***Базовые типы данных*** (размер в битах):

byte (8), short (16), int (32), long (64), char (16), float (32), double (64), boolean

int i; – объявление переменной, i=90 - присвоение

Приведение типов:

(1) расширяющее (автоматическое): расширение типа

(2) сужающее (явное): сужение типа, нужно явное приведение, (int)5L.

Литералы:

1, 507 целые в 10м; 02, 056 в 8м; 0х12А в 16м

15L – long, 10.7e+7f – float, 10.25e-9 – double (можно в 8м и 16м)

0b101110 – целое в двоичном, 0х12.2P2 – число с плавающей точкой в 16м

цифры числа можно разделять «\_» для удобства: 100\_000

‘a’, ’\n’, ‘\u52’,’\\’ – char, в последнем случае указан код символа.

Массив (статический размер):

type[] name – ссылка на массив,

int[] arr = new int[5] – создание массива, он заполняется нулями

int[] arr = {1,3,4}; - создание массива с явной инициализацией

arr = new int[]{1,3,4}; - если массив и ссылка создаются в разных строках

int[][] matrix = new int[7][8] – двумерный массив, прямоугольная матрица

int[][] matrix = {{1,3},{4},{4,7,9}} – двумерный непрямоугольный массив

matrix[1] = new int[2] – создание второго измерения массива

for(int v : arr) – v принимает значения каждого элемента массива. Чтобы поменять элемент массива, нужен перебор по индексу.

arr.clone() - копирование массива целиком

arr.length – длина массива

Массив + статические методы Arrays:

import java.util.Arrays; Arrays.metod(...)

toString(arr) – массив в виде строки типа [1, 4, 6, 0]

sort(arr) – сортировка по возрастанию

copyOfRange(arr, i,j) – копирование элементов массива с arr[i] по arr[j-1]

copyOf(arr, j) - копирование c arr[0] по arr[j-1]

equals(arr1,arr2) – сравнение массивов (arr1.equals(arr2) – не работает)

fill(arr,i,j,6) – заполнение массива с i по j-1 например цифрой “6”

String

“символьная строка” String str; - ссылка на тип, str=”присваивание”

! ни один из методов не меняет строку, а возвращает ссылку на новую

s1.equals(s2) – сравнение строк s1.length – длина строки

s1.equalsIgnoreCase(s2) – сравнение строк без учета регистра символов

**s1.compareTo(s2) – сравнение строк., возвращает -a,0,+a (<,==,>)**

**s1.compareToIgnoreCase(s2) – тоже но без учета регистра.**

s1.charAt(i) – символ строки

Long.parseLong(s1), Integer.parseInt(s1) и.т.п – строку в число

s1.isEmpty() – true если строка пустая

toUpperCase(), toLowerCase() – строку в верхний/нижний регистр

substring(i,j) – подстрока c i до j-1; substring(i) – подстрока с i до конца

s1.starsWith(s2) – true если строка s1 начинается с подстроки s2

s1.endsWith(s2) – true если строка s1 заканчивается подстрокой s2

s1.contains(s2) – true если s1 содержит s2

s1.replace(s2,s3) – замена в строке s1 подстроки s2 на подстроку s3

s1.replaceAll(“регулярное выражение в кавычках”,s2); - замена символов из регулярного выражения на подстроку s2, “” – символы просто удаляться

[a-dxy] – перечисленные символы, [^a-cj-k] – кроме перечисленных

s1.trim() – удаление пробелов в начале и в конце строки

s1.toCharArray() – строку в массив **char[]**

String[] strarr – ссылка на массив строк

strarr = s1.split(“regex”,n) – строка разбивается на подстроки, разделенные символами из регулярного выражения, результат превращается в массив. n-максимальное число строк (можно опустить). regex можно опустить, слова будут считаться разделенными пробельными символами.

Операции над строками:

toString() – содержимое объекта в строку

String.format("name=%s, age=%d", *name*, *age*);} – форматная строка String

итог: ***name=Paul, age=36***

s1=s1+s2 – слияние двух строк, получение новой строки вместо s1

s1=5 + “=х” – число приводиться к строке, затем слияние

s1=5+’g’+”х=” – ‘g’ в число, результат складывается с 5,результат в строку

s1.indexOf(‘a’) – индекс первого вхождения символа/подстроки в строку

# Изменяемые строки StringBuilder и StringBuffer

# StringBuilder – работает быстрее

# StringBuffer – безопасная работа с потоками

# new StringBuilder(“строка”) – создание объекта изменяемой строки

# s1.reverse() – разворот строки

# s1.delete(i,j) – удаление из строки элементов с i по j-1(либо до конца строки)

# s1.insert(i,value) – вcтавка символа/массива символов, числа или строки в позицию i.

# replace(i, о, str)- заменяет подстроку [i,j) на строку str любой длины

# s1.append(s2) – слияние строк

# Разное:

! В файле один public класс, его имя совпадает с именем файла

public static void main(String[] args){} – точка входа в приложение

package test; - созданные классы будут в этом пакете

import java.util.Array; - подключение класса из пакета или всего пакета

java.lang – подключается неявно автоматически

a **instanceof** A – возвращает true если ссылка ‘a’ указывает на О типа ‘А’

a.getClass() – возвращает тип О

Операторы ветвления:

**if** (логическое условие) {операторы}

**else if**(второе логическое условие){операторы}

**else** {операторы}

! else и else if потребуются при наличии двух и более путей

при наличии одного оператора {} можно опустить

**switch**(byte, short, int, char, enum либо String){

**case** 1: операторы;**break**;

**case** 2: операторы;**break**;

**default**: операторы;//необязательный оператор

} – если break опустить то выполняться операторы и последующих случаев, пока не встретиться команда break или до конца switch

(логическое условие) **?** (выражение или число в случае true) **:** ( если false)

int i= (a>0) ? a : -a; или a=b + ( (c==b) ? c : c\*c);

Циклы:

do {операторы} while (условие); while(условие){операторы}

for(int i; условие; инеремент/декремент)

for(int value: array) – перебор элементов массива, для перебором массива пользовательских типов требуется наличие итератора.

break, continue – выход из текущего цикла / переход к следующей итерации

out: - метка перед циклом, позволяет выйти сразу из нескольких циклов

break out – выход из всех циклов, стоящих после метки

continue out – переход к следующей итерации цикла после метки

Методы:

! методы можно определять только внутри класса

Type name(args){операторы;return} - синтаксис

! перегруженные методы имеют одно имя

! переопределенные методы имеют одно имя и сигнатуру

! аргументов по умолчанию в java нет

int f(){return 4;} – возвращает int, void f(){} – ничего не возвращает,

void f(int i){} или int f(int i, float g) – принимает один/два аргумента

! при передаче объекта или массива, передается только ссылка на него

void f(MyClass mc);void f(int[] arr) – передача ссылки на объект/массив

int[] f(){}; MyClass f(){} – возврат ссылки на массив/объект

void f(int...arr) – метод с переменным числом аргументом, они превращаются в массив int[]. f(1,2,3) или f(intarr) – вызов метода

Перечисления enum:

enum Planets{MERCURY, VENUS} – синтаксис

! каждой константе соответвует номер начиная с 0

Planets pl=Planets.MERCURY; - ссылка и инициализация значением

pl.ordinal() – номер константы, pl.values() – массив констант

! pl может принимать только указанные значения

enum Planets{ // группа связанных констант

MERCURY(0.055, 0.38), SATURN(0.815, 0.95);

private double mass, radius;//обязательно

double getMass(){}; double getRadius(){}

private Planets(double d, radius r){mass=m;radius=r;} //обязательно

}// при вызове pl.getMass() программа вернет 0.055

enum Calc{ // здесь уникальные для каждой константы методы

ADD{public int exec(int a, int b){return a+b}},

DEF{public int exec(int a, int b){return a-b}};

public abstract int exec(int a, int b);//обязательный абстрактый метод

}// вызов такого метода pl.exec(5,7), похоже на указатель на функцию

Классы:

class A{} – синтаксис;A(){} – конструктор; A(int i){this.i=i;} – конструктор с аргументами

!static методам доступны только static поля, методы и конструкторы вложенных static классов, доступ к ним только через имя класса A.f()

!статические методы нельзя переопределять и наследовать.

final int i; - просто константа, своя для каждого объекта

static int i; - переменная, общая для всех О-в класса

final static; - константа уровня класса, общая для всех О-в класса

A a=new A(7); - создание ссылки и создание объекта вызовом конструктора

! при создании О поля инициализируются нулевыми значениями

private / protected / [не указан] / public - доступ к полям и методам класса соответственно для методов того же класса / +дочерний класс/ в пределах пакета / доступ для всех.

! если конструктор сделать public, тогда О можно создать static методом.

A(int i){this(); } – вызов конструктора из конструктора (this всегда первым)

! Поля метода рекомендуется всегда делать private, для доступа к ним используются геттеры и сетторы,это обычные методы, обычный синтаксис

Type getИмяПоля() – возвращает значение поля

boolean isИмяПоля() – тоже, при условие что поле boolean

setИмяПоля(Type value) – устанавливает новое значение переменной

protected void finalize() {} – этот метод выполняется при удаление объекта, что может никогда не произойти за все время работы программы.

Вложенные классы:

class A{ A a=new A(); A.B b=a.new B() – O внутреннего

class B{ не static класса. b.i – доступ к полю внутр. класса

void f(){} a.i – доступ к полю внешнего класса.

int i; ! Если вложенный класс с модификатором private

} его О можно создать только внутри внешнего

int i; ! Вложенному классу доступны даже private поля внешнего

} ! Если вложенный класс static, то ему доступны только static поля

внешнего. A.B b=new A.B() – O статического вложенного класса, объект внешнего класса создавать не нужно.

Абстрактый класс и интерфейс:

abstract class A{ // О абстрактного класса создать нельзя

abstract void f1();// абстрактный метод, не требует реализации

void f2(){}//обычный метод, имеет тело

int i;

A(){}//конструктор у абстрактного класса есть, даже по умолчанию

}

interface IA{//нельзя создать О интерфейса, и конструктора нет

abstract void f1();//могут быть только абстрактные методы, но

default void f2(){}// при расширении интерфейса необходим

} // модификатор default и тело для метода для обратной совместимости

слово abstract здесь тоже не обязательно.

! Интерфейс может иметь статические методы, вызов: IA.f()

@FunctionalInterface–проверка,что интерфейс функциональный,т.е.имеет только один абстр. метод,число методов default/static не ограничено.

java.util.function – пакет функциональных интерфейсов, для лямбд.

Параметризированные типы. Дженерики.

class A<T>{ //тип передается в качествве параметра

T f(T t){return t;} // использование полученного типа

A(){}//обычный конструктор, может использовать полученный тип

T t;//поле, переменная

}

A<String> astr; - ссылка; new A<String>() – создание О вызовом конструктора

Создание О с помощь статического метода:

private A(T t){this.t=t;} //приватный конструктор, не принимает тип

static <**X**> A<**X**> makeA(**X** xx){ return new A<**X**>(xx); }; // статический метод класса A<T> принимает тип **X** и переменную этого типа x, вызывает приватный конструктор, передавая ему тип и переменную, **int[]->X->T**, полученная от конструктора ссылка типа A<X> возвращается из метода.

A<**int[]**> astr = A.<**int[]**>makeA(arr); - вызов такого метода в коде.

! Функциональный интерфейс может быть дженериком: Fun<X,Y>{R f(T t);}

Наследование и полиморфизм:

final class A(){} – от такого класса наследовать нельзя

final void f(){} – такой метод нельзя переопределить

class B extends A{} – наследование класса, можно только один

class B implements I1,I2{} – реализация одного или нескольких интерфейсов

class B extends A implements I1,I2{} – и то и другое

! при наследовании поля и методы суперкласса становять частью класса

! private методы и поля не наследуются

@Override void f(){} – реализация/переопределение метода

! переопределение создает полиморфные методы

A a=new B(); или I1 i1=new B() - можно создать ссылку типа суперкласса, абстрактного класса или интерфейса, при этом будут доступны только методы суперкласса/абстрактного/интерфейса. Этой переменной можно присвоить ссылку на дочерний класс, при использовании ссылки будут запускаться переопределенные/реализованные методы, а не базовые.

i1.f() – запуститься версия f() из B

((B)i1).g() – вызов уникального метода g() из B, в суперклассе его нет.

super.f() – вызов метода супер/абстрактного класса из метода дочернего.

! при переопределении метода уровень доступа не должен уменьшаться, у абстрактных методов в интерфейсе уровень доступа всегда public.

! при одновременной реализации двух и более интерфейсов возможны совпадения имен и сигнатур методов: нужна одна реализация метода, даже если у метода в одном/всех интерфейсах у метода есть реализация.

Анонимные классы и лямды:

class A{ void f(){...}} – класс; interface I{void g();} – интерфейс

A a=new A(){...@Override void f(){новое тело}} – О анонимного класса, наследуется от класса/абстрактного класса А.

I i=new I(){{...@Override void g(){реализация}} – реализация интерфейса.

! анонимный класс используется когда нам нужен один объект, при этом класс небольшой, и в нем переопределяется одна/две функции, реализация которых уместна только для одного этого объекта.

! анонимный класс имеет доступ к полям класса, в котором он определен.

interface IO{int calc(int a,int b);} – функциональный интерфейс, один метод.

IO sum=(x,y)->x+y; IO def=(x,y)->x-y; - реализация интерфейсов где-то в коде:после стрелки тело метода, по сути это объекты ананонимного класса.

IO oper; - может указывать на О анонимного класса, реализующего IO.

oper=sum; oper.calc(x,y) – вызов лямды

! лямды можно передвать в метод void g(IO io){}, и возвращать из метода

IO h(){return oper}.

Корневой класс Object:

! От этого класса явно или неявно наследуются все методы, этот тип будет использоваться в дженериках, если тип <T> не указывать.

final: getClass(),notify(),notifyAll(),wait()

необходимо переопределение методов:

@Override public **boolean** equals(Object obj){

if(this == obj) return true;

if(obj == null || obj.getClass() != this.getClass() ) return false;

MyClass pr=(MyClass)obj;

//далее следует сравнение полей, если поля – ссылки - повторение

} //для каждой трех пунктов выше

@Override public **int** hashCode(){

int k=31; int resault=1;//тип result соответвует широчайшему типу поля

resault=k\*resault+field1;// если поле это ссылка на объект, проверяем

resault=k\*resault+field2;//если null подставляем 0, в противном случае

return (**int**)resault;// подставляем field.hashCode()

} // возвращаем результат, приведенный к типу int

@Override public String toString(){// показ содержимого любых полей

return String.format("name=%s, age=%d", name, age);} // как угодно

//в противном случае вернется имя объекта и значение ссылки на него.

***! Сравнение объектов на больше-меньше***, нужно для коллекций.

(1) «natural ordering»: Comparable<T> и compareTo(T t) «естественный»

Класс должен реализовывать интерфейс Comparable<MyClass>

class MyClass ***implements Comparable<MyClass>***{

private int id;

private int age;

public int getId(){return id;}

public int getAge(){return age;}

MyClass(int id, int age){this.id=id;this.age=age; }//Collections.sort(list);

@Override//следующая конструкция работает и для дочернего класса

***public int compareTo(MyClass right){return this.id-right.id;}***

}! натуральную сортировку можно менять только внутри объекта.

(2) Компаратор: создаем класс, реализующий интерфейс компаратор, сортировка возможна только по полям, получаемым по getField.

***Comparator***<MyClass> cmpr=new ***Comparator***<>(){//исп. анонимный класс

public int ***compare***(MyClass m,MyClass r){ return m.getId()-r.getId();} };

Collections.sort(list, cmpr);// исп. Компаратора на основе анон. класса

Collections.sort(list, ***(x,y)->Integer.compare(x.getId(),y.getId())*** );//исп.дельту

! определенный компаратор аналогично работает и с дочерним классом, для сортировки компаратором класс менять не нужно.

# Дата и время Date

# import java.util.Date; Date dt=new Date(); Date dt=new Date(); Date(мс)

# getTime() – текущее время в мс с 01.01.1970

# setTime(10мс) – установить время

# ! Есть методы чтения/записи даты, времени, часов минут ит.п., но эти методы считаются устаревшими. Лучше использовать класс Calendar.

Генератор случайных чисел Random:

import java.util.Random;

Random rnd=new Random(any number)

rnd.nexInt(n) – число [0,n), rndDouble() – число [0.0, 1.0) ит.п.

Scanner (чтение с консоли):

import java.util.Scanner;

Scanner sc = new Scanner(System.in);

sc.nextInt(), nextFloat() – чтение чисел с консоли

next() – строка до пробела, табуляции и т.п. т.е слово

nextLine() – строка до символа конца строки

hasNextLine(), hasNextShort() и.т.п– true если в буфере строка/short и.т.п.

hasNext() – true если в буфере что-то есть

! в java очистки буфера нет, надо просто прочитать до конца строки.