Hot keys

ctrl+P - внутри скобок показывает конструкторы классы

alt+insert - показывает контекстное меню

String txt = JOptionPane.showInputDialog(""); - выводит диалоговое окно

JOptionPane.showMessageDialog(null, txt); - выводит введенное в диалоговом окне

import com.company.switvh\_learn; - подключение библиотеки

import java.util.Scanner;

String j = (x % 2 == 0) ? "the number is even" : "the number is odd";

System.out.println(j); - тернарная операция

Scanner read = new Scanner(System.in);

int x = read.nextInt(); - ввод с клавиатуры, надо обращаться к переменной

read.переменная

int x = new Scanner(System.in).nextInt(); - ввод с клавиатуры, но для каждого ввода надо

создавать новею переменную

int[] array = new int[5];

Arrays\_class[] mass = new Arrays\_class[3];

int[] array2 = {1, 2, 3, 4};

Arrays\_class box1 = new Arrays\_class();

Arrays\_class box2 = new Arrays\_class();

Arrays\_class box3 = new Arrays\_class();

Arrays\_class[] box = {box1, box2, box3}; - создание массивов

for (int i = 0; i < array2.length; i++) - длина массива .length, скобочки не нужны

int [][] mas1 = new int[2][3];

int [][] mas2 = {{1, 2}, {3, 4}}; - массив массивов

int [][][] mas3 = {{{1, 2}, {3, 4}},{{5, 6}, {7, 8}}};

System.out.println(mas3[0][1][1]);

switch (x){

case "1":

System.out.println("it's a 1");

break;

case "2":

System.out.println("it's a 2");

break;

default:

System.out.println("give a fuck");

} - цикл switch-case, break нужен так как выполняются все опреаторы

после сработавшего случая,

используются только константы и String,

необходимо слово final для констант и String.

final int VALUE1 = 10, VALUE2 = 25, VALUE3 = 48, VALUE = 48;

switch (VALUE) {

case VALUE1:

System.out.println("10");

break;

case VALUE2:

System.out.println("25");

break;

case VALUE3:

System.out.println("48");

break;

default:

System.out.println("Default");

}

for (int x = 0; x++ <= 5; ) или for (int x = 0; x <= 5; х++)

System.out.print(x + " "); - цикл for

for(<Тип элемента> <Имя переменной, куда будет записан очередной элемент> : <Название массива>) {

// Тело цикла

} - цикл for

public void printAllElements(String[] stringArray) {

for(String s : stringArray) {

System.out.println(s);

}

} - цикл for

do {

System.out.printf("x is %s\n", x);

x++;

} while (x <= 10); - цикл do-while

while (x <= 10){

System.out.printf("\t\tx is %s\n", x);

x++;

} - цикл while

Arrays.toString() - превращает массив в строку

Arrays.copyOf - копирует значение из массива nums в новый массив

str.charAt(i) - позвращает символ под индексом i переменной типа String

int i = Integer.parseInt("10") - перенвод строки в тип int, аналогично для других типов

i = Integer.parseInt("00000001", 2) - можно и так, есл есть основание

x.toString() - перевод числа в строку

Integer.toBinaryString(i) - перевод int в бинарное число типа String

System.out.println(x.toString()); - Преобразовать число x в строку

System.out.println(Integer.toString(12)); - Преобразовать int в string

System.out.println(Integer.toBinaryString(Integer.parseInt(x, 2) - Integer.parseInt(y, 2))); -пример

System.out.println(x instanceof int); - сравнивает x с типом Integer, вместо int можно написать переменную

какого-то типа

System.getProperty("java.version") - данные ПО, см. инфо getProperty

Модификаторы

private, default, protected, public

private можно применять только к вложенному классу. Использовать этот вложенный private класс можно только в методах

основного класса. Это также относится к конструкторам, методам, полям

private class CommonMain {

default данный модификатор не пишется, то есть не надо писать это слова, это означает наш класс и его сотовляющие

доступны только в нашем пакете

class CommonMain {

protected может создаваться только во вложенном классе он работает почти так же как и default

Наследование

public class Ostrich extends Birds{ - класс Ostrich наследует(extends) свойства Birds

}

Наследоване передают свойства и методы класса Birds классу Ostrich. Также класс Ostrich имеет свои свойства и методы,

т.е. у объекта ostrich будут свойства классов Ostich и Birds. Множественного наследования не существует.

Ключевое слово super, this

public class Birds {

private String name;

private Wings wings;

public Birds(String name, Wings wings) {

this.name = name;

this.wings = wings;

}

}

public class Ostrich extends Birds{

public Ostrich(String name, Wings wings) {

super(name, wings); - super должно идти первой строчкой, обращение к родительскому классу

}

}

super() - текущий экземпляр родительского класса, нестатическая переменная

this()- Внутри класса для вызова своего конструктора без аргументов используется, нестатическая перменная

И this, и super могут использоваться внутри конструкторов для вызова других конструкторов по цепочке, нпр.,

this() и super() вызывают конструктор без аргументов наследующего и родительского классов соответственно.

Внутри конструктора this и super должны стоять выше всех других выражений, в самом начале, иначе компилятор выдаст сообщение об ошибке.

Из чего следует, что в одном конструкторе не может быть одновременно и this(), и super().

Каждый конструктор при отсутствии явных вызовов других конструкторов неявно вызывает с помощью super() конструктор без

аргументов родительского класса, при этом у вас всегда остается возможность явно вызвать любой другой конструктор

с помощью либо this(), либо super().

Полиморфизм

- это свойство позволяющее с помощья одного интерфейса обращаться к общему классу дкйствий

- это способность определять верстсию переопределенного метода в зависимости о типа объекта

1. Статический полиморфизм- появлется перегруженный метод, но не в родительском классе. На примере конструктора

public class Birds {

private String name;

public void walk(){

System.out.println("walking");

}

}

public class Ostrich extends Birds{

public void walk(String arg){

System.out.println("I'm walking" + arg);

}

}

2. Динамический полиморфизм- появляется переоопределение метода, т.к. название метода и входные параметры полностью совпадают.

На примере конструктора

public class Birds {

private String name;

public void walk(){

System.out.println("walk");

}

}

public class Ostrich extends Birds{

@Override - значит, что метод переопределен

public void walk() {

System.out.println("ostrich walk");

}

}

Абстрактные классы и методы

Абстракция - это принцип ООП, согласно которому при проектировании классов и создании объектов необходимо выделять только

главные свойства сущности, и отбрасывать второстепенные.

1.экземпляр абстрактного класса создать нельзя

2.класс является абстрактным, если хотя бы один из его методов является абстрактным

3.словом abstract помечаются и классы, и методы внутри этих классов

4.абстрактные методы необходимо переопределять

public abstract class Weapon {

}

public class AK47 extends Weapon {

}

public class Main {

public static void main(String[] str){

AK47 ak47 = new AK47();

Weapon weapon = new AK47(); -можно так, но будут ограничены методы, если они не переопределены

}

}

Интерефейсы

-невозможно создать экземпляр этого класса

-интерфейсы помогают создавать многоуровневюу структуру

Все поля автоматически public static final, даже если это не указывать, все методы public abstract

public class Mi8 extends Helicopter implements VerticalTakeOff{ -implements реализовать

@Override

public void fly() {

System.out.println("Mi8 is flying");

}

@Override

public void verticalTakeOff() {

System.out.println("Mi8 vertical");

}

}

implements

- реализовывать интерфейсы можно неограниченное количество(implements ) в отличие от extends.

- интерфейсы необходимо потом перопределять

Клонирование

-создаем точную копию выбранного объекта, так как мы передаем ссылку в определенный метод мы продолжаем работать

с этим же объектом

- если мы хотим клонировать в другом пакете, надо переопределить метод clone(), который находится в классе Object

alt+insert, override, clone():Object, protected меняем на public

- добавляем implements Cloneable, Cloneable явлеется интерфейсом меткой и он просто помечает класс, переопредлеять ниче не нужно

public class DollySheet implements Cloneable{

private String name;

public String getName() {

return name;

}

@Override

public Object clone() throws CloneNotSupportedException {

return super.clone();

}

public void setName(String name) {

this.name = name;

}

}

- при клонировании необходимо приводить клонированные данные к классу переменной, где эти данные будут хранится

- необходимо при клонирование обраюатывать ошибки

public class CommonMain {

public static void main(String[] str){

DollySheet dollySheet = new DollySheet();

DollySheet dollySheet2 = foo(dollySheet);

dollySheet.setName("Dolly");

dollySheet2.setName("Dolly clone");

System.out.println(dollySheet.getName());

System.out.println(dollySheet2.getName());

}

public static DollySheet foo(DollySheet dollySheet2){

DollySheet dollySheet\_clone = null;

try {

dollySheet\_clone = (DollySheet)dollySheet2.clone();

} catch (CloneNotSupportedException e) {

e.printStackTrace();

}

return dollySheet\_clone;

}

}

Если при клонировании объекта полностью(глубокое клонирование), необходимо, чтобы все поля(созданные кмной классы например)

реализовывали интерфейс Cloneable

Строки

в пакете javalang есть несколько встроенных классов для строк String, StringBuilder, StringBuffer.

charAt() - возвращает символ строки

System.out.println(str1.charAt(0));

codePointAt() - возвращает кодировку ascii данного символа

System.out.println(str1.codePointAt(5));

str1.compareTo(str2) - сравнивает строки, включая размер шрифт и возращает 0, если они одинокавые

System.out.println(str1.compareTo(str2));

str1.compareToIgnoreCase(str2) - сравнивает строки, не включая размер шрифт и возращает 0, если они одинокавые

System.out.println(str1.compareToIgnoreCase(str2));

str1.concat(str2) - объединяет две строки, аналогичен str1 + str2, но строка не изменяется

System.out.println(str1.concat(str2));

str1.equals(str2) - сравнивает строки, выдает true or false

System.out.println(str1.equals(str2));

str1.equalsIgnoreCase(str2) - сравнивает строки, выдает true or false, игнорирует регистр

System.out.println(str1.equalsIgnoreCase(str2));

str1.hashCode() - возвращает hash code объекта

System.out.println(str1.hashCode());

str1.indexOf(32) - возвращает номер символа, которые совпадает символом ASCII, -1 если не встретился символ

System.out.println(str1.indexOf(32));

str1.isEmpty() - возвращает true, если строка пустая

System.out.println(str1.isEmpty());

str1.length() - возвращает длинну строки

System.out.println(str1.length());

split() - разделяет строку на строки и записывать надо в массив, в скобках символ или символы разделитель

String str1 = new String("hot Java 1");

String[] s = str1.split(" ");

for (String x : s)

System.out.println(x);

str1.substring(2, 5) - возвращает строку от 2(включая) символа до 5(не включая) символа

System.out.println(str1.substring(2, 5));

toCharArray() - разбивает строку на массив символов

char[] arr = str1.toCharArray();

for (char i : arr)

System.out.println(i);

str1.toLowerCase() - все символы в маленьктй регистр

str1.toUpperCase() - все символы в большой регистр

trim() - обрезает пробелы в начале и в конце строки

String.valueOf() - приводит введеный тип к String

System.out.println(String.valueOf(65));

String.copyValueOf() - преобразует char[] к String

StringBuilder / StringBuffer

public class CommonMain {

public static void main(String[] arg){

String str = new String();

StringBuilder stringBuilder = new StringBuilder();

StringBuffer stringBuffer = new StringBuffer();

System.out.println(stringBuilder.capacity());

}

}

-объект класса StringBuilder / StringBuffer можно преобразовать к String

- StringBuilder / StringBuffer используются, когда складываются много строк

-метод toString()- метод класса Object, приводит StringBuilder / StringBuffer к классу String

-StringBuilder быстрее, но он не потокобезопасен в отличие от StringBuffer

- StringBuilder() конструтор по умолчанию выделяет памяитт под 16 символов

можно проверить System.out.println(stringBuilder.capacity());

stringBuilder.ensureCapacity(17) - задает минимальную вместимость, т.е. при выводе stringBuilder.capacity()

размер может быть больше или равен написанному

stringBuilder.append(" ho t ") - к stringBuilder прибавляется строка/int/double/etc. , но

меняется строка stringBuilder, в отличие от concat()

stringBuilder.setLength(15) - задает размер буфера, типо ensureCapacity, но хз

stringBuilder.insert(1,"h") - добавляет символ h после 1 символа

stringBuilder.delete(1, 4) - удаляет символы с 1 по 4

stringBuilder.deleteCharAt(2) - удаляет определенный символ

stringBuilder.reverse() - переворачивает строку

Generic / Параметризация / Обобщения

public class Robot<T> { - <T> - параметр, может быть любая буква или

private Body body;

private T head;

public Robot(Body body, T head) {

this.body = body;

this.head = head;

}

public Body getBody() {

return body;

}

public void setBody(Body body) {

this.body = body;

}

public T getHead() {

return head;

}

public void setHead(T head) {

this.head = head;

}

}

public class Robot<T1, T2> { - или несколько букв, если несколько параметров

private T2 body;

private T1 head;

}

public class CommonMain {

public static void main(String[] arg){

Body body = new Body();

SmallHead smallHead = new SmallHead();

MediumHead mediumHead = new MediumHead();

BigHead bigHead = new BigHead();

Leg leg = new Leg();

Robot<SmallHead> robot = new Robot<SmallHead>(body, smallHead); - <SmallHead> становится <T>

при вызове этого класса

Robot<BigHead> robot1 = new Robot<BigHead>(body, bigHead);

robot.getHead().burn(); - у Bighead, SmallHead свои burn()

robot1.getHead().burn();

robot1 = robot; - так делать нельзя, будет ошибка

Robot robot2 = new Robot(body, bigHead); - создается пустой экземпляр, нет методов Bighead,

но ему можно плставить значение полного экземпляра

robot2 = robot

}

}

<T>- при компиляции заменяется на переднный в <> класс, не надо дохуя раз описывать для разных комбинаций

public class Robot<T extends Head> { - extends Head ограничивает классы, чтобы вместо головы

не предать ногу

private Body body;

private T head;

public class Robot<T> {

private Body body;

private T head;

public <T1, T2 extends Head> T2 foo(T1 a, T2 b){

T1 leg = new T1(); - так делать нельзя, т.к. не понятно какой тип имеет T1

a + b; - нельзя, можем использовать только методы класса Object

return b;

}

public class CommonMain {

public static void main(String[] arg){

Body body = new Body();

SmallHead smallHead = new SmallHead();

MediumHead mediumHead = new MediumHead();

BigHead bigHead = new BigHead();

Leg leg = new Leg();

Robot robot = new Robot(body, smallHead);

robot.<SmallHead, MediumHead>foo(smallHead, mediumHead); - передача параметров в метод

}

}

public class Robot <T extends Head>{

private Body body;

private T head;

public void foo(Robot <?> obj){ - если не знаем класс

return;

}

public Robot(Body body, T head) {

this.body = body;

this.head = head;

}

public class CommonMain {

public static void main(String[] arg){

Body body = new Body();

SmallHead smallHead = new SmallHead();

MediumHead mediumHead = new MediumHead();

BigHead bigHead = new BigHead();

Leg leg = new Leg();

Robot<SmallHead> robot = new Robot(body, smallHead);

Robot<MediumHead> robot1 = new Robot<MediumHead>(body, mediumHead);

robot.foo(robot1);

}

}

Исключения, ошибки и их иерархия. Блоки try, catch, finally

-при использовании try catch программа продолжает работать, если возникает ошибка

public class Abc {

public void show(){

}

}

public class ErrorExample {

private Abc abc;

public void foo(){

// System.out.println(1/0);

// int arr[] = {1, 2, 3};

// arr[5] = 5;

try

{

abc.show();

} catch (NullPointerException e) {

System.err.println("division by 0"); - System.err.println значит, что выводимая

информация будет красного цвета

e.printStackTrace(); - выведет сообщение об ошибке, почти такоеже как при

ошибке при отсутствии блока обработки ошибок

}

}

}

public class CommonMain {

public static void main(String[] arg){

ErrorExample errorExample = new ErrorExample();

errorExample.foo();

}

}

-проще определенную часть наследнеиков RuntimeExeption обраюатывать блоком if, because try-catch

is very resourse-intensive process

public class ErrorExample {

private Abc abc;

public void foo(){

// System.out.println(1/0);

// int arr[] = {1, 2, 3};

// arr[5] = 5;

if (abc == null)

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

else {

abc.show();

}

-handling several exeption, try-catch will end, when there is a first exception

try

{

abc.show();

System.out.println(1/0);

} catch (NullPointerException e) {

e.printStackTrace();

} catch (ArithmeticException ex){

ex.printStackTrace();

}

System.out.println("\ntry");

}

}

-or

try

{

abc.show();

System.out.println(1/0);

} catch (NullPointerException | ArithmeticException e) {

e.printStackTrace();

}

-rarely we have to add the throws and class of exception

public class ErrorExample {

private Abc abc;

public void foo() throws FileNotFoundException {

FileInputStream fileInputStream = new FileInputStream("");

}

}

-finally needs to evecute action, if any exception have been found by try-catch

public class ErrorExample {

private Abc abc;

public void foo() {

FileInputStream fileInputStream = null;

// Throwable // - hover the cursor over Throwable and press ctrl+H

there is will be hierarchy of exception

try {

fileInputStream = new FileInputStream("");

} catch (FileNotFoundException e){

e.printStackTrace();

} finally { - usually finally closes the input/output strems

try {

fileInputStream.close();

} catch (IOException e){

e.printStackTrace();

}

}

}

}

Создание собственных исключений. Оператор throw

//метод считывает строку с клавиатуры

public String input() throws MyException {//предупреждаем с помощью throws,

// что метод может выбросить исключение MyException

BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

String s = null;

//в блок try заключаем код, в котором может произойти исключение, в данном

// случае компилятор нам подсказывает, что метод readLine() класса

// BufferedReader может выбросить исключение ввода/вывода

try {

s = reader.readLine();

// в блок catch заключаем код по обработке исключения IOException

} catch (IOException e) {

System.out.println(e.getMessage());

// в блоке finally закрываем поток чтения

} finally {

// при закрытии потока тоже возможно исключение, например, если он не был открыт, поэтому “оборачиваем” код в блок try

try {

reader.close();

// пишем обработку исключения при закрытии потока чтения

} catch (IOException e) {

System.out.println(e.getMessage());

}

}

if (s.equals("")) {

// мы решили, что пустая строка может нарушить в дальнейшем работу нашей программы, например, на результате этого

//метода нам надо вызывать метод substring(1,2), поэтому мы вынуждены прервать выполнение программы

//с генерацией своего типа исключения MyException с помощью throw

throw new MyException("String can not be empty!");

}

return s;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public class PerimetrExeption extends Exception { //конструкторы для Exception

public PerimetrExeption() {

}

public PerimetrExeption(String message) {

super(message);

}

public PerimetrExeption(String message, Throwable cause) {

super(message, cause);

}

public PerimetrExeption(Throwable cause) {

super(cause);

}

public PerimetrExeption(String message, Throwable cause, boolean enableSuppression, boolean writableStackTrace) {

super(message, cause, enableSuppression, writableStackTrace);

}

}

public class PerimetrSquare {

public void getPerimetr(){

Square square = new Square();

try {

square.setSide(0);

} catch (PerimetrExeption e) { //обработка исключений

// e.printStackTrace(); //we can retutn printStackTrace, but because we create our

//message below, message will still be displayed

System.err.println(e.getMessage()); //and we can make own message using getMessage()

}

}

}

public class Square {

private double side;

public double getSide() {

return side;

}

public void setSide(double side) throws PerimetrExeption {

if (side <= 0){

throw new PerimetrExeption("Side is incorrect"); //this is our message, it looks like this, because

//constuctors in PerimetrExeption extends Exception

}

this.side = side;

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public class PerimetrExeption extends Exception {

public PerimetrExeption() {

}

public PerimetrExeption(String message) {

super(message);

}

public PerimetrExeption(String message, Throwable cause) {

super(message, cause);

}

public PerimetrExeption(Throwable cause) {

super(cause);

}

public PerimetrExeption(String message, Throwable cause, boolean enableSuppression, boolean writableStackTrace) {

super(message, cause, enableSuppression, writableStackTrace);

}

}

public class PerimetrSquare {

public void getPerimetr(String str) throws PerimetrExeption {

Square square = new Square();

try {

double side = Double.parseDouble(str);

square.setSide(0);

} catch (NumberFormatException e) {

throw new PerimetrExeption("String is incorrect", e);

} catch (PerimetrExeption ex) {

System.err.println(ex.getMessage());

}

}

}

public class Square {

private double side;

public double getSide() {

return side;

}

public void setSide(double side) throws PerimetrExeption {

if (side <= 0){

throw new PerimetrExeption("Side is incorrect");

}

this.side = side;

}

}

public class CommonMain {

public static void main(String[] arg){

PerimetrSquare perimetrSquare = new PerimetrSquare();

try {

perimetrSquare.getPerimetr("g");

} catch (PerimetrExeption e) {

e.printStackTrace();

}

}

}

//this is the returned message of exception. in the first line our message is displayed

//and from four line to the end causes are described

com.company.lesson\_30.PerimetrExeption: String is incorrect

at com.company.lesson\_30.PerimetrSquare.getPerimetr(PerimetrSquare.java:11)

at main\_progs.CommonMain.main(CommonMain.java:14)

Caused by: java.lang.NumberFormatException: For input string: "g"

at java.base/jdk.internal.math.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:2054)

at java.base/jdk.internal.math.FloatingDecimal.parseDouble(FloatingDecimal.java:110)

at java.base/java.lang.Double.parseDouble(Double.java:549)

at com.company.lesson\_30.PerimetrSquare.getPerimetr(PerimetrSquare.java:8)

... 1 more

Потоки ввода вывода (I/O)

input stream

public static void main(String[] arg){

int b = 0;

FileInputStream fis = null;

InputStreamReader isr = null;

try {

fis = new FileInputStream("c:/1.txt"); //FileInputStream- byte type stream

isr = new InputStreamReader(fis, "UTF-8"); //потомки от абстактных классрв

//Reader и Rider, we have to use them,

//because we need int type stream

while ((b = isr.read()) != -1) {

System.out.println((char)b);

}

} catch (FileNotFoundException e) { //error handling if there is no file

e.printStackTrace();

} catch (IOException e) { //error handling if file is broken, but IOException

//is more important exception, but i left it for accuracy

e.printStackTrace();

}

}

public class CommonMain {

public static void main(String[] arg){

int b = 0;

FileInputStream fis = null;

InputStreamReader isr = null;

try {

fis = new FileInputStream("c:/2.txt");

isr = new InputStreamReader(fis, "UTF-8");

while ((b = isr.read()) != -1) {

System.err.print((char)b);

}

} catch (IOException e) {

e.printStackTrace();

} finally { //we use finally to close input stream, also we ought to

//surround try-catch block

try {

fis.close(); //we closed used byte type variable fis here

} catch (IOException e) {

e.printStackTrace();

}

try {

isr.close() //we closed used int type variable isr here, don't forget try-catch

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

output stream

public static void main(String[] arg){

FileOutputStream fileOutputStream = null;

String str = "dhfdy";

try {

fileOutputStream = new FileOutputStream("d:/1.txt"); //i use disk the d because

// i haven't got access rights for the c disk

fileOutputStream.write(str.getBytes()); //method write is output stream,

//but this method replaces written text in this file

// we use str.getBytes() because write takes byte[] type,

//also we're going to exploit a try-catch

} catch (IOException e) {

e.printStackTrace();

}

}

public class CommonMain {

public static void main(String[] arg){

FileWriter fileWriter = null;

String str = " scsadc";

try {

fileWriter = new FileWriter("d:/1.txt", true);//FileWriter adds text after

//written text early

fileWriter.write(str);

} catch (IOException e) {

e.printStackTrace();

} finally { //if we don't use finally, new text will not add to our file, i don't know cases

//therefore we have to close a variable, therefore we write finally and handle

//with try-catch block fileWriter.close() action

if (fileWriter !=null){ //before closing it's suppose fileWriter is filled in, to do

//this we check, that fileWriter isn't empty

//we must do it every time.

try {

fileWriter.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

}

public static void main(String[] arg){

FileWriter fileWriter = null;

String str = " scsadc";

File file = new File("d:/3.txt");// class File works with external drive,

//it can delete, change, rename, etc.

try {

fileWriter = new FileWriter(file, true);//FileWriter adds text after

//written text early

fileWriter.write(str);

} catch (IOException e) {

e.printStackTrace();

} finally { //if we don't use finally, new text will not add to our file, i don't know cases

//therefore we have to close a variable, therefore we write finally and handle

//with try-catch block fileWriter.close() action

if (fileWriter != null){

try {

fileWriter.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

Сериализация | Десериализация Объектов

Сериализация — это процесс сохранения состояния объекта в последовательность байт.

Десериализация — это процесс восстановления объекта из этих байт.

Интерфейсы-маркеры- это специальные информативные интерфейсы, которые просто помечают наши классы

дополнительной информацией, в будущем полезной для Java-машины. Никаких методов, которые нужно

было бы имплементировать, у них нет. Так вот, Serializable — один из таких интерфейсов.

Java Collections Framework ( коллекции Java )