и регулярному выражению. Splitter является final. Ниже представлены методы данного класса:

|  |  |
| --- | --- |
| Объявление | Описание |
| static Splitter fixedLength (int *длина*) | Возвращает Splitter разделяющий строку на части по *длине*. |
| Splitter limit (int *длина*) | Возвращает Splitter, у который может обработать строку с *длиной*. |
| Splitter omitEmptyStrings() | Возвращает Splitter, который пропускает пустые значения. |
| Splitter trimResult () | Применяет метод trim() к отдельной части. |
| static Splitter on (char *символ\_разделитель*) | Разделяет строку на части по *символу разделителю*. |
| static Splitter on (String *разделитель*) | Разделяет строку на части по *разделителю*. |
| static Splitter onMatcher (String *шаблон*) | Разделяет строку на части, который разделителем считается часть строки совпавшая с *шаблоном*. |
| Iterator <String> split (CharSequence *последовательность*) | Разделяет *последовательность* на части. |
| List <String> splitToList(CharSequence *последовательность*) | Разделяет *последовательность* на части в список. |
| Splitter.MapSplitter withKeyValueSeparator(Splitter *разделитель*) | Возвращает MapSplitter, который разделяет пары по *разделителю*. |
| Splitter.MapSplitter withKeyValueSeparator(String *разделитель*) | Возвращает MapSplitter, который разделяет пары по *разделителю*. |

В классе Splitter есть внутренний класс MapSplitter, у него определяются схожие методы, но для работы с парами. Ниже представлен пример Splitter:

import com.google.common.base.\*;

public class GSplitterExample {

public static void main(String[] args) {

Splitter split = Splitter.on(", ").trimResults().limit(200);

Iterable<String> split1 = split.split("1, 2, 3, 4, 5, 6, 7");

split1.forEach(str -> System.out.print(str+" "));

System.out.println();

Splitter splitLength = Splitter.fixedLength(4);

Iterable<String> res = splitLength.split("qwertyuioasdfghjklzxcvbnm");

res.forEach(s -> System.out.print(s+" "));

}

}

Перечисление CaseFormat.

Перечисление CaseFormat служит для преобразования форматов ASCII. Ниже перечислены константы:

|  |  |
| --- | --- |
| Объявление | Пример |
| LOWER\_CAMEL | lowerCamel |
| UPPER\_CAMEL | UpperCamel |
| LOWER\_HYPHEM | lower-hyphen |
| LOWER\_UNDERSCORE | lower\_underscore |
| UPPER\_UNDERSCORE | Upper\_Underscore |

Как можно догадаться LOWER\_CAMEL и UPPER\_CAMEL отвечают за формат без нижнего подчеркивания, а LOWER\_UNDERSCORE и UPPER\_UNDERSCORE наоборот. Ниже перечислены методы данного перечисления:

|  |  |
| --- | --- |
| Объявление | Описание |
| Converter <String, String> converterTo (CaseFormat *формат*) | Возвращает Converter, который преобразует строки в указанный *формат*. |
| String to(CaseFormat *формат*, String *строка*) | Изменят формат *строки* в *формат* |
| static CaseFormat valueOf (String *имя*) | Возвращает константу с указанным *именем* |
| static CaseFormat [] values () | Возвращает массив констант. |

Ниже представлен пример использования перечисления для изменения формата строки:

package Example;

import com.google.common.base.CaseFormat;

public class GCaseFormatExample {

public static void main(String[] args) {

String text;

text = CaseFormat.LOWER\_HYPHEN.to(CaseFormat.UPPER\_UNDERSCORE, "data-test");

System.out.println(text);

text = CaseFormat.LOWER\_CAMEL.to(CaseFormat.LOWER\_HYPHEN, "dataTest");

System.out.println(text);

text = CaseFormat.LOWER\_UNDERSCORE.to(CaseFormat.LOWER\_CAMEL, "data\_test");

System.out.println(text+"\n");

CaseFormat [] array = CaseFormat.values();

for(CaseFormat f : array) {

System.out.println(f.name()+", номер "+f.ordinal());

}

}

}

Примитив утилиты.

Так как примитивы нельзя передать в generic или в коллекцию входным параметром для этого Гуава предоставляет ряд оболочек примитивов. Ниже они перечислены:

* Bytes
* Shorts
* Ints
* Longs
* Doubles
* Chars
* Booleans

Класс Bytes.

Класс утилит Bytes служит оболочкой для примитивов типа byte. Его можно рассматривать как обогащенный аналог класса Byte. Все его методы static, а конструктор закрыт. Ниже представлены методы данного класса:

|  |  |
| --- | --- |
| Объявление | Описание |
| static List<Byte> asList (byte … *байты*) | Возвращает список *байтов*. |
| static byte [] concat (byte[] … *массивы*) | Соединяет все *массивы*. |
| static boolean contains (byte [] *массив*, byte *байт*) | Проверяет есть ли *байт* в *массиве*. |
| static byte [] ensureCapacity (byte [] *массив*, int *минимальная\_длина* , int *размер*) | Возвращает массив, содержащий те же значения, но с указанным объемом. |
| static int hashCode (byte *значение*) | Возвращает хэш-код *значения*. |
| static int indexOf (byte [] *массив*, byte *байт*) | Ищет первое вхождение *байта* в *массиве*. |
| static int indexOf (byte [] *массив* , byte [] *значения* ) | Ищет первое вхождение *значения* в *массив*, иначе -1. |
| static int lastIndexOf (byte [] *массив*, byte *байт*) | Ищет последнее вхождение *байта* в *массива*, иначе -1. |
| static byte [] toArray (Collections <? extends Number > *коллекция*) | Преобразуют *коллекцию* в массив. |
| static void reserve (byte *массив*) | Переворачивает *массив* |

Ниже представлен пример использования данного класса:

package Example;

import static com.google.common.primitives.Bytes.\*;

import java.util.\*;

public class GBytesExample {

public static void main(String[] args) {

byte [] array = {1, 24, 3, 6, 12, 24, 48, 96};

List<Byte> listOfByte = asList(array);

System.out.println("asList (): "+listOfByte);

array = toArray(listOfByte);

System.out.println("toArray (): "+Arrays.toString(array));

reverse(array);

System.out.println("reserve (): "+Arrays.toString(array));

System.out.println("\n==============================");

System.out.println("Первое вхождение 87: "+indexOf(array, (byte) 87));

System.out.println("Последнее вхождение 24: "+lastIndexOf(array, (byte) 24));

System.out.println("6 содержится в array: "+contains(array, (byte) 6));

}

}

Класс Shorts.

Данный утилитный класс служит оболочкой для примитивов типа short. Его можно рассматривать как обогащенный вариант класс Short. Все его методы static, а конструктор закрыт. Ниже представлены методы данного класса:

|  |  |
| --- | --- |
| Объявление | Описание |
| static List<Short> asList (char … *значения* ) | Возвращает List *значений*. |
| static short checkedCast (long *значение*) | Вернет short равная *значению* без потери(если это возможно). |
| static int compare (short *значение1*, short *значение2*) | Сравнивает оба значения. |
| static short [] concat (short []… *массивы*) | Соединяет *массивы*. |
| static boolean contains (short [] *массив*, short *значение*) | Проверяет на наличие *значения* в *массиве*. |
| static short [] ensureCapacity (short [] *массив*, int *минимальная\_длина*, int *длина*) | Возвращает массив с *длиной* и с теми же значениями *массива*. |
| static short fromBytesArray (byte [] *массив*) | Возвращает short, который равен содержимым *массива*. |
| static short fromBytes (byte *байт1*, byte *байт2*) | Эквивалент fromBytesArray (new byte [] {byte *байт1*, byte *байт2*}). |
| static int hashCode (short *значение*) | Возвращает хэш-код *значения*. |
| static int indexOf (short [] *массив*, short *значение*) | Ищет первое вхождение *значения* в *массиве*, иначе -1. |
| static int indexOf (short [] *массив*, short [] *значения*) | Ищет первое вхождение *значений* в *массиве*, иначе -1 |
| static String join (String *разделитель*, char… *значения*) | Возвращает строку содержащие все *значения* через *разделитель*. |
| static int lastIndexOf (short[] *массив*, short *значение*) | Ищет последнее вхождение *значения* в *массиве*, иначе -1. |
| static Comparator<short []> lexicographicalComparator() | Возвращает компаратор, который сравнивает два short лексикографически. |
| static short min (short … *массив*) | Ищет наименьший элемент. |
| static short max (short … *массив*) | Ищет наибольший элемент. |
| static short saturatedCast (long *значение*) | Возвращает наиболее близкий по *значению* тип short. |
| static short [] toArray (Collections <? extends Number> *коллекция*) | Преобразует *коллекцию* в массив short. |
| static void reserve (short [] *массив*) | Переворачивает *массив*. |
| static void sortDescending (short [] *массив*) | Перемешивает *массив*. |

Ниже представлен пример использования данного класса:

package Example;

import static com.google.common.primitives.Shorts.\*;

import java.util.\*;

public class GShortsExample {

public static void main(String[] args) {

short [] array = {12, 2, 3, 45, 2, 7, 9};

List<Short> listOfShort = asList(array);

System.out.println("asList (): "+listOfShort);

array = toArray(listOfShort);

System.out.println("toArray (): "+Arrays.toString(array));

reverse(array);

System.out.println("reserve (): "+Arrays.toString(array));

byte[] byteArray = toByteArray((short) 13);

System.out.println("toByteArray (): "+Arrays.toString(byteArray));

String text = join(", ", array);

System.out.println("join (): "+text);

System.out.println("\n =========================");

System.out.println("Максимальный элемент: "+max(array));

System.out.println("Минимальный элемент: "+ min(array));

if(contains(array, (short) 45))

System.out.println("45 содержится в array.");

else

System.out.println("45 не содержится в array.");

System.out.println("\nПоследнее вхождение 56: "+lastIndexOf(array, (short) 56));

System.out.println("Первое вхождение 45: "+indexOf(array, (short) 45));

}

}

Класс Ints.

Данный утилит класс служит оболочкой для примитивов типа int. Его можно рассматривать как обогащенный аналог Integer. Все методы данного класса static и конструктор данного класса закрыт. Ниже перечислены методы данного класса:

|  |  |
| --- | --- |
| Объявление | Описание |
| static List<Integer> asList (int.. *числа*) | Возвращает List *чисел*. |
| static int checkedCast (long *число*) | Возвращает int равного значения *числа* без потери, если это возможно. |
| static int compare (int *значение1*, int *значение2*) | Сравнивает два числа. |
| static int[] concat (int []… *массивы*) | Соединяет *массивы*. |
| static boolean contains (int [] *массив*, int *число*) | Проверяет наличие *числа*. |
| static int [] ensureCapacity (int [] *массив*, int *минимальная\_длина*, int *длина*) | Возвращает массив с *длиной* и с теми же значениями *массива*. |
| static int fromBytesArray (byte [] *массив*) | Возвращает int, который равен содержимым *массива*. |
| static int fromBytes (byte *байт1*, byte *байт2*, byte *байт3*, byte *байт4*) | Эквивалент fromBytesArray (new byte [] {byte *байт1*, byte *байт2*, byte *байт3*, byte *байт4*}). |
| static int hashCode (int *значение*) | Возвращает хэш-код *значения*. |
| static int min (int … *массив*) | Возвращает наименьший элемент из *массива*. |
| static int max (int … *массив*) | Возвращает наибольший элемент из *массива*. |
| static int indexOf (int [] *массив*, int *значение*) | Ищет первое вхождение *значения* в *массиве*, иначе -1. |
| static int lastIndexOf (int [] *массив*, int [] *значение*) | Ищет первое вхождение *значения* в *массиве*, иначе -1. |
| static String join (String *разделитель*, int… *числа*) | Возвращает строку содержащие *числа* через *разделитель*. |
| static Comparator<int []> lexicographicalComparator() | Возвращает компаратор, который сравнивает два int лексикографически. |
| static int [] toArray (Collections <? extends Number> *коллекция*) | Преобразует *коллекцию* в массив int. |
| static byte [] toByteArray (int *значение*) | Представляет байтовое представление *значения*. |
| static Convertor <String, Integer> stringConvertor() | Возвращает Convector. |
| static Integer tryParse (String *строка*) | Преобразует *строку* в число, если формат строки не позволяет то null |
| static int saturatedCast (long *значение*) | Возвращает наиближайший к *значению* значение. |
| static void reserve (int [] *массив*) | Переворачивает *массив*. |
| static void sortDescending (int [] *массив*) | Перемешивает *массив*. |

Ниже представлен пример использования данного класса:

package Example;

import static com.google.common.primitives.Ints.\*;

import java.util.\*;

public class GIntsExample {

public static void main(String[] args) {

int [] array = {2, 1, 46, 34, 65, 124, 54};

List<Integer> listOfIntegers = asList(array);

System.out.println("asList (): "+listOfIntegers);

array = toArray(listOfIntegers);

System.out.println("toArray (): "+Arrays.toString(array));

reverse(array);

System.out.println("reserve (): "+Arrays.toString(array));

byte[] bytes = toByteArray(1522);

System.out.println("toByteArray (): "+Arrays.toString(bytes));

System.out.println("\n ============================= ");

System.out.println("Минимальный элемент: "+min(array));

System.out.println("Максимальный элемент: "+max(array));

if (contains(array, 34))

System.out.println("34 содержится в array.");

else

System.out.println("34 не содержится в array.");

System.out.println("\nПервое вхождение 1: "+indexOf(array, 1));

System.out.println("Последнее вхождение 57: "+lastIndexOf(array, 57));

}

}

Класс Longs.

Данный утилит класс служит оболочкой для примитивов типа long. Его можно рассматривать как обогащенный аналог класса Long. Все его методы статичны, а конструктор закрыт. Методы его класса абсолютно схожи с методами классов Bytes, Short и Ints.

Ниже представлен пример использования данного класса:

package Example;

import static com.google.common.primitives.Longs.\*;

import java.util.\*;

public class GLongsExample {

public static void main(String[] args) {

long [] array = {35, 70, -23, 245, 127, 953, -12, -324};

List<Long> listOfLong = asList(array);

System.out.println("asList (): "+listOfLong);

array = toArray(listOfLong);

System.out.println("toArray (): "+Arrays.toString(array));

reverse(array);

System.out.println("reserve (): "+Arrays.toString(array));

byte[] bytes = toByteArray(1375);

System.out.println("toByteArray (1375): "+Arrays.toString(bytes));

String text = join(", ", array);

System.out.println("join (): "+text);

System.out.println("\n =============================== ");

System.out.println("Минимальный элемент: "+min(array));

System.out.println("Максимальный элемент: "+max(array));

if(contains(array, -12))

System.out.println("В array содержится -12.");

else

System.out.println("В array не содержится -12 \n");

System.out.println("Первое вхождение 70: "+indexOf(array, 70));

System.out.println("Последнее вхождение 245: "+lastIndexOf(array, 245));

}

}

Класс Floats.

Класс утилит Floats служит для оболочек типа floats. Его можно рассматривать как обогащенный аналог класс Float. Ниже перечислены методы данного класса:

|  |  |
| --- | --- |
| Объявление | Описание |
| static List <Float> asList (float … *числа*) | Возвращает List *чисел*. |
| static int compare (float *число1*, float *число2*) | Сравнивает оба числа. |
| static float [] concat (float []… *массивы*) | Соединяет *массивы*. |
| static boolean contains (float []... *массив*, float *значение*) | Проверяет *значение* на наличие его в *массиве*. |
| static float [] ensureCapacity (float [] *массив*, int *минимальная\_длина*, int *длина*) | Создает массив, содержащий те же значения, что и *массив*, но с указанной *длиной*. |
| static int hashCode (float *значение*) | Возвращает хэш-код *значения*. |
| static int indexOf (float [] *массив*, float *число*) | Возвращает первое вхождение *числа* в *массиве*, иначе -1. |
| static int indexOf (float [] *массив*, float [] *значения*) | Возвращает первое вхождение *значения* в *массиве*, иначе -1. |
| static int lastIndexOf (float [] *массив*, float *значение*) | Возвращает последнее вхождение *значение* в *массиве*. |
| static boolean isFinite (float *число*) | Проверяет на действительность *числа*. |
| static String join (String *разделитель*, float… *значения*) | Возвращает строку содержащая *значения* через *разделитель*. |
| static Comparator<float []> lexicographicalComparator() | Возвращает компаратор, который сравнивает два float лексикографически. |
| static float min (float … *числа*) | Возвращает наименьшее число из *чисел*. |
| static float max (float … *числа*) | Возвращает наибольшее число из *чисел*. |
| static Converter <String, Float> stringConvertor () | Возвращает Converter. |
| static float [] toArray (Collections <? extends Number> *коллекция*) | Возвращает массив, содержащий значения из *коллекции*. |
| static Float tryParse (String *строка*) | Преобразует *строку* в тип float, иначе null. |
| static void reserve (float [] *массив*) | Переворачивает *массив*. |
| static void sortDescending (float [] *массив*) | Перемешивает *массив*. |

Ниже представлен пример использования класса Floats:

package Example;

import com.google.common.primitives.Floats;

import static com.google.common.primitives.Floats.\*;

import java.util.\*;

public class GFloatsExample {

public static void main(String[] args) {

float [] array = {12.23f, -89.0f, 24f, 75.32f, -456.23f};

List<Float> listOfFloat = asList(array);

System.out.println("asList (): "+listOfFloat);

array = toArray(listOfFloat);

System.out.println("toArray (): "+Arrays.toString(array));

reverse(array);

System.out.println("reserve (): "+Arrays.toString(array));

String text = join(", ", array);

System.out.println("join (): "+text);

Float value = tryParse("234.12");

System.out.println("tryParse (): "+value);

System.out.println("\n============================= ");

System.out.println("137.34f реальное число: "+isFinite(137.34f));

System.out.println("Наименьший элемент: "+min(array));

System.out.println("Наибольший элемент: "+max(array));

if(Floats.contains(array, 35.34f))

System.out.println("35.34 содержится в array.");

else

System.out.println("35.34 не содержится в array.");

System.out.println("\nПервое вхождение 24: "+indexOf(array, 24f));

System.out.println("Последнее вхождение -89.0f: "+lastIndexOf(array, -89.0f));

}

}

Класс Doubles.

Данный утилит класс Doubles является оболочкой для примитивов типа double. Его можно рассматривать как обогащенный аналог класса Double. Ниже перечислены его методы.

|  |  |
| --- | --- |
| Объявление | Описание |
| static List<Double> asList (double … *числа*) | Возвращает List *чисел*. |
| static int compare (double *число1*, double *число2*) | Сравнивает два числа. |
| static double [] concat (double []… *массив*) | Соединяет *массивы*. |
| static boolean contains (double []*массив* , double *значение*) | Проверяет на наличие *значения* в *массиве*. |
| static int hashCode (double *значение*) | Возвращает хэш-код *значения*. |
| static int indexOf (double [] *массив*, double *значение*) | Возвращает первое вхождение *значения* в *массиве*, иначе -1. |
| static int indexOf (double [] *массив*, double [] *значение*) | Возвращает первое вхождение *значения* в *массиве*, иначе -1. |
| static int lastIndexOf (double [] *массив*, double *значение*) | Возвращает последнее вхождение *значения* в *массиве*, иначе -1. |
| static String join (String *разделитель*, double … *значения*) | Возвращает строку, содержащая *значения* через *разделитель*. |
| static boolean isFinite (double *число*) | Проверяет на действительность *числа*. |
| static boolean [] ensureCapacity (double [] *массив*, int *минимальная\_длина*, int *длина*) | Создает массив, содержащий те же значения, что и *массив*, но с указанной *длиной*. |
| static double min (double … *массив*) | Возвращает наименьший элемент *массива*. |
| static double max (double … *массив*) | Возвращает наибольший элемент *массива*. |
| static Converter<String, Double> stringConverter() | Возвращает Converter. |
| static void reserve (double [] *массив*) | Переворачивает *массив*. |
| static double [] toArray (Collection <? extends Number> *коллекция*) | Преобразует содержимое *коллекции* в массив. |
| static Double tryParse (String *строка*) | Преобразует содержимое *строки* в Double, иначе null. |
| static void sortDescending (double [] *массив*) | Перемешивает *массив*. |

Ниже представлен пример использования выше описанного класса:

package Example;

import static com.google.common.primitives.Doubles.\*;

import java.util.\*;

public class GDoublesExample {

public static void main(String[] args) {

double [] array = {34.4, -2, 235.34, 345.2, -12.5};

List<Double> listOfDouble = asList(array);

System.out.println("asList (): "+listOfDouble);

array = toArray(listOfDouble);

System.out.println("toArray (): "+Arrays.toString(array));

reverse(array);

System.out.println("reserve (): "+Arrays.toString(array));

Double value = tryParse("12.46");

System.out.println("tryParse (): "+value);

String text = join(", ", array);

System.out.println("join (): "+text);

sortDescending(array);

System.out.println(Arrays.toString(array));

System.out.println("\n========================== ");

System.out.println("Число Пи это реальное число: "+isFinite(22 / 7));

System.out.println("Наименьший элемент: "+min(array));

System.out.println("Наибольший элемент: "+max(array));

if(contains(array, -12.5))

System.out.println("-12.5 содержится в array.");

else

System.out.println("-12.5 не содержится в array.");

System.out.println("\nПервое вхождение -2: "+indexOf(array, -2));

System.out.println("Последнее вхождение -12.5: "+lastIndexOf(array, -12.5));

}

}

Класс Booleans.

Данный утилит класс Booleans является оболочкой для примитивов типа boolean. Его можно рассматривать как обогащенный аналог класса Boolean. Ниже перечислены его методы:

|  |  |
| --- | --- |
| static List <Boolean> asList (boolean … *значения*) | Возвращает List *значений*. |
| static int compare (boolean *значение1*, boolean *значение2*) | Сравнивает оба значения. |
| static boolean [] concat (boolean [] … *массивы*) | Соединяет все *массивы*. |
| static boolean contains (boolean [] *массив*, boolean *значение*) | Проверяет на наличие *значения* в *массиве*. |
| static int countTrue (boolean … *массив*) | Посчитывает количество trueв *массиве*. |
| static boolean [] ensureCapacity (boolean [] *массив*, int *минимальная\_длина*, int *длина*) | Создает массив, содержащий те же значения, что и *массив*, но с указанной *длиной*. |
| static int hashCode (boolean *значение*) | Возвращает хэш-код *значения*. |
| static int indexOf (boolean [] *массив*, boolean *значение*) | Возвращает первое вхождение *значение* в *массиве*. |
| static int indexOf (boolean [] *массив*, boolean [] *значение*) | Возвращает первое вхождение *значение* в *массиве*. |
| static int lastIndexOf (boolean [] *массив*, boolean *значение*) | Возвращает последнее вхождение *значение* в *массиве*. |
| static String join (String *разделитель*, boolean … *значение* ) | Возвращает строку, содержащая *значения* через *разделитель*. |
| static boolean [] toArray (Collection <Boolean> *коллекция*) | Преобразует *коллекцию* в массив. |
| static Comparator<boolean []> lexicographicalComparator() | Возвращает компаратор, который сравнивает два boolean лексикографически. |
| static void reserve (boolean [] *массив*) | Переворачивает *массив*. |

Ниже представлен пример использования выше описанного класса:

package Example;

import static com.google.common.primitives.Booleans.\*;

import java.util.\*;

public class GBooleansExample {

public static void main(String[] args) {

boolean [] array = {false, false, true, false, true, true};

List<Boolean> listOfBooleans = asList(array);

System.out.println("asList (): "+listOfBooleans);

array = toArray(listOfBooleans);

System.out.println("toArray (): "+Arrays.toString(array));

reverse(array);

System.out.println("reserve (): "+Arrays.toString(array));

System.out.println("\n==============================");

System.out.println("В array содержится true: "+contains(array, true));

System.out.println("Последнее вхождение false: "+lastIndexOf(array, false));

System.out.println("Первое вхождение true: "+indexOf(array, true));

System.out.println("Количество true: "+countTrue(array));

System.out.println("\n true vs false: "+compare(true, false));

System.out.println(" false vs true: "+compare(false, true));

System.out.println(" false vs false: "+compare(false, false));

System.out.println(" true vs true: "+compare(true, true));

}

}

Класс Chars.

Данный утилит класс оболочкой для примитивов типа char. Его можно рассматривать как обогащенный аналог класса Character. Ниже перечислены методы данного класса:

|  |  |
| --- | --- |
| Объявление | Описание |
| static List<Character> asList (char … *символы*) | Возвращает List *символов*. |
| static char checkedCast (long *значение*) | Если можно возвращает символ равное *значению*. |
| static int compare (char *символ*, char *символ1*) | Сравнивает оба символа. |
| static char [] concat (char [] … *массив*) | Соединяет все *массивы*. |
| static boolean contains (char [] *массив*, char *символ*) | Проверяет наличие *символа* в *массиве*. |
| static char [] ensureCapacity (char [] *массив*, int *минимальная\_длина*, int *длина*) | Создает массив, содержащий те же значения, что и *массив*, но с указанной *длиной*. |
| static char fromByteArray (byte [] *массив*) | Преобразует содержимое *массива* в символ. |
| static char fromBytes (byte *байт1*, byte *байт2*) | Преобразует оба байта в символ. |
| static int hashCode (char *символа*) | Возвращает хэш-код *символа*. |
| static int indexOf (char [] *массив*, char *значение*) | Возвращает первое вхождение *значения* в *массиве*, иначе -1. |
| static int indexOf (char [] *массив*, char [] *значение*) | Возвращает первое вхождение *значения* в *массиве*, иначе -1. |
| static int lastIndexOf (char [] *массив*, char *символ*) | Возвращает последнее вхождение *символа* в *массив*, иначе -1. |
| static String join (String *разделитель*, char … *символы*) | Возвращает строку, содержащая *символы* через *разделитель*. |
| static Comparator<char []> lexicographicalComparator() | Возвращает компаратор, который сравнивает два char лексикографически. |
| static char min (char … *массив*) | Возвращает наименьший символ из *массива*. |
| static char max (char … *массив*) | Возвращает наибольший символ из *массива*. |
| static char saturatedCast (long *число*) | Возвращает наиближайшее к *числу* char. |
| static char [] toArray (Collection <Character> *коллекция*) | Возвращает массив, содержащий элементы *коллекции*. |
| static byte [] toByteArray (char *символ*) | Возвращает байтовое представление *символа*. |

Ниже представлен пример использования данного класса:

package Example;

import static com.google.common.primitives.Chars.\*;

import java.util.\*;

public class GCharsExample {

public static void main(String[] args) {

char [] array = {'a', 'b', 'c', '1', '2', '3', ' ', '+'};

List<Character> listOfChar = asList(array);

System.out.println("asList (): "+listOfChar);

array = toArray(listOfChar);

System.out.println("toArray (): "+Arrays.toString(array));

reverse(array);

System.out.println("reserve (): "+Arrays.toString(array));

sortDescending(array);

System.out.println("sortDescending (): "+Arrays.toString(array));

byte[] bytes = toByteArray('k');

System.out.println("toByteArray ('k'): "+Arrays.toString(bytes));

String text = join(", ", array);

System.out.println("join (): "+text);

System.out.println("\n =================================");

System.out.println("Наименьший элемент: "+min(array));

System.out.println("Наибольший элемент: "+max(array));

System.out.println("j находится в array: "+contains(array, 'j'));

System.out.println("\nПервое вхождение 2: "+indexOf(array, '2'));

System.out.println("Последнее вхождение b: "+lastIndexOf(array, 'b'));

System.out.println("\n 265 : "+checkedCast(265));

System.out.println(" 45768 : "+saturatedCast(45768));

}

}

Коллекции.

В Гуава предоставляется ряд коллекций. Она основаны на опыте и потребностях разработчиков. Для коллекций выделен отдельный пакет com.google.common.collect. Ниже перечислены основные коллекции:

* Multiset
* Multimap
* BiMap
* Table

Интерфейс Multiset.

Данный интерфейс расширяет интерфейс Collection и условно расширяет класс Set . За отличием обычных множеств он отличается тем, что он может иметь повторяющие значения.

Такие коллекции называются *мультимножествами*. Также он предоставляет методы для работы с данными множествами.

Он является обобщенным, где E обозначает тип содержащихся объектов:

interface Multiset <E> extends Collection<E>

Ниже перечислены методы данного интерфейса:

|  |  |
| --- | --- |
| Объявление | Описание |
| boolean add (E *объект*) | Добавляет *объект* в множество. |
| int add (E *объект*, int *индекс*) | Добавляет *объект* по *индексу* в множеству. |
| boolean contains (Object *объект*) | Проверяет наличие *объекта* в множестве. |
| boolean containsAll (Collection <?> *коллекция*) | Проверяет наличие *коллекции* в множестве. |
| int count (Object *объект*) | Подсчитывает количество *объектов*. |
| Set<E> elementSet () | Возвращает Set исходя вызывающего Multiset. |
| Set<Multiset.Entry<E>> entrySet() | Возвращает Set состоящий изMultiset.Entry. |
| void forEach (Consumer <? super T> *действие*) | Передает содержимое *действию*. |
| boolean equals (Object *объект*) | Сравнивает *объект* с вызывающим мультимножеством. |
| int hashCode () | Возвращает хэш-код множества. |
| Iterator <E> iterator () | Возвращает Iterator из элементов мультимножества. |
| boolean remove (Object *объект*) | Удаляет *объект* из множества. |
| boolean removeAll (Collection <?> *коллекция*) | Удаляет все совпавшие элементы с *коллекции.* |
| boolean retainAll (Collection <?> *коллекция*) | Сохраняет лишь те методы, которые совпадают с элементами *коллекции*. |
| String toString () | Возвращает строковое представление коллекции. |

Пример мультимножества на классе HashMultiset.

Класс HashMultiset реализует интерфейс Multiset. Он служит аналогом HashSet. Для создания его можно воспользоваться фабричным методом create (). Остальные методы представлены ниже:

|  |  |
| --- | --- |
| Объявление | Описание |
| Stream <T> stream () | Возвращает Stream API |
| Stream <T> parallelStream () | Возвращает параллельный Stream API |
| boolean isEmpty () | Проверяет на пустоту массив. |
| boolean removeIf (Predicate <? super T> *условие*) | Удаляет лишь те элементы, которые проходят условие. |
| void clear () | Чистит множество. |
| void forEach (Consumer <? super T> *действие*) | Передает элементы *действию*. |
| Object [] toArray () | Преобразуем множество в массив. |
| boolean addAll (Collection <?> *коллекция*) | Добавляет в множество *коллекцию*. |

Ниже представлен пример создания мультимножества:

package Example;

import com.google.common.collect.\*;

import java.util.Arrays;

public class GMultisetExample {

public static void main(String[] args) {

Multiset<String> set = HashMultiset.create();

set.add("Optional");

set.add("Ordering");

set.add("Doubles");

set.add("Floats");

set.add("CaseFormat");

set.add("Joiner");

set.add("Splitter");

set.add("Joiner");

set.removeIf("CaseFormat"::equals);

long lengthAll = set.stream().

map(str -> str.length()).

reduce(0, (val, val1) -> val + val1);

System.out.println("Размер: "+set.size());

System.out.println("Множество пусто: "+set.isEmpty());

System.out.println("Длина всех строчек: "+lengthAll);

System.out.println("Хэш-код: "+set.hashCode());

System.out.println("Количество Joiner: "+set.count("Joiner"));

System.out.println("Chars находится в множестве: "+set.contains("Chars"));

System.out.println("\nСодержимое множества: ");

set.forEach(System.out::println);

Object [] array = set.toArray();

System.out.println("\nМассив: "+Arrays.toString(array));

}

}

Интерфейс Multimap.

Интерфейс Multimap служит для создания пар, которые могут использовать повторяющие ключи. Он является обобщенным, где K обозначает тип ключа, а V тип значения.

Ниже представлены методы данного интерфейса:

|  |  |
| --- | --- |
| Объявление | Описание |
| Map <K, Collection <V> > asMap () | Возвращает представление мульти пар в виде Map. |
| int size () | Возвращает размер. |
| boolean isEmpty () | Проверяет на пустоту отображение. |
| int hashCode () | Возвращает хэш-код коллекции. |
| boolean equals (Object *объект*) | Сравнивает *объект* с отображением. |
| boolean put (K *ключи*, V *значение*) | Добавляет *значение* с *ключом*. |
| boolean putAll (K *ключ*, Iterable <? extends V> *значение*) | Добавляет *значения* по *ключу*. |
| boolean putAll (Multimap <? extends K, ? extends V> *отображение*) | Добавляет *отображение*. |
| boolean remove (Object *ключ*, Object *значение*) | Удаляет *значение* с *ключом*. |
| boolean removeAll (Object *ключ*) | Удаляет все значения по *ключу*. |
| Collection <V> get (K *ключ*) | Возвращает все значения *ключа*. |
| Set<K> keySet () | Возвращает множество ключей. |
| Multiset <K> keys () | Возвращает мультимножество ключей. |
| Collection <V> values () | Возвращает коллекцию значений. |
| Collection <Entry <K, V>> entries () | Возвращает представление в виде Entry<K, V> |
| default void forEach (BiConsumer <? super K, ? super V> *действие*) | Передает содержимое *действию*. |
| Collection <V> replaceValues (K *ключ*, Iterable <? extends V> *значение*) | Заменяет значения *ключ* на *значение*. |
| boolean containsEntry (Object *ключ*, Object *значение*) | Проверяет наличие пары. |
| boolean containsValue (Object *значение*) | Проверяет наличие *значения*. |
| boolean containsKey (Object *ключ*) | Проверяет наличие *ключа*. |

Класс HashMultimap.

Класс HashMultimap служит аналогом HashMap. Он определяет выше описанные методы, а для то чтоб его создать следует воспользоваться фабричным метод create (). Ниже представлен пример использования класса – коллекции:

package Example;

import com.google.common.collect.\*;

import java.util.Arrays;

public class GMultimapExample {

public static void main(String[] args) {

Multimap<String, String> map = HashMultimap.create();

map.put("guava", "Multiset");

map.put("java", "TimerTask");

map.put("guava", "Ordering");

map.put("jsoup", "Document");

map.put("java", "DateFormat");

map.put("guava", "Shorts");

System.out.println("Размер: "+map.size());

System.out.println("Он пуст: "+map.isEmpty());

System.out.println("Хэш-код: "+map.hashCode());

System.out.println("\nСодержимое: "+map);

System.out.println("Ключи: "+map.keySet());

System.out.println("Значение: "+map.values()+"\n");

System.out.println("Значения по ключу guava: "+map.get("guava"));

System.out.println("Значения по ключу java: "+map.get("java"));

System.out.println("Значение по ключу jsoup: "+map.get("jsoup"));

map.replaceValues("java", Arrays.asList("Runnable", "Thread", "Locale"));

System.out.println("После замены: "+map.get("java")+"\n");

map.forEach(

(s, s1) ->

System.out.println(s.toUpperCase()+" |"+s1.toLowerCase()));

}

}

Интерфейс BIMap.

Данный интерфейсы определяет *бимапы*, который не может иметь дублированный ключей. BiMap расширяет Map и является обобщенным. Где K тип ключа, V тип значения. Полное объявление:

interface BiMap <K, V> extends Map <K, V>

Также он определяет ряд своих методов. Ниже они перечислены:

|  |  |
| --- | --- |
| Объявление | Описание |
| V forcePut (K *ключ*, V *значение*) | Добавляет *значение* и удаляет все совпавшие значения |
| BiMap <V, K> inverse () | Возвращает бимап, где ключ – значение, а значение – ключ. |
| Set <V> values () | Возвращает множество значений. |
| V put (K *ключ*, V *значение*) | Добавляет *значение* по *ключом*. |
| void putAll (Map <? extends K, ? extends V> *карта*) | Добавляет *карту*. |

Класс HashBiMap.

Класс коллекция HashBiMap является одной из реализации BiMap. Для его создания следует воспользоваться фабричным методом create (). Ниже представлен пример бимапа:

package Example;

import com.google.common.collect.\*;

import java.util.Set;

public class GBiMapExample {

public static void main(String[] args) {

BiMap<String, String> map = HashBiMap.create();

map.put("java.util.Optional", "Optional") ;

map.forcePut("com.google.common.base.Optional", "Optional");

map.put("javax.swing.text", "Document");

map.put("java.lang.Math", "Math");

map.put("com.google.common.collection.BiMap", "BiMap");

System.out.println("Размер: "+map.size());

System.out.println("Отображение пусто: "+map.isEmpty());

System.out.println("\n"+"Содержимое: ");

map.forEach(

(str, str1) ->

System.out.println(str+" | "+str1));

Set<String> values = map.values();

System.out.println("\n" + "Значения: "+values);

System.out.println("\n"+"inverse ():");

BiMap<String, String> inverse = map.inverse();

inverse.forEach((str, str1) ->

System.out.println(str+" | "+str1));

}

}

Интерфейс Table.

Данный интерфейс служит для создания таблиц. Он является обобщенным и полное объявление выглядит так:

interface Table <R, C, V>

Здесь R тип ключа таблицы, C тип ключа колоны и V тип значений. Ниже перечислены методы данного интерфейса:

|  |  |
| --- | --- |
| Объявление | Описание |
| V put (R *ключ*, C *ключ\_колоны*, V *значение*) | Добавляет *значение* в *колоне* по *ключу*. |
| void put (Table <? extends R, ? extends C , ? extends V> *таблица*) | Добавляет *таблицу*. |
| V get (Object *ключ*, Object *колона*) | Возвращает значение в *колоне* по *ключу*. |
| V remove (Object *ключ*, Object *колона*) | Удаляет связанное отображение. |
| void clear () | Чистит таблиц. |
| int size () | Возвращает количество ключей. |
| int hashCode () | Возвращает хэш-код. |
| boolean equal (Object *объект*) | Сравнивает *объект* с таблицей. |
| boolean isEmpty () | Проверяет на пустоту. |
| Collection <V> values () | Возвращает все значения. |
| columnKeySet () | Возвращает все колоны. |
| Set <R> rowKeySet () | Возвращает все ключи. |
| boolean containsColumn (C *колона*) | Проверяет наличие *колоны*. |
| boolean containsRow (R *ключ*) | Проверяет наличие *ключа*. |
| boolean containsValue (V *значение*) | Проверяет наличие *значение*. |

Класс HashBasedTable.

Данный класс является одним из многих классов реализующих интерфейс Table. Для создания следует воспользоваться методом create (). Ниже представлен пример создании таблицы:

package Example;

import com.google.common.collect.\*;

public class GTableExample {

public static void main(String[] args) {

Table <String, String, String> table = HashBasedTable.create();

table.put("IBM", "101","Mahesh");

table.put("IBM", "102","Ramesh");

table.put("IBM", "103","Suresh");

table.put("Microsoft", "111","Sohan");

table.put("Microsoft", "112","Mohan");

table.put("Microsoft", "113","Rohan");

table.put("TCS", "121","Ram");

table.put("TCS", "122","Shyam");

table.put("TCS", "123","Sunil");

System.out.println("Размер: "+table.size());

System.out.println("Он пуст: "+table.isEmpty());

System.out.println("Хэш-код: "+table.hashCode()+"\n");

System.out.println("IBM "+table.row("IBM"));

System.out.println("Microsoft "+table.row("Microsoft"));

System.out.println("TCS "+table.row("TCS")+"\n");

System.out.println("values (): "+table.values());

System.out.println("columnKeySet (): "+table.columnKeySet());

System.out.println("rowKeySet (): "+table.rowKeySet()+"\n");

if(table.containsColumn("121"))

System.out.println("Колона 121 существует.");

else

System.out.println("Колона 121 не существует.");

if (table.containsRow("IBM"))

System.out.println("Ключ IBM существует.");

else

System.out.println("Ключ IBM не существует");

}

}

Кэширующая утилита.

Библиотека Гуава предоставляет обширные средства кэширования. Все интерфейсы и класс в guava находятся в пакете com.google.common.cache.

Интерфейс Cache.

Интерфейс Cache служит базовым для кэширования в Гуава. Он основные методы в кэширования. Он является обобщенным и где V тип значения, а K тип ключа. Ниже перечислены методы данного интерфейса:

|  |  |
| --- | --- |
| Объявление | Описание |
| ConcurrentMap asMap () | Возвращает потоко - безопасное представление содержимого кэша. |
| void cleanUp () | Выполняет любые ожидающие операции обслуживания, необходимые кэшу. |
| CacheStats stats() | Возвращает текущий снимок кумулятивной статистики этого кэша или набор значений по умолчанию, если кэш не записывает статистику |
| V get (K *ключ*, Callable <? extends V> *загрузчик*) | Возвращает значение по *ключу* применяя *загрузчик*. |
| V getIfPresent (Object *ключ*) | Возвращает значение по *ключу*, если значение не пусто. |
| ImmutableMap<K,V> getAllPresent (Iterable<?> *ключи*) | Возвращает значения по *ключам*. |
| void put (K *ключ*, V *значение*) | Добавляет *значение* в хэш по *ключу*. |
| void putAll (Map <? extends K , ? extends V> *карта*) | Вводит *карту* в хэш. |
| void invalidate (Object *ключ*) | Удаляет хэш *ключа*. |
| void invalidateAll () | Чистит кэш. |
| void invalidateAll (Iterable <?> *ключи*) | Чистит значения по с совпавшими *ключам*. |

Класс CacheBuilder.

Данный класс предоставляет методы для создания Cache т.е. его строит. Среди всех стоит первоначально отметить статический метод newBuilder (), который возвращает CacheBuilder. Полное объявление:

static CacheBuilder<Object, Object> newBuilder()

CacheBuilder объявляет «промежуточные методы для стройки». Каждый из них возвращает измененный Cache. К примеру метод concurrencyLevel (*количество*), возвращает Cache, который одновременно могут использовать *количество* потоков. Ниже представлены избранные «промежуточные методы для стройки»:

|  |  |
| --- | --- |
| Объявление | Описание |
| CacheBuilder<K,V> maximumSize(long *n*) | Возвращает CacheBuilder, который сможет содержать *n* количество записей. |
| CacheBuilder<K,V> initialCapacity(int *начальный\_размер*) | Возвращает CacheBuilder с *начальным размером*. |
| CacheBuilder<K,V> expireAfterWrite(long *время*, TimeUnit *тип*) | Возвращает CacheBuilder, который автоматически удаляет кэш по *типу времени*. |
| CacheBuilder<K, V> removalListener (RemoveListener <? super K1,? super V1> *слушатель*) | Возвращает CacheBuilder с *слушателем*. |
| CacheBuilder <K, V> softValues() | Возвращает CacheBuilder, значения которого обвернуты в SoftReference. |

И последнее, что нужно чтоб построить кэш воспользоваться в конце методом build (), которые вернет Cache.

Пример создания кэша.

Ниже представлен пример создания кэша:

package Example;

import com.google.common.cache.\*;

import java.util.concurrent.\*;

public class GOrderingExample {

static Cache<Integer, String> cache =

CacheBuilder.

newBuilder().

initialCapacity(36).

concurrencyLevel(8).

softValues().

maximumSize(500).

expireAfterWrite(5, TimeUnit.SECONDS).

build();

public static void main(String[] args) {

cache.put(1, "one");

cache.put(2, "two");

cache.put(3, "three");

cache.put(4, "four");

cache.put(5, "five");

System.out.println("Размер кэша: "+cache.size());

System.out.println("Содержимое: ");

cache.asMap().forEach((k, v) -> System.out.println(k + ": " + v));

System.out.println("Содержимое после удаления: ");

cache.invalidate(3);

cache.asMap().forEach((k, v) -> System.out.println(k+": "+v));

}

}

Математическая утилита.

Библиотека Гуава предоставляет ряд классов для математических операций. Все они являются утилитами, с закрытыми методами и статичными методами.

Класс IntMath.

Данный утилит класс IntMath объявляет методы для математических операций. В нижней таблице перечислены методы данного класса:

|  |  |
| --- | --- |
| Объявление | Описание |
| static int binomial (int *n*, int *k*) | Возвращает биномиальный коэффициент *n* и *k*. |
| static int checkedAdd (int *a*, int *b*) | Возвращает сумму *a* и *b*, если она не переполняется. |
| static int checkedMultiply(int *a*, int *b*) | Возвращает продукт a и *b* если она не переполняется. |
| static int checkedPow (int *a*, int *b*) | Возвращает *a* в степени *b*, если она не переполняется. |
| static int checkedSubtract (int *a*, int *b*) | Возвращает разницу *a* и *b*, если она не переполняется. |
| static int divide (int *a*, int *b*, RoundingMode *режим*) | Возвращает результат деления *a* и *b*, округленная *режимом*. |
| static int factorial (int *n*) | Возвращает факториал *n*. |
| static int gcd (int *a*, int *b*) | Возвращает НОД *a* и *b*. |
| static boolean isPowerOfTwo (int *x*) | Вернет true если *x* содержит мощность двух. |
| static int log10 (int *x*, RoundingMode *режим*) | Возвращает логарифм 10 x, округленный в *режимом* округления. |
| static int log2 (int *x*, RoundingMode *режим*) | Возвращает логарифм 2 x, округленный в *режимом* округления. |
| static int pow (int *a*, int *b*) | Возвращает *a* в степени *b*. |
| static int sqrt (int *x*, RoundingMode *режим*) | Возвращает квадратный корень *x*, округленный в *режимом* округления. |

Ниже представлен пример использования класса IntMath:

package Example;

import java.math.RoundingMode;

import static com.google.common.math.IntMath.\*;

public class GIntMathExample {

public static void main(String[] args) {

System.out.println("binomial (34, 32) -> "+binomial(34, 32));

System.out.println("sqrt (34, DOWN) -> "+sqrt(34, RoundingMode.DOWN));

System.out.println("pow (23, 23) -> "+pow(23, 23));

System.out.println("factorial (34) -> "+factorial(34));

System.out.println("log2 (23, DOWN) -> "+log2(23, RoundingMode.DOWN));

System.out.println("log10 (23, DOWN) -> "+log10(23, RoundingMode.DOWN));

try {

System.out.println(checkedAdd(Integer.MAX\_VALUE, Integer.MAX\_VALUE));

} catch (ArithmeticException exc) {

System.out.println("Exception: "+exc.getMessage());

}

try{

System.out.println(divide(22, 7, RoundingMode.UNNECESSARY));

}catch (ArithmeticException exc){

System.out.println("Exception: "+ exc.getMessage());

}

}

}

Класс BigIntegerMath.

Данный класс предоставляет методы для математических операций с BigInteger. В нижней таблице перечислены методы данного класса:

|  |  |
| --- | --- |
| Объявление | Описание |
| static BigInteger binomial (BigInteger *n*, BigInteger *k*) | Возвращает биномиальный коэффициент *n* и *k*. |
| static BigInteger sqrt (BigInteger *x*, RoundingMode *режим*) | Возвращает квадратный корень *x*, округленный в *режимом* округления. |
| static BigInteger log10 (int *x*, RoundingMode *режим*) | Возвращает логарифм 10 x, округленный в *режимом* округления. |
| static BigInteger log2 (BigInteger *x*, RoundingMode *режим*) | Возвращает логарифм 2 x, округленный в *режимом* округления. |
| static BigInteger factorial (int *n*) | Возвращает факториал *n*. |
| static BigInteger divide(BigInteger *a*, BigInteger *b*, RoundingMode *режим*) | Возвращает результат деления *a* и *b*, округленная *режимом*. |
| static boolean isPowerOfTwo (BigInteger *x*) | Вернет true если *x* содержит мощность двух. |

Ниже представлен пример использования класса BigIntegerMath:

package Example;

import java.math.BigInteger;

import java.math.RoundingMode;

import static com.google.common.math.BigIntegerMath.\*;

public class GBigIntegerMathExample {

public static void main(String[] args) {

System.out.println("binomial (34, 32) -> "+binomial(34, 32));

System.out.println("sqrt (34, DOWN) -> "+sqrt(new BigInteger("34"), RoundingMode.DOWN));

System.out.println("factorial (12) -> "+factorial(12));

System.out.println("log2 (367, DOWN) -> "+log2(new BigInteger("367"), RoundingMode.DOWN));

System.out.println("log10 (12, DOWN) -> “+log10(new BigInteger(“12”), RoundingMode.DOWN));

try{

System.out.println(divide(new BigInteger(“27”),

new BigInteger(“7”),

RoundingMode.UNNECESSARY));

}catch (ArithmeticException exc){

System.out.println(“Exception: “+ exc.getMessage());

}

}

}

Класс LongMath.

Класс LongMath предоставляет обширный ассортимент методов для математических операций для примитивов типа long. В нижней таблице они представлены:

|  |  |
| --- | --- |
| Объявление | Описание |
| static long binomial (long *n*, long *k*) | Возвращает биномиальный коэффициент *n* и *k*. |
| static long checkedAdd (long *a*, long *b*) | Возвращает сумму *a* и *b*, если она не переполняется. |
| static long checkedMultiply(long *a*, long *b*) | Возвращает продукт a и *b* если она не переполняется. |
| static long checkedPow (long *a*, long *b*) | Возвращает *a* в степени *b*, если она не переполняется. |
| static long checkedSubtract (long *a*, long *b*) | Возвращает разницу *a* и *b*, если она не переполняется. |
| static long divide (long *a*, long *b*, RoundingMode *режим*) | Возвращает результат деления *a* и *b*, округленная *режимом*. |
| static long factorial (long *n*) | Возвращает факториал *n*. |
| static long gcd (long *a*, long *b*) | Возвращает НОД *a* и *b*. |
| static boolean isPowerOfTwo (int *x*) | Вернет true если *x* содержит мощность двух. |
| static long log10 (long *x*, RoundingMode *режим*) | Возвращает логарифм 10 x, округленный в *режимом* округления. |
| static long log2 (long *x*, RoundingMode *режим*) | Возвращает логарифм 2 x, округленный в *режимом* округления. |
| static long pow (long *a*, long *b*) | Возвращает *a* в степени *b*. |
| static long mean (long *x*, long *y*) | Возвращает среднеарифметическое *x* и y. |
| static long sqrt (long *x*, RoundingMode *режим*) | Возвращает квадратный корень *x*, округленный в *режимом* округления. |

Ниже представлен пример использования LongMath для некоторых математических операций:

package Example;

import java.math.RoundingMode;

import static com.google.common.math.LongMath.\*;

public class GLongMathExample {

public static void main(String[] args) {

System.out.println("binomial (34, 32) -> "+binomial(34, 32));

System.out.println("sqrt (34, DOWN) -> "+sqrt(34, RoundingMode.DOWN));

System.out.println("factorial (12) -> "+factorial(12));

System.out.println("pow (34, 23) -> "+pow(34, 23));

System.out.println("log2 (367, DOWN) -> "+log2(367, RoundingMode.DOWN));

System.out.println("log10 (23, DOWN) -> "+log10(12, RoundingMode.DOWN));

System.out.println("isPowerOfTwo (46) -> "+isPowerOfTwo(46));

try{

System.out.println(divide(27,7, RoundingMode.UNNECESSARY));

}catch (ArithmeticException exc){

System.out.println("Exception: "+ exc.getMessage());

}

try{

System.out.println(checkedAdd(Long.MAX\_VALUE, Long.MAX\_VALUE));

}catch (ArithmeticException exc){

System.out.println("Exception: "+exc.getMessage());

}

}

}

Парсинг с использованием Jsoup.

Jsoup это библиотека использованная для анализа документа HTML. Он заслужил свое внимание тем, что он располагал средства для парсинга сайта, т.е. получение ее html документа сайта. Эта функциональность равна ручного захода на сайт -> правой кнопки мыши -> нажать «просмотреть код страницы» -> скопировать содержимое.

Maven зависимость:

<dependency>

<groupId>org.jsoup</groupId>

<artifactId>jsoup</artifactId>

<version>1.9.1</version>

</dependency>

Получение html документа.

Класс Jsoup является основным классом в Jsoup. Он предоставляет методы, которые позволяют парсить html – файл, сайт либо же строку. Все они возвращают объект класс Document. Он инкапсулирует сам html документ и содержит методы для работы с ним.

Вот так может выглядеть парсинг файла, первый параметр это объект File, а второй это кодировка:

Document document = Jsoup.parse(new File("htmlFile"), "UTF-8");

Можно просто передать строку:

String html = "<html><head><title>Sample Title</title></head>"

+ "<body><p>Sample Content</p></body></html>";у

Document doc = Jsoup.parse(html);

Чтобы с парсить сайт есть варианта. Первый это использоваться connect (), а далее у Connection вызвать метод get (). Такой способ хорош когда нужно что – то настроить.

Document doc = Jsoup.connect("https://ru.stackoverflow.com/").get();

Избранные методы Document.

Document определяет ряд методов для манипуляции с документом, получения информации и настройки. Следует отметить тот факт, что Document реализует интерфейс Iterable.

|  |  |
| --- | --- |
| Объявление | Описание |
| Charset charset() | Возвращает кодировку. |
| void charset (Charset *кодировка*) | Устанавливает кодировку. |
| String title () | Возвращает заголовок. |
| String location () | Возвращает ссылку. |
| Document clone () | Возвращает точную копию документа. |
| Element head () | Возвращает содержимое тега head. |
| Element body () | Возвращает содержимое тега body. |
| Elements select (String *тег*) | Возвращает все *теги* из документа. |

Пример использования выше описанные методов:

String html = "<!DOCTYPE>\n" +

"<html>\n" +

" <head>\n" +

" <meta http-equiv=\"Content-Type\" content=\"text/html; charset=utf-8\">\n" +

" <title>Пример веб-страницы</title>\n" +

" </head>\n" +

" <body>\n" +

" <h1>Заголовок</h1>\n" +

" <!-- Комментарий -->\n" +

" <p>Первый абзац.</p>\n" +

" <p>Второй абзац.</p>\n" +

" <h1>Заголовок 2</h1>\n"+

" </body>\n" +

"</html>";

Document doc = Jsoup.parse(html);

System.out.println("Заголовок: "+doc.title());

System.out.println("Кодировка: "+doc.charset());

Element body = doc.body();

System.out.println("\nbody:\n"+body+"\n");

Element head = doc.head();

System.out.println("head:\n"+head+"\n");

Elements paragraph = doc.select("p");

System.out.println("\nВсе теги p:\n"+paragraph);

Класс Element и Elements.

Класс Element представляет собой отдельный элемент HTML документа. Ниже представлены основные методы для получения информации о элементе и его настройке:

|  |  |
| --- | --- |
| Объявление | Описание |
| String attr (String *атрибут*) | Возвращает значение *атрибута*. |
| void attr (String *атрибут*, String *значение*) | Заменяет *атрибут* на *значение*. |
| String id () | Возвращает ID атрибута. |
| String className () | Возвращает значение атрибута class. |
| Set <String> classNames () | Возвращает множество значений class. |
| String text() | Возвращает сочетание элемента и подэлементов. |
| void text (String *текст*) | Устанавливает *текст* в элементе. |
| Tag tag () | Возвращает объект Tag. |
| Element after (String *код*) | Возвращает следующий код после *код*. |
| String tagName () | Возвращает имя тега. |
| Element before (String *код*) | Возвращает предыдущий код после *код*. |

Также есть ряд методов для операции не посредственно с самим текстом. Ниже они перечислены:

|  |  |
| --- | --- |
| Объявление | Описание |
| Element append (String *код*) | Добавляет *код* в конец документа. |
| Element prepend (String *код*) | Добавляет *код* в начало документа. |
| Element prependText (String *строка*) | Добавляет *строку*  в начале. |
| Element appendText (String *строка*) | Добавляет *строку* в конце. |
| Element appendElement (String *тег*) | Добавляет *тег*. |
| Element prependElement (String *тег*) | Добавляет тег в начале. |
| Element html (String *код*) | Меняет содержимое на *код*. |

Класс Elements аналогичен классу за исключением того, что он представляет собой сразу несколько Element.

Сразу стоит отметить связь элементов от Document. Так, что если получить элементы от документа и поменять его это отразится и на Document.

String html = "<!DOCTYPE>\n" +

"<html>\n" +

" <head>\n" +

" <meta http-equiv=\"Content-Type\" content=\"text/html; charset=utf-8\">\n" +

" <title>Пример веб-страницы</title>\n" +

" </head>\n" +

" <body>\n" +

" <h1>Заголовок</h1>\n" +

" <!-- Комментарий -->\n" +

" <p>Первый абзац.</p>\n" +

" <p>Второй абзац.</p>\n" +

" <h1>Заголовок 2</h1>\n"+

" </body>\n" +

"</html>";

Document doc = Jsoup.parse(html);

System.out.println("Заголовок: "+doc.title());

System.out.println("Кодировка: "+doc.charset());

Element body = doc.body();

System.out.println("\nbody:\n"+body+"\n");

body.prependText("Текст в начале");

Element img = body.appendElement("img");

img.attr("height", "250");

img.attr("weight", "250");

img.attr("src", "http://kvistrel.su/\_nw/50/86155379.jpg");

body.append("<h1>Конец</h1>");

System.out.println("\n"+body);

Element head = doc.head();

System.out.println("head:\n"+head+"\n");

Elements paragraph = doc.select("p");

System.out.println("\nВсе теги p:\n"+paragraph);

Класс Jsoup.

Раннее класс Jsoup упоминался только в парсинге документа, но кроме этого он определяет следующие методы:

|  |  |
| --- | --- |
| Объявление | Описание |
| Element getElementById (String *id*) | Возвращает элементы по *id*. |
| Elements getElementByTag (String *тег*) | Возвращает элементы по *тэгу*. |
| Element getElementsByClass (String *класс*) | Возвращает элементы по *классу*. |
| Elements getElementsByAttribute(String *атрибут*) | Возвращает элементы по *атрибуту*. |
| Elements siblingElements() | Возвращает братские элементы текущего элемента. |
| Element firstElementSibling() | Возвращает первый братский элемент текущего элемента. |
| Element lastElementSibling() | Возвращает последний братский элемент текущего элемента. |



Проект фонда Apache Software Foundation, имеющий своей целью разработку и поддержку открытого программного обеспечения повторного использования на языке Java, то есть библиотек Java. В более узком смысле Apache Commons — это «большая коллекция маленьких Java-утилит».

Это не единая библиотека, а ряд разных библиотек не связанных между собой. В основном каждый пакет является дополнением из стандартной библиотеки. К примеру, к пакету java.lang есть библиотека org.apache.commons.lang3 и т.д.

Библиотека lang3.

Как ранее говорилось библиотека org.apache.commons.lang3 является логическим продолжением библиотеки java.lang.

Класс ArrayUtils.

Данный утилит содержит методы для работы с массивами. Он определяет ряд методов, которые уже были в других библиотеках. К ним можно отнести indexOf (), lastIndexOf (), reserve (), shuffle (), contains (), isEmpty (), isNotEmpty () и тд.

Благодаря методу isSorted () можно узнать отсортирован ли массив. Также isSameLength (), который сравнивает длину *массив1* и *массив2*. Также есть метод shift (), который сдвигает массив и метод swap (), который заменяет два элемента в заданном массиве.

Он также определяет метод remove (), который удаляет заданный *индекс* в *массиве*. Схожим является метод removeAll (), который удаляет все переданные *индексы*. Аналогичными считаются removeElement () и removeElements (), которые удаляют по значению. Также есть метод removeAllOccurences (), удаляющий все вхождения.

Метод toObject () можно *массив* примитивов превратить в массив в оболочек и противоположный ему toPrimitive (). Также есть метод toMap (), который возвращает *массив* в представлении карты. А также toStringArray (), который обворачивает элементы массива в строки.

Методом sudarray (), который создает новый массив, содержащий элементы между *начальным* и *конечным* индексами. Полезный метод nullToEmpty (), который если переданный *массив*  null вернет пустой массив.

Ниже представлен пример использования данного утилита:

import static org.apache.commons.lang3.ArrayUtils.\*;

public class ArrayUtilsDemo {

public static void main(String[] args) {

int [] array = {1, 3, 5, 7, 9, 9, 9, 11};

System.out.println("Начальный массив: "+Arrays.toString(array));

int [] array1 = {2, 4, 6, 8, 10, 12};

System.out.println("Второй массив: "+Arrays.toString(array1));

array = add(array, 13);

System.out.println("\nadd -> "+Arrays.toString(array));

array = addAll(array, 2, 4, 6);

System.out.println("addAll -> "+Arrays.toString(array));

array = remove(array, 2);

System.out.println("\nremove -> "+Arrays.toString(array));

array = removeAll(array, 4, 5);

System.out.println("removeAll -> "+Arrays.toString(array));

array = removeElement(array, 7);

System.out.println("removeElement -> "+ArrayUtils.toString(array));

array = removeElements(array, 2, 3);

System.out.println("removeElements -> "+Arrays.toString(array));

array = removeAllOccurences(array, 9);

System.out.println("removeAllOccurences -> "+Arrays.toString(array));

shift(array, 2);

System.out.println("\nshift -> "+Arrays.toString(array));

swap(array, 2, 4);

System.out.println("swap -> "+Arrays.toString(array));

reverse(array);

System.out.println("reserve -> "+Arrays.toString(array));

shuffle(array);

System.out.println("shuffle -> "+Arrays.toString(array));

Integer[] arrayInts = toObject(array);

System.out.println("\ntoObject -> "+Arrays.toString(arrayInts));

array = toPrimitive(arrayInts);

System.out.println("toPrimitive -> "+Arrays.toString(array));

System.out.println("\nМассив пуст: "+isEmpty(array));

System.out.println("Массив не пуст: "+isNotEmpty(array));

System.out.println("Массив отсортирован: "+isSorted(array));

System.out.println("Совпадает ли длина: "+isSameLength(array, array));

array = nullToEmpty((int[]) null);

System.out.println("nullToEmpty -> "+Arrays.toString(array));

}

}

Класс AnnotationUtils.

Данный утилит представляет методы для работы с аннотации. Она позволяет сравнить аннотации ( метод equals ()), представить в виде строки ( toString ()), узнать действительно ли это аннотация ( isValidAnnotationMemberType ()) и получить хэш – код (hashCode()).

Пример использования класса:

Annotation firstAnnotation= () -> NotNull.class;

Annotation secondAnnotation = () -> SafeVarargs.class;

boolean equals = AnnotationUtils.equals(

firstAnnotation,

() -> NotNull.class);

System.out.println(equals);

int hashCode = AnnotationUtils.hashCode(secondAnnotation);

System.out.println(hashCode);

boolean isValid = AnnotationUtils.isValidAnnotationMemberType(Integer.class);

System.out.println(isValid);

System.out.println(AnnotationUtils.toString (firstAnnotation));

Перечисление JavaVersion.

Перечисление JavaVersion представляет все версии спецификации Java. Использование данного перечисления:

JavaVersion thisVersion = JavaVersion.valueOf("JAVA\_1\_8");

System.out.println(thisVersion.atLeast(JavaVersion.JAVA\_1\_5)+"\n");

JavaVersion[] values = JavaVersion.values();

for (JavaVersion v: values) {

System.out.println(v.toString());

}

Утилит RandomUtils.

Утилит RandomUtils представляет методы для работы с случайными числами. Он позволяет получить случайное число без явного заведения объекта класса Random. У него есть методы nextInt () и тому подобное. Они также позволяют указать рамки генерации.

Для байтов данный предназначается метод nextBytes (), который возвращает массив с *длиной* случайных чисел.

Использование утилита:

import static org.apache.commons.lang3.RandomUtils.\*;

public class RandomUtilsDemo{

public static void main(String[] args) {

System.out.println("nextInt -> "+nextInt(10, 30));

System.out.println("nextBoolean -> "+nextBoolean());

System.out.println("nextDouble -> "+nextDouble(12.3, 16));

System.out.println("nextFloat -> "+nextFloat(13.4f, 123f));

System.out.println("nextLong -> "+nextLong(5, 31));

byte[] bytes = nextBytes(10);

System.out.println("nextBytes -> "+ Arrays.toString(bytes));

}

}

Класс RandomStringUtils.

Обычный Random не способен генерировать случайные строки. Для apache определяет класс – утилит RandomStringUtils. Ниже перечислены:

|  |  |
| --- | --- |
| static String random(int *длина*) | Возвращает случайную строку с *длиной*. |
| static String randomAlphabetic(int *длина*) | Возвращает случайную строку с *длиной* состоящая из латиницы. |
| static String randomNumeric(int *длина*) | Возвращает случайную строку с *длиной* состоящая их цифр. |
| static String randomAscii (int *длина*) | Возвращает случайную строку с *длиной* состоящая из символов – ascii. |
| static String randomGraph (int *длина*) | Возвращает случайную строку с *длиной* состоящая из символов – ascii, но без пробелов и управляющих символов. |
| static String randomPrint (int *длина*) | В отличии от randomGraph (), может содержать пробелы. |

Кроме выше показанных они имеют другие перегруженные методы. К примеру можно указать минимальную и максимальную длину из которых выберется случайная длина.

Использование:

import static org.apache.commons.lang3.RandomStringUtils.\*;

public class StringRandomDemo {

public static void main(String[] args) {

System.out.println("random -> "+random(12));

System.out.println("randomAscii -> "+randomAscii(1, 5));

System.out.println("randomNumeric -> "+randomNumeric(12));

System.out.println("randomGraph -> "+randomGraph(12));

System.out.println("randomAlphabetic -> "+randomAlphabetic(12));

System.out.println("randomPrint -> "+randomPrint(12));

}

}

Класс ClassPathUtils.

Утилит StringUtils.

Данный класс представляет ряд методов для работы с методами. Они позволяют проверить строку на наличие подстроки, начало и конец строки и т.д.; добавить строку в начало, либо в конец и т.д.

Метод abbreviate () укорачивает *строку* на *длину*, но три последние символа будут выделены под знак троеточия. Методы remove (), removeStart () и removeEnd () удаляют из *строки* переданную *часть*.

Чтобы удалить пробелы с начала и с конца следует воспользоваться методами trim ()/ strip (). Методы isAlpha () узнает состоит ли строка из символов Unicode, isNumeric () узнает состоит ли строка из цифр, isWhitespace () узнает состоит ли строка из пробела, isAsciiPrintable () узнает состоит ли строка из символов ascii.

Метод isBlank () проверяет не состоит ли строка из "". Метод isMixedCase () узнает не состоит ли строка из разных слов. Чтобы получить правые или левые символы от *строки* следует воспользоваться методами left (), right () и указать *количество* символов.

Чтобы после *количество* символов переместить в начало следует воспользоваться метод rotate (), также можно указать обратное количество и сделать наоборот.

Также можно поменять регистр символов на обратный с помощью метода swapCase (). Чтоб получить все цифры из *строки* в виде строки следует воспользоваться методом getDigits ().

В StringUtils есть продвинутый join (). Он позволяет склеить список, массив Object, для примитивов и т.д. Еще определяется ряд методов для проверки массива CharSequence. К ним можно отнести isAllEmpty (), isAnyEmpty (), isAllBlank () и isAnyBlank (). При помощи метода firstNonBlank () можно получить первое значение в массиве, который не является пустым "", null или только в виде пробела; firstNonEmpty () первое значение в массиве, который не пуст.

Использование утилита:

import static org.apache.commons.lang3.StringUtils.\*;

public class StringUtilsExamples {

public static void main(String[] args) {

String str = " СтроКа 124 а 734 Java ";

System.out.println(str);

str = strip(str);

System.out.println("strip -> "+str);

String right = right(str, 2);

System.out.println("right -> "+right);

String left = left(str, 2);

System.out.println("left -> "+left);

str = removeEnd(str, "va");

System.out.println("removeEnd -> "+str);

String abbreviate = abbreviate(str, 7);

System.out.println("abbreviate -> "+abbreviate);

String numbers = getDigits(str);

System.out.println("getDigits -> "+numbers);

String rotate = rotate(str, 2);

System.out.println("rotate -> "+rotate);

String swapCase = swapCase(str);

System.out.println("swapCase -> "+swapCase);

System.out.println("\n=====================\n");

System.out.println("\n=====================\n");

System.out.println("isEmpty -> "+isEmpty(str));

System.out.println("isBlank -> "+isBlank(str));

System.out.println("isMixedCase -> "+isMixedCase(str));

System.out.println("isAsciiPrintable -> "+isAsciiPrintable(str));

System.out.println("isAlpha -> "+isAlpha(str));

System.out.println("isNumeric -> "+isNumeric(str));

System.out.println("isWhitespace -> "+isWhitespace(str));

System.out.println("isAllLowerCase -> "+isAllLowerCase(str));

System.out.println("\n=====================\n");

String [] array = {null, "af", "apache", "java", "3.8", " "};

System.out.println("Начальный массив: "+Arrays.toString(array));

System.out.println("isAllBlank -> "+isAllBlank(array));

System.out.println("isAllEmpty -> "+isAllEmpty(array));

System.out.println("firstNonBlank -> "+firstNonBlank(array));

System.out.println("firstNonEmpty -> "+firstNonEmpty(array));

System.out.println("isAnyBlank -> "+isAnyBlank(array));

System.out.println("isAnyEmpty -> "+isAnyEmpty(array));

System.out.println("\n=====================\n");

String join = join(array, ' ');

System.out.println("join(object [], разделитель) -> "+join);

List<String> list = Arrays.asList(array);

join = join(list, "; ");

System.out.println("join(список, разделитель) -> "+join);

join = join(new int[]{1, 2, 3, 4, 5, 6, 7, 8}, '>');

System.out.println("join(число, разделитель) -> "+join);

}

}

Утилит CharUtils.

Методы для работы с символами можно найти в классе CharUtils. Чтобы проверить то, что *символ* относится ASCII можно методом isAscii (); является буквой и относится к ASCII можно проверить метод isAsciiAlpha(), схожими являются isAsciiAlphaLower() и isAsciiAlphaUpper (), но для них важен регистр.

Чтоб проверить является ли *символ* контрольным символом можно методом isAsciiControl (); чтобы проверить не цифра ли *символ* можно использовать isAsciiNumeric ().

Метод isAsciiAlphanumeric () проверяет, что *символ* число или буква. И чтобы узнать будет ли символ отображен в печати isAsciiPrintable ().

Метод compare () сравнивает два символа. Также утилит определяет ряд методов для конвертации символа с строка и на оборот и т.д.

import static org.apache.commons.lang3.CharUtils.\*;

public class CharUtilsExample {

public static void main(String[] args) {

System.out.println("€ это символ из ASCII: "+

isAscii('€'));

System.out.println("g это символ из ASCII с низким регистром: "+

isAsciiAlphaLower('g'));

System.out.println("c это символ из ASCII с верхним регистром: "+

isAsciiAlphaUpper('c'));

System.out.println("\\n это контрольный символ: "+

isAsciiControl('\n'));

System.out.println("$ это символ или число: "+

isAsciiAlphanumeric('$'));

System.out.println("1 это число: "+

isAsciiNumeric('1'));

System.out.println("\\f это символ для печати: "+

isAsciiPrintable('\f')+"\n");

int compare = compare('f', '5');

if (compare == 1)

System.out.println("f больше 5");

else if(compare == 0)

System.out.println("f равно 5");

else

System.out.println("f меньше 5");

}

}

Утилит RegExUtils.

Класс RegExUtils содержит методы для работы с регулярными выражения. Чтобы удалить все подстроки совпадающие по *шаблону* следует воспользоваться removeAll (), removeFirst () или removePattern ().Также есть возможность заменить все подстроки, совпадающие с *шаблон* на *строку* методами replaceAll (), replaceFirst () или replacePattern ().

import java.util.regex.Pattern;

import static org.apache.commons.lang3.RegExUtils.\*;

public class RegExUtilsDemo{

public static void main(String[] args) {

//launch(args);

Pattern pattern = Pattern.compile("[Д-д]о Java [1-7]\\s");

String text = "До Java 5 не было даже обобщений и до Java 7 не было jni!";

System.out.println("Изначальный текст: "+text+"\n");

String str = replaceAll(text, pattern, "раньше");

System.out.println(str);

str = replaceFirst(text, pattern, "Раньше");

System.out.println(str+"\n");

str = removeAll(text, pattern);

System.out.println(str.trim());

str = removeFirst(text, pattern);

System.out.println(str.trim()+"\n");

str = removePattern(text, pattern.pattern());

System.out.println(str);

str = replacePattern(text, text, pattern.pattern());

System.out.println(str.trim());

}

}

Утилит ObjectUtils.

Класс ObjectUtils представляет методы для защиты от null, копирования, сравнения, поиска и т.д. Он предназначен для работы с объектами.

Метод firstNotNull () возвращает первый не null объект из переданного *массива*, allNotNull () проверяет все ли значения *массива* null, anyNonNull () проверяет нет ли null в *массиве*. Также метод defaultifNull (), который если *объект* null, тогда вернет *значение\_по\_умолчанию*.

Методом max (), min () можно получить наименьшие и наибольшие значение *массива*; medium () возвращает среднее значение, а mode () возвращает наиболее часто встречающие значение *массива*.

При помощи метод cloneIfPossible () можно безопасно клонировать *объект*; еще есть метод identityToString() получает toString (), который будет создан, объект не переопределит toString ().

Использование данного утилита:

import static org.apache.commons.lang3.ObjectUtils.\*;

public class ObjectUtilsDemo {

public static void main(String[] args) {

Object[] array = new Object[]{

12, null,

Arrays.asList(new File(""), null)

};

System.out.println("Начальный массив: "+Arrays.toString(array));

System.out.println("allNotNull -> "+allNotNull(array));

System.out.println("anyNotNull -> "+anyNotNull(array));

System.out.println("firstNonNull -> "+firstNonNull(array));

System.out.println("\n===================================\n");

String [] array1 = {"1", "12", "123", "123", "123", "1234", "12345"};

System.out.println("Начальный массив: "+Arrays.toString(array1));

String min = min(array1);

System.out.println("min -> "+min);

String max = max(array1);

System.out.println("max -> "+max);

String median = median(array1);

System.out.println("media -> "+median);

String mode = mode(array1);

System.out.println("mode -> "+mode);

System.out.println("notEquals -> "+notEqual (array, array1));

System.out.println("\n===================================\n");

File f = null;

f = defaultIfNull(f, new File("java.txt"));

System.out.println(f);

File copy = cloneIfPossible(f);

System.out.println(copy);

System.out.println("compare -> "+compare(f, copy));

String identity = identityToString(f);

System.out.println("identityToString -> "+identity);

}

}

Пакет math.

По переводу понятно то, что пакет org.apache.commons.lang3.math предназначен для математических классов. У него определено всего лишь три класса:

* Fraction для долей.
* IEEE754rUtils для чисел с плавающей точкой.
* NumberUtils определяет общие методы для работы с числами.

Класс IEEE754rUtils.

Данный класс предлагает всего два: min () и max (). Каждый из них перегружается для float и double. Первая версия выбирает число из двух переданных. Вторая выбирает уже из трех, а третья из массива.

float f1 = 2124.1234f;

float f2 = 463.35754f;

float f3 = 75.1357f;

System.out.println(min(f1, f2));

System.out.println(min(new float[]{f1, f2, f3}));

System.out.println(max(f1, f2));

System.out.println(max(new float[]{f1, f2, f3}));

double d1 = 3562.24;

double d2 = 134623.13;

double d3 = 15314.1463;

System.out.println(min(d1, d2));

System.out.println(min(new double[]{d1, d2, d3}));

System.out.println(max(d1, d2));

System.out.println(max(new double[]{d1, d2, d3}));

Утилит LocaleUtils.

Класс представляет методы для работы с Locale. Чтобы получить список или множество стран говорящих на *языке* следует воспользоваться метод countriesByLanguage () и наоборот languagesByCountry ().

Также можно объект Locale на *строке* в виде домена высшего уровня. Чтобы получить список множество состоящий из всех локалей можно методом availableLocaleList () или множество availableLocaleSet().

Методом localeLookupList () получает список локалей для поиска при выполнении поиска по локали. Также можно задать *значение по умолчанию*.

import org.apache.commons.lang3.LocaleUtils;

import java.util.\*;

public class LocaleUtilsDemo {

public static void main(String[] args) {

System.out.println("countriesByLanguage -> ");

List<Locale> locales = LocaleUtils.countriesByLanguage("en");

for (Locale l : locales) {

System.out.println(" - "+l.getCountry()+", "+l.getDisplayName(Locale.getDefault()));

}

System.out.println("\nlanguagesByCountry -> ");

List<Locale> languages = LocaleUtils.languagesByCountry("RU");

for (Locale l: languages) {

System.out.println(" - "+l.getCountry()+", "+l.getDisplayName(Locale.getDefault()));

}

Locale russia = LocaleUtils.toLocale("ru");

System.out.println("\ntoLocale -> "+russia);

System.out.println("availableLocaleList -> ");

locales = LocaleUtils.availableLocaleList();

for (Locale l : locales) {

System.out.println(l.getDisplayCountry(Locale.getDefault()));

}

System.out.println("\nlocaleLookupList -> ");

List<Locale> list = LocaleUtils.localeLookupList(Locale.ENGLISH);

for (Locale l: list) {

System.out.println(l);

}

}

}

Утилит NumberUtils.

Утилит NumberUtils предлагает ряд методов для работы с примитивами. К ним можно отнести метод compare (), перегружается для каждого типа и проверяет больше ли, меньше, либо равно. Также есть методы min () и max ().

Для удобного парсинга есть методы createInteger (), createFloat () и т.д. Также методам isCreatable () можно проверить состоит ли *строка* из цифр, а методом isDigits () проверить состоит ли строка из цифр.

Схожими являются toDouble (), toLong () и т.д., которые в отличии от выше описанных не генерируют, а возвращают значение по умолчанию. Вторые версии данных методов также позволяют указать *значение по умолчанию*.

int i1 = 124;

int i2 = 64;

int i3 = 853;

System.out.println("i1 "+i1+"; i2 "+i2+"; i3 "+i3);

int compare = compare(i1, i3);

if(compare == 0)

System.out.println("i1 равен i3");

else if(compare == 1)

System.out.println("i1 больше i3");

else

System.out.println("i1 меньше i3");

int num = toInt("no number", 31);

System.out.println("toInt -> "+num);

System.out.println("min -> "+min(i1, i2, i3));

System.out.println("max ->"+max(i1, i2, i3));

Класс Fraction.

Полезный класс Fraction инкапсулирует дроби. Он определяет разные способы создания и работы с дробями. Сразу стоит сказать, что в основу класса лег int и по этому проблемы с переполнением.

Для создания объекта используется фабричный метод getFraction (). Он имеет следующие версии:

Fraction getFraction (String *дробь*)

Fraction getFraction (int *знаменатель*, *числитель*)

Fraction getFraction (int *целое\_число*, int *знаменатель*, *числитель*)

Fraction getFraction (double *число*)

Для того чтоб перемножить две дроби следует воспользоваться методом multiplyBy(), поделить дроби divideBy (), сложить две дроби метод add () и чтобы вычесть subtract ().

Метод reduce () сокращает дробь, а pow () наоборот. Метод invert () переворачивает дробь, а метод negate () возвращает такую же дробь, но с отрицательным знаменателем.

Получить дробь в десятичной форме можно методом doubleValue () или floatValue (). Также можно получить отдельную часть дроби методами getNumerator (),getDenominator() и getProperWhole().

Пример дроби:

import org.apache.commons.lang3.math.Fraction;

import static org.apache.commons.lang3.math.Fraction.\*;

public class FractionExample{

public static void main(String[] args) {

Fraction f1 = getFraction(4, 8);

Fraction f2 = getFraction(1.25);

Fraction f3 = getFraction("2 1/7");

System.out.println("f1: "+f1);

System.out.println("f2: "+f2);

System.out.println("f3: "+f3);

System.out.println("\n=======================\n");

System.out.println("divideBy -> "+f1.divideBy(f3));

System.out.println("multiplyBy -> "+f1.multiplyBy(f3));

System.out.println("invert -> "+f1.invert());

System.out.println("pow -> "+f1.pow(2));

System.out.println("reduce -> "+f1.reduce());

System.out.println("add -> "+f1.add(f2));

System.out.println("subtract -> "+f1.subtract(f2));

System.out.println("negate -> "+f1.negate());

System.out.println("\n=======================\n");

System.out.println(f3);

System.out.println(f3.doubleValue());

System.out.println(f3.getDenominator());

System.out.println(f3.getNumerator());

System.out.println(f3.getProperWhole());

}

}

Пакет mutable.

Пакет mutable представляет типизированные оболочки примитивов и Object. Но нужно обратить внимание на то, что классы – оболочки в mutable не расширяют классы оболочки из java, за исключением ObjectMutable - это происходит не явно.

Класс MutableObject.

Класс MutableObject параметризован типом T. Он определяет четыре метода. Первый equals (), второй hashCode (), третий setValue (), который устанавливает *значение*, а getValue () возвращает значение. Также есть метод toString ().

MutableObject<String> obj = new MutableObject<>("MutableObject Example");

String lastValue = obj.getValue();

System.out.println("Значение "+lastValue);

System.out.println("хэш – код "+obj.hashCode());

obj.setValue("Java");

System.out.println("Новое значение "+obj.getValue());

System.out.println("хэш – код "+obj.hashCode());

Класс BooleanMutable.

Класс BooleanMutable является оболочкой для boolean. Но класс не расширяет Boolean. Он определяет ряд удобных методов.

Метод isFalse () можно проверить является ли содержимое false, также есть метод isTrue (). Чтобы получить содержимое можно получить методом getValue (), а также присвоить setValue ().

Также можно установить значение true методом setTrue () и false – setFalse (). Чтобы получить содержимое в виде примитива следует использовать метод booleanValue (), а для того чтоб получить оболочку toBoolean ().

MutableBoolean bool = new MutableBoolean(true);

System.out.println("Значение "+bool.getValue());

System.out.println("Это лож: "+bool.isFalse());

System.out.println("Это правда: "+bool.isTrue());

bool.setFalse();

System.out.println("\nНовое значение: "+bool.getValue());

System.out.println("Это лож: "+bool.isFalse());

System.out.println("Это правда: "+bool.isTrue());

int res = bool.compareTo(new MutableBoolean(true));

if(res == 1)

System.out.println("false больше true");

else if(res == 0)

System.out.println("false равен true");

else

System.out.println(“false меньше true”);

Класс MutableInt.

Класс MutableInt представляет оболочку для int. Чтоб понять его методы следует посмотреть на название. К примеру getAndAdd (*число*) – он сначала вернет число, а далее добавить в него *число*.

Также он представляет удобные методы преобразования int в short, long и float: shortValue (), longValue () и floatValue (). Чтобы получить оболочку следует использовать toInteger ().

MutableInt i = new MutableInt("123");

System.out.println("Значение " + i.getValue());

System.out.println("addAndGet -> " + i.addAndGet(2));

System.out.println("Значение " + i.getValue());

System.out.println("decrementAndGet ->" + i.decrementAndGet());

System.out.println("Значение " + i.getValue());

System.out.println("getAndAdd -> " + i.getAndAdd(12.3));

System.out.println("Значение " + i);

i.increment();

System.out.println("Значение " + i);

i.subtract(13);

System.out.println("Значение " + i);

int compare = i.compareTo(new MutableInt(23));

if (compare == 1)

System.out.println("i больше 23");

else if(compare == 0)

System.out.println("i равно 23");

else

System.out.println("i меньше 23");

Другие классы пакета mutable.

Кроме выше описанных классов существует ряд оболочек для других примитивов. Все они схожи с MutableInt. Они перечислены ниже:

|  |  |
| --- | --- |
| MutableShort | Представляет оболочку для примитивов типа short. |
| MutableByte | Представляет оболочку для примитивов типа byte. |
| MutableLong | Представляет оболочку для примитивов типа long. |
| MutableDouble | Представляет оболочку для примитивов типа double. |
| MutableFloat | Представляет оболочку для примитивов типа float. |

Robot rd=com.sun.glass.ui.Application.GetApplication().createRobot()

далить объект самостоятельно нельзя так, как это работа сборщика мусора.

Но можно указать сборщику мусора то, что ему следует удалить объект присвоив экземпляру значение `null`, т.е. убрав ссылки на него.

\*Конструктор\* в [tag:ооп] - это метод, это конструирующий объект с определенными полями, но с возможностью изменений, иными словами инициализатор объекта.

Java

----------

В [tag:java] конструктор представляет собой метод с именем класса. Конструктор также можно перегружать.

Нужно учитывать, что не следует вставлять сложную логику в конструктор, а наоборот легкую.

Пример:

public class Point{

int i, c;

Point(int i, int c){

this.i=i;

this.c=c;

}

@Ovveride

public String toString (){

return "Объект{ "+i+":"+c+"}";

}

}