

Съдържание

1. Увод.....	3
	4
	6
3.1. Основни типове данни и оператори.....	6
Тема 1: Алгоритми и програми.....	7
Тема 2: Синтаксис и семантика на език ЕП.....	11
Тема 3: Типове данни.....	17
Тема 4: Линейни алгоритми.....	27
Тема 5: Вземане на решения.....	35
Тема 6: Многовариантен избор.....	43
Тема 7: Повтарящи се действия.....	48
Тема 8: Повтарящи се действия (продължение).....	56
Тема 9: Обобщение: основни типове данни и оператори...	62
3.2. Програмиране на основни алгоритми.....	71
Тема 10: Наследяване.....	72
Тема 11: Интерфейс.....	79
Тема 12: Подалгоритми и подпрограми.....	84
Тема 13: Тип масив.....	93
Тема 14: Изключителна ситуация	100
Тема 15: Работа с текстове.....	109
Тема 16: Рекурсивни алгоритми.....	120
Тема 17: Обработка и съхранение на данни.....	126
5. Приложение 1. Допълнителни задачи.....	135
	162
	167
8. Литература.....	182

РЪКОВОДСТВО ПО ПРОГРАМИРАНЕ НА БАЗАТА НА ЕЗИКА JAVA

ЧАСТ 1 **ОСНОВНИ ТИПОВЕ ДАННИ И ОПЕРАТОРИ**

Тема 2: Синтаксис и семантика на ЕП

Тема 3: Типове данни

Тема 4: Линейни алгоритми

Тема 5: Вземане на решения

Тема 6: Многовариантен избор

Тема 7: Повтарящи се действия

Тема 8: Повтарящи се действия (продължение)

Тема 9: Обобщение: основни типове данни и оператори

ТЕМА 2

СИНТАКСИС И СЕМАНТИКА НА ЕЗИК ЗА ПРОГРАМИРАНЕ JAVA

Всяка програма на език за програмиране се изгражда, като се използва определен набор от знакове, наричан **азбука** на езика. Азбуките на повечето езици за програмиране включват латинските буква, десетични цифри и някои други знакове като '+', '/', ':' и др.

При писането на програми трябва да се спазват определени строги правила. Има два типа правила, определящи съответно начина на записване (синтаксиса) и смисъла (семантиката) на езиковите конструкции.

Синтактичните правила определят кои последователности от знакове на азбуката на съответния език за програмиране са допустими езикови конструкции. Съгласно тези правила едни последователности от знакове са правилни, а други – не.

Семантичните правила определят смисъла на синтактично правилните конструкции, т.е. как те трябва да се разбират от човек и как ще бъдат интерпретирани при изпълнение от компютър.

Елементи на език за програмиране

Служебните думи представляват съвкупност от думи, които са запазени от езика за програмиране и не могат да се използват за имена на променливи и т.н. Примери за служебни думи в Java са **import, if, public, case** и др.

Чрез знаковете от азбуката на езика за програмиране могат да се записват числа (напр. 3, 12, -7.77), да се дават имена на програмите и на величините (напр. X, Y, price) и да се изграждат по-сложни езикови конструкции като изрази, оператори и т.н.

Имената в езиците за програмиране са последователности от букви и цифри, започващи с буква. Такива последователност се наричат **идентификатори**. Идентификатори в езика Java са

например age, price, age3, a2 и т.н. Не са идентификатори 2age, 1a2.

В програмите се използват и изрази (например $3+X$, $\text{age}*\text{coef_k}$, $((A+B/C)/(B-A/C))$ и др), в които участват константи, променливи и знаци за операции между тях.

Програмата на език за програмиране представлява конкретен компютърен алгоритъм, а алгоритмите се характеризират с два типа параметри – данни и правила за тяхната обработка. Това определя структурата на програмата. Най-общо програмата се състои от две части – част за описание на данните и част за описание на тяхната обработка.

В първата част се дават сведения за характера на данните (дали са целочислени, реални, логически и т.н.), за техния вид (константи или променливи), за имената им и др. Тези сведения улесняват разбирането на програмата и откриването на допуснати грешки.

Във втората основна част на програмата (класа) се описва обработката на данните с помощта на т. нар. Оператори на езика за програмиране.

Операторът задава определено елементарно действие, например:

```
Y = Y + 7; // да се увеличи Y със 7,  
if (age >= 18)  
    {Status = adult}; // ако годините са повече или 18, то на  
//променлива с име Status присвояваме пълнолетен (adult).
```

Ако алгоритъмът, който програмата описва, е по-сложен и е представен с използване на подалгоритми, то подалгоритмите се оформят като подпрограми, чиято структура наподобява тази на програмите.

С цел програмите (класовете) да са разбираеми за хората, в техния текст могат да се вмъкват, по определени правила, т.нар. **коментари**, които съдържат обяснения за човека и които не се взимат предвид от транслятора.

Например *// пояснение за програмисти* е коментар на езика Java. Когато сме поставили знака *//*, то всичко, което сме написали след него, но само до края на реда, се игнорира от транслятора.

Методи за описание на синтаксиса и семантиката

Семантиката на конструкциите в езиците за програмиране се описва точно доста трудно и обикновено се определя с изречения на естествен език.

За описване на синтактичните правила са намерени прости способности и средства.

За описание на синтаксиса на езика Java в следващите уроци ще се използва нагледен метод, като въвеждаме следното условие: всеки израз, заграден в прави скоби [] е **незадължителен**, а определението на всеки израз, заграден в <> е дадено другаде.

Пр. 1 Нагледен метода за определяне на:

а) цифра;

0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

б) цяло без знак.

цифра[цифра[цифра[...]]]

Където пример 1а определя понятието цифра. Показали сме, че цифра, това е всеки един елемент от изброеното множество. Т.е., че цифра е всеки един от знаците 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Пример 1б определя понятието “цяло число без знак”, като ползва определението за “цифра”. Т.е. посочва, че цяло без знак може да бъде и само една цифра или много цифри наредени една до друга. С други думи цялото число без знак е последователност от една или повече цифри.

Пр. 2 Нагледен метода за определяне на:

а) знак :

+, -

б) цяло число:

[<знак>] <цяло без знак>

За определя <знак> като един от двата знака: или + или - . Всяко <цяло число> от 2б се образува от <цяло без знак>, пред което има или няма <знак>. В този случай квадратните скоби са използвани за да се изрази възможността за пропускане на част от конструкцията.

Структура на клас (програма) в езика Java

Основните елементи на езика Java, както и структурата на класа на Java съответстват изцяло на разгледаните по-горе общи принципи. Както се вижда от следващото описание, основна част на класа е блока (тялото), който се предшества от име (заглавие) на класа. Имената (идентификаторите) на всички програмни елементи се образуват по правилата за конструиране на идентификатори в езика.

```
class идентификатор {  
    блок;  
}
```

Блокът, от своя страна, съдържа имена на променливи и константи, задават се начални стойности, описват се оператори и методи, като винаги трябва да включва в себе си главен метод (мейн /main/ метод).

За да си изясните по-пълно синтактичните правила на Java, свързани с подреждането на основните елементи на класовете, проучете внимателно следващия пример.

Пр. 3. Структура и основни елементи на програма на Java

```
import java.io.*;      //указва, че ще използваме библиотека io  
class ex_22 {          //създаваме нов клас с име ex_22  
    int x=23;  
    int y=45;  
    void change_x_y () { //процедура за смяна на променливите  
        int buf = x;
```

```

        x = y;
        y = buf;                // буферна променлива
    }
    public static void main(String[] args) { // главна прогр.
        ex_22 newex_22=new ex_22();
        System.out.println("Въведена стойност за
x:"+newex_22.x+"стойност за y:"+newex_22.y);
        newex_22.change_x_y ();        //използва подпрограма
        System.out.println("След размяната x е: " + newex_22.x + ", а y
e:"+newex_22.y);
    }
}

```

ТЕМА 3

ТИПОВЕ ДАННИ

Величини

Всички величини, които обработват методите на класа се наричат с общото име величини. Те биват два вида - константи и променливи.

Константите имат точно определена стойност, която не се променя по време на изпълнение на класа, а само се използва в процеса на изчисление.

Променливите могат да се изменят по време на изпълнение на класа. всяка променлива има име, което се избира от програмиста. За предпочитане е името да се избира така, че да е свързано със смисъла на променливата.

Типове данни

За улесняване обработката на данните е въведено понятието тип на данните. Посочвайки типа на една променлива, се посочва неявно както диапазона и вида от стойности, които може да приема променливата, така и кои са характерните и допустими операции, които могат да се извършват с нея. Т.е. задавайки типа, ние неявно посочваме и:

- множеството от стойности, които може да приема;
- множеството от операции, които може да се извършват над променливата;
- множеството от отношенията (релациите) между променливите от типа;
- множеството от стандартни функции за типа (ако има такива).

За някои типове се задава и множество от стандартни функции, които улесняват програмиста и чрез които се извършват по-сложни преобразувания на данните.

Пр.1

Реален тип данни:

а/ стойности: 3.27, 1.25, -31.2 и т.н.

б/ операции: "+" (събиране), "-" (изваждане), "*" (умножение) и т.н.

в/ релации: ">" (по-голямо), ">=" (по-голямо или равно), "<" (по-малко) и т.н.

г/ стандартни функции от вида: $|x|$, x^2 , $\cos x$ и т.н.

При изучаване на всеки нов тип данни е необходимо да се обърне внимание на следното:

1. Начин за деклариране на типа и на величините от този тип.
2. Множеството от стойности на типа.
3. Допустими операции над величините от този тип.
4. Стандартните функции за този тип (ако има такива).

Прости типове данни, наричаме онези типове, чиито стойности са неделими. Най-използваните прости типове данни

(цял, реален, знаков и логически) са градени в езиците за програмиране и могат да се използват непосредствено в класовете (програмите). Тези типове се наричат стандартни. Програмистите могат да създават и собствени прости типове данни на базата на стандартните типове данни.

Константите от простите типове се записват по начин, близък до общоприетия в математиката. Например:

- 27, 14 - целочислена константа;
- -3.15 и 5E-3 - реални константи;
- true и false - логически константи със съответни стойности "вярно" и "невярно";
- 'a', 'A', '_' - знакови константи.

Простите типове данни биват два вида - дискретни и реални. Дискретните типове се характеризират с това, че всички стойности на типа могат да се подредят по големина последователно една след друга. Това означава, че за всяка константа от някой дискретен тип е известно коя е предходната и коя е следващата (с изключение съответно на най-малката и най-голямата).

Пр.2:

1. Целочисленият тип е дискретен.
2. Реалният тип не е дискретен т.к. за дадена реална константа не може да се посочи коя е непосредствено следващата (предходната) по големина.

Реален тип данни

Съществуват два стандартни идентификатора (имена), чрез които се дефинират реален тип данни: float и double.

1. Множество от стойности.

а/ множеството от стойности на реален тип float се състои от числа в диапазона от -3.40282347E38 до 3.40282347E38, записвани със седем значещи цифри.

б/ множеството от стойности на реален тип double се състои от числа в диапазона от -1.79769313486231570E до 1.79769313486231570E, записвани с петнайсет значещи цифри.

Трябва да се внимава да не се излиза извън посоченото множество от стойности. Обратното може да доведе до грешки в програмата.

Реалните константи се записват с десетична точка (3.1415, -0.0007, 9.9) или с порядък (-0.002E-6, 5E11, 0.1E10). При запис чрез порядък изразът "E-6" е равностоеен на 10^{-6} , съответно "E+11" е еквивалентно на 10^{11} .

Пр.3.

1. $0.2E3=200$

2. $0.00025E4=2.5$

2. С величините от реален тип (float и double) могат да се извършват операциите:

"+" - събиране;

"-" - изваждане

"*" - умножение;

"/" - деление.

Пр.4.

Нека x е име на променлива от реален тип float (double). Тогава следните записи са коректни:

1. $x + (1.35 - 67.89)$

2. $((x*2.24 + 5.112) - 3.14) / 5$

3. За реалните величини са валидни известните от математиката релации за сравнение. Те се осъществяват чрез следния запис:

"==" - равно;

"!=" - различно;

">" - по-голямо;

">=" - по-голямо или равно;

"<" - по-малко;
"<=" - по-малко или равно.

4. В Java са вградени някои стандартни функции за работа с реални числа. Най-често използваните от тях са:

- `Math.abs (x)` - връща като резултат абсолютната стойност на израза в скобите т.е. в случая връща $|x|$;
- `Math.sqrt (x)` - връща като резултат корен квадратен от израза в скобите;
- `Math.cos (x)` - връща като резултат косинуса на израза в скобите;
- `Math.sin (x)` - връща като резултат синуса на израза в скобите;
- `Math.tan (x)` - връща като резултат тангенса на израза в скобите;
- `Math.PI` - връща като резултат числото π т.е. 3.14;
- `Math.pow (x, y)` - връща като резултат числото x на степен y , където x и y са променливи от реален тип (`float` или `double`);
- `Math.round (x)` – закръгля реалното число x до цяло число;
- `Math.floor (x)` – закръгля реалното число x до по-малкото цяло число;
- `Math.ceil (x)` – закръгля реалното число x до по-голямото цяло.

Пр.5.

1. `Math.abs (-34.7)` - връща като резултат 34.7;
2. `Math.sqrt (16)` - връща като резултат 4;
3. `Math.pow (4, 2)` - връща като резултат 16;
4. `Math.round (3.4)` - връща като резултат цялото число 3;
5. `Math.round (3.6)` - връща като резултат цялото число 4;
6. `Math.floor (3.6)` - връща като резултат цялото число 3;
7. `Math.ceil (3.6)` - връща като резултат цялото число 4.

Всяка стандартна функция има име и определен брой аргументи от даден тип. Тя извършва съответното действие над своите аргументи и връща като резултат единствена стойност от определен тип.

Дискретни типове данни

Стойностите от всеки дискретен тип могат да се сравняват. Те могат да се номерират последователно с поредни номера - числата 0,1,2,3... По този начин 0 съответства на най-малката стойност, 1 - на следващата и т.н. Целочисленият тип данни е дискретен тип.

Целочислен тип данни

Съществуват четири стандартни идентификатора, чрез които се дефинира целочислен тип данни: byte, short, int и long.

1. Множество от стойности.

а/ множеството от стойности на целочислен тип byte е в диапазона от -128 до 127;

б/ множеството от стойности на целочислен тип short е в диапазона от -32 768 до 32 767;

в/ множеството от стойности на целочислен тип int е в диапазона от -2 147 483 648 до 2 147 483 647;

г/ множеството от стойности на целочислен тип long е в диапазона от -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807.

Нека отново отбележим, че трябва да се внимава да не се допускат грешки от препълване, възникващи когато някой от методите на класа се опитва да присвои по-голяма (по-малка) стойност, отколкото може да съхрани променливата от зададения тип.

2. С величини от целочислен тип могат да се извършват следните операции:

- “+” – събиране;
- “-” – изваждане;
- “*” – умножение;
- “/” – деление;
- “%” – деление по модул;
- “++” – увеличение с 1;
- “--” – намаление с 1.

Когато последните две са в ляво от променливата, то те се прилагат преди използването на променливата в израз. Ако са в дясно, увеличението или намалението се прави след използване на променливата в израз.

Пр. 6

1. $9\%2$ – т.к. $9/2$ връща резултат 4.5, имаме цяла част 4 и модул (остатък) 5, следователно $9\%2$ ще върне като резултат 5;
2. $8\%2$ – аналогично, следва че ще получим като резултат 0 т.к. 8 се дели на 2 без остатък;
3. $9\%4$ – ще върне като резултат 25 т.к. $9/4$ е 2.25.

3. За целочислените типове данни са в сила релациите за сравнение: “=” (равно), “!=” (различно), “>” (по-голямо), “<” (по-малко), “>=” (по-голямо или равно), “<=” (по-малко или равно).

4. Най често използваните стандартни функции за работа с цели числа са:

а/ `Math.abs (x)` – връща като резултат абсолютната стойност на x;

б/ `Math.pow (x, y)` – връща като резултат x на степен y.

Знаков тип данни

Стойностите на величините от знаков тип са елементи на крайно множество от знакове, между които се предполага, че е

въведена наредба. В паметта на компютъра знаковете се представят с кодове, представляващи цели неотрицателни числа, а наредбата между два знака се определя от техните кодове – един знак е “преди” друг, ако неговият код е по-малък. Едно от най-често използваните в компютрите множество от знакове и кодове, съответства на американския стандартен код за обмен на информация (ASCII).

Знаковият тип в езика Java се декларира чрез стандартния идентификатор **char**.

1. Множеството от стойности на знаковият тип данни се състои от 256 знака с поредни номера от 0 до 255. За означаване на определен знак (константа от знаков тип) той се загражда с апострофи, например 'a', '_', ' ' (интервал).
2. Стандартни операции за работа със знакови величини няма, но са позволени релациите за сравнение: <, >, >=, <= и т.н.

Деклариране на данните

Всяка величина в програма (клас), написана на Java, има строго определен тип.

Декларирането на константи се извършва с помощта на служебната дума **final**. Тя се изписва пред типа на константата и показва, че зададената величина е постоянна и няма да се променя по време на изпълнението на класа.

Пр. 7

```
final int const_1 = 375; //дефиниране на константа с име  
//const_1 от mun int  
final float const_2 = 67.354; // дефиниране на  
константа // с име const_2 от mun float
```

При декларирането на променливи първо посочваме какъв е типа на променливата и след това нейното име. Името на променливата трябва да е свързано с контекста ѝ.

Задаването на стойност на променливата може да се извърши още при нейното деклариране или по-късно в програмата.

Пр. 8

```
float price; // деклариране на реална променлива с име
// price
int age; // деклариране на целочислена променлива с
// име age
int My_age = 23; //деклариране на целочислена
//променлива с име My_age и задаване на начална
стойност //на променливата 23.
```

ТЕМА 4

ЛИНЕЙНИ АЛГОРИТМИ

Линейни алгоритми, наричаме онези алгоритми, за които посочените в тях действия се изпълняват по реда на записването им.

Пр. 1.

Нека да съставим алгоритъм за изчисляване на чистата заплата (ЧЗ) на работниците по дадена стойност на основната им заплата (ОЗ). ЧЗ се получава, като към ОЗ се прибави сума за прослужени години, равна на 21% от ОЗ (считаме, че всички работници са с еднакъв трудов стаж) и полученият сбор се намали с 20% (удръжки за данък общ доход). Алгоритъмът, по който може да извършите тези изчисления, е следният:

Вход: Стойност на ОЗ.

Изход: Стойност на ЧЗ.

Стъпка 1: Въведете и запомнете ОЗ.

Стъпка 2: Пресметнете и извършете присвояването
 $ЧЗ = ОЗ + ОЗ * 21 / 100$.

Стъпка 3: Пресметнете и извършете присвояването
 $ЧЗ = ЧЗ - ЧЗ * 20 / 100$.

Стъпка 4: Съобщете стойността на ЧЗ като резултат.

Стъпка 5: Прекратете работа.

Предложеният алгоритъм е линеен алгоритъм, защото посочените в него действия се изпълняват по реда на записването им.

Записвайки този алгоритъм с помощта на език за програмиране, ще получим съответно **линейна програма**.

За да изпълним алгоритъма е необходимо да въведем и изведем стойностите на ОЗ и ЧЗ и да извършим някои изчисления и присвоявания и да изведем крайния резултат. Подобно е предназначението и на операторите, които езиците за програмиране предоставят за описание на линейни програми.

При съставяне на линейни програми на език за програмиране обикновено се използват следните основни оператори: **оператор за въвеждане на начални данни, оператор за присвояване и оператор за извеждане на резултат**.

Чрез оператор за извеждане може да извеждаме на екран кратки съобщения или текущата стойност на променливата за момента, в който извикваме този оператор.

Оператора за въвеждане служи за задаване на различни стойности на променливата от клавиатурата по време на изпълнението на програмата.

Чрез оператора за присвояване може да присвояваме различни стойности на променливите в тялото на програмата и по този начин променяме текущата стойност на променливата.

Оператор за присвояване

Оператора за присвояване служи за присвояване на междинни или крайни стойности на променливите. Той се използва също така и при деклариране на променливите за определяне (задаване) на техните начални стойности.

Синтаксис на оператор за присвояване:

$$\langle \text{идентификатор} \rangle = \langle \text{израз} \rangle;$$

, където идентификатора е име на някаква променлива, а израза е израз, който искаме да присвоим на тази променлива.

Стойността на израза трябва да е от множеството на допустимите стойности, характерно за типа на променливата.

При изпълнение на оператора за присвояване първо се намира стойността на израза и след това тя се присвоява на променливата.

Пр. 2.

Нека I и J са декларирани предварително като целочислени променливи, а X и Y като реални:

1. $I = (15 \% 2) * 2;$ *// вярно*

```
2. X = Math.sqrt(I);           // верно
```

3. J = Math.abs(I) + X; *//ошибка !*

4. $Y = -\text{Math.abs}(X) + I$; // *вярно*

5. `int I = 100;` *// вярно присвояване извършено*

//още при деклариране на променливата

6. float X = 3.758; // *вярно присвояване извършено*

//още при деклариране на променливата.

В Java са вградени и така наречените “съкратени оператори за присвояване”.

Нека `x` и `y` са декларирани променливи от подходящ тип. Съкратените оператори за присвояване са показани в следната таблица:

Оператор	Пример	Действие
<code>+=</code>	<code>x +=y;</code>	<code>x = x + y;</code>
<code>-=</code>	<code>x -=y;</code>	<code>x = x - y;</code>
<code>/=</code>	<code>x /=y;</code>	<code>x = x / y;</code>
<code>*=</code>	<code>x *=y;</code>	<code>x = x * y;</code>
<code>%=</code>	<code>x %=y;</code>	<code>x = x % y;</code>

Оператор за въвеждане на данни

Операторът за въвеждане на данни позволява някои от променливите на изпълняваната програма да получат стойности от външно устройство. В езика за програмиране Java въвеждането на данни става с помощта на стандартният метод **read**. Този метод е част от библиотека **io** на Java и за да можем да ги използваме в тялото на класа, който създаваме, трябва преди да започнем да описваме класа, да посочим, че ще използваме тази библиотека. Това става по следния начин:

```
import java.io.*; // с този ред посочваме, че ще ползваме
//методите на библиотека io.
```

Методът за въвеждане на данни изглежда по следния начин:

```
System.in.read();
```

и чете байт от стандартния входен поток т.е. клавиатурата

Оператор за извеждане на данни

Както споменахме чрез оператора за извеждане на данни на компютърния екран могат да се отпечата стойностите на избрани променливи, като и различни съобщения.

Методите за извеждане на данни изглеждат така:

```
1. System.out.print(идентификатор [+ идентификатор+...]);
//когато искаме да отпечата стойността на даден
//параметър
```

```
2. System.out.print("Кратко съобщение");  
// когато искаме да отпечата кратко съобщение  
Аналогично е за System.out.println(...);
```

Разликата между методите **print** и **println** се състои в това, че при използване на **print**, след изписване на съобщението или стойността на променливата, указателят остава на същият ред, а при използване на **println** след изписването, указателят се премества на следващия ред.

Както и при оператора за въвеждане на данни, за да можем да използваме тези методи, трябва преди това да посочим, че в класа ще се включват методи от библиотека `io` на Java. Това става отново чрез изписване на следният ред, преди започване описанието на класа:

```
import java.io.*;
```

Не е необходимо този ред да се записва за всеки отделен метод. Т.е. щом веднъж сме изписали реда преди описанието на класа, то в тялото на класа можем да използваме методите на библиотеката неограничен брой пъти.

Пр. 3

```
System.out.print("have a nice day"); // извеждане на съоб-  
// щение на екран.
```

```
System.out.println("Name:" + name1); //извеждане на  
//съобщение Name и текущата стойност на променлива name1 на  
//екран.
```

```
System.out.println(age+", "+name1+")."); // извежда на  
//екран стойностите на променливите age и name1, разделени със  
//запетая и завършва с точка.
```

Първа програма (клас) на Java

С помощта на изучените до момента възможности на езика Java ние вече можем да съставяме цялости програми (макар и само линейни).

Нека нашата първа програма да бъде реализация на алгоритъма за изчисляване на сумата на две числа, който съставихме в началото на този урок. С това, че вече имаме готов алгоритъм, работата ни е улеснена до голяма степен, но все пак преди да напишем програмата, е необходимо да уточним от какъв тип и колко на брой са променливите, които ще използваме, за да представим водните данни, междинните и крайните резултати на алгоритъма. В този случай е лесно да се определи, че са ни необходими две реални променливи – за основната и чистата заплата съответно. Ще ги наречем съответно OZ и CZ. Като използваме вече наученото за операторите за вход, изход и присвояване, лесно достигахме до следната програма (клас):

```
import java.io.*;
class first {
    public static void main(String[] args) {
        float MainPay=240, CPay=0;
        System.out.println("Основна заплата:"+MainPay);
        CPay = MainPay + MainPay*21/100;
        CPay = CPay - CPay*20/100;
        System.out.println ("Чиста заплата: " + CPay);
    }
}
```

Във всички случаи, когато се съставя компютърна програма, преди да се премине към нейното непосредствено написване, е необходимо да се уточнят: алгоритъмът на обработката; типът на участващите величини; променливите, представени в програмата и т.н.

Предварителният анализ, планиране и проектиране на програмата могат да се направят просто наум (при по-елементарните задачи), но в повечето случаи е необходимо да

отделите време и ресурси за тази цел. Направление на компютърната информатика, което се занимава с изучаването на тези въпроси, е софтуерното инженерство. Предварително на софтуерното инженерство са методите и средствата за планиране, анализ, проектиране, разработване и внедряване на компютърни програми.

ТЕМА 5

ВЗЕМАНЕ НА РЕШЕНИЯ

Разклонени алгоритми

Процесът на избор на един от няколко възможни варианта за действие нарича **вземане на решение**.

Какво решение ще вземем обикновено е свързано с това – дали е изпълнено или не някакво условие. Например ако се чудите дали днес да отидете на училище или не, то вероятно ще вземете решение според това дали броят на неизвинените ви отсъствия до момента не надхвърля максималния позволен брой. Подобни условия се наричат **логически условия**.

За представяне на такива условия в езиците за програмиране е въведен специален тип данни, наричан **логически (булев) тип данни**.

Алгоритми, които включват елементарни действия, съдържащи избор между няколко възможни варианта (т.е. моделиращи процес на вземане на решение) се наричат **разклонени алгоритми**.

Реализацията на разклонените алгоритми в езиците за програмиране се извършва чрез различни **оператори за разклонение**. В езика за програмиране Java има два такива оператора – условен оператор **if...else** (за избор между два възможни варианта) и оператор (за избор между произволен брой варианти).

Стандартният логически тип, както и операторите за разклонение улесняват в много голяма степен програмирането на разклонени алгоритми.

Логически (булев) тип

Множеството от стойности на логически тип съдържа само две константи – **true** и **false** (съответно истина и лъжа).

Логическият тип се декларира със стандартния идентификатор **boolean**. Например логическите променливи **B** и **Da** се декларират по следния начин:

```
boolean B, Da;
```

С логически величини могат да се извършват трите основни логически операции: двуаргументните – логическо умножение (което на Java се записва **&&**) и логическо събиране (на Java - **||**) и едноаргументната – логическо отрицание (**!** – на Java).

Операторите за сравнение дават **логически резултат**, въпреки че се прилагат върху други типове, основавайки се на сравнението на операндите – за равенство(=), за неравенство (!=), за по-малко (<), за по-голямо (>), за по-голямо или равно (>=), за по-малко или равно (<=).

Пр. 1. Логически изрази

1. Ако броят на неизвинените отсъствия, направени до момента от ученика, е **Neizv**, а максималния допустим брой е **MaxNeizv**, то логическото условие, от което се определя решението на ученика, ще изглежда така:

Neizv < MaxNeizv

2. Ако добавим още едно изискване към горното условие, напричер часът (**Chas**) да не е повече от 11 преди обяд, то ще получим по-сложен логически израз:

(Neizv < MaxNeizv) && (Chas <= 11)

За константите `true` и `false` е прието, че са подредени по следния начин: **`false` < `true`**, което означава, че и логическите величини може да се сравняват.

Логически стойности **не могат** да се въвеждат чрез **read**.

Условен оператор

Един от операторите, който представя вземане на решение в програмата, е условният оператор. Най-простият вариант има вида:

```
if (<условие>) <оператор>;
```

, където условието е израз, водещ към логически резултат. Операторът може да бъде произволен оператор на Java, включително друг оператор **if** или блок от оператори. **Ако е повече от един оператор, то задължително трябва да са заградени във фигурни скоби.**

Пълната форма на условния оператор е следната:

```
if (<условие>) {  
    <оператори 1>;  
} else {  
    <оператори 2>;  
}
```

, където действието е следното: ако е изпълнено логическото условие след **if** (т.е. има стойност **true**), то се изпълняват <оператори 1>, ако не (т.е. условието има стойност **false**)– се изпълняват <оператори 2>, записани след **else**.

Пр. 2. Правилно записани условни оператори:

1. Кратка форма:

```
if (Neizv<MaxNeizv)
```

```
Neizv = Neizv + 1; //може да се запише и така: //  
Neizv = ++Neizv
```

// не сме записали фигурни скоби т.к. има само 1 оператор

2. Пълна форма (предполагаме, че са направени декларации `int A,B; char D`):

```
if (A == B)  
    D = 'Д';  
else D = 'Н';
```

Когато е само един оператор, може и да не се пишат фигурни скоби.

3. Условен оператор, съдържащ друг условен оператор (предполага се, че са направени декларации `float X; boolean Sign`):

```
if (X<0) Sign = true;  
else  
    if (X>0) Sign = false;  
    else System.out.println ("стойността е 0");
```

4. Условен оператор със съставно условие (предполага се, че са направени декларации `int x; boolean y`):

```
if ((x>3) && (x<8)) y = true;  
else y = false;
```

Семантиката на условния оператор е следната: в зависимост от изпълнението на зададено условие се извършва (или не) едно определено действие (`if` в кратка форма) или се избира и изпълнява едно от двете записани действия (`if` в пълна форма).

Изпълнението на условния оператор в кратката му форма е следното: определя се стойността на логическото условие; само когато тя е `true`, се изпълняват оператори1. При пълната форма – ако стойността на логическото условие е `true`, се изпълняват оператори1, а ако е `false` - оператори2 след `else`.

- Когато има само един оператор в “оператори1”, то фигурните скоби могат да се пропуснат;
- Условният оператор **if** може да съдържа кой да е оператор на Java, в това число и друг оператор **if**. Трябва да се има в предвид, че **else** съответства винаги на най-близкия предхождащ **if**. В сложните случаи може да се сгреша и за това се препоръчва използването на отместване (Пр. 2.3.).

Примерна програма

При програмиране на разклонени алгоритми се спазват същите етапи, които илюстрирахме при нашата първа програма на Java.

Пр. 3. Съставяне на програма, която определя дали дадено яло число, въведено от клавиатурата, се дели на 5 или на 15.

Проектиране:

Началните данни на програмата се състоят само то една-единствена целочислена стойност, която се въвежда от клавиатурата. Ето защо в програмата по-долу се декларира променлива `IntNum` от цял тип. Крайният резултат е съобщение дали `IntNum` се дели или не на 5 или 15. За съхраняване на резултата в програмата не е необходима променлива.

Алгоритъмът, по който ще проверяваме делимостта на числото, е известен – едно число се дели на 5, ако остатъкът при целочисленото му деление на 5 е 0. Следователно дали едно число се дели или не на 5 може да се определи чрез проверяване на стойността на логическия израз $(IntNum \% 5) = 0$. Дали едно число се дели на 15 може да проверим, като установим дали то се дели едновременно на 5 и на 3.

Програмиране:

Следващата програма показва как може да се използва условния оператор `if`, за да програмираме разклонени алгоритми.

```

import java.io.*;
class Divisible {
public static void main(String[] args) {
    int IntNum=10;
    System.out.print("Числото "+ IntNum);
    if ((IntNum % 5) == 0)
    {
        System.out.print("се дели на 5 ");
        if ((IntNum % 3) == 0)
            System.out.print("и на 15");
        else System.out.println(" и не се дели на 15");
    }
    else System.out.println(" не се дели на 5 и 15");
}
}

```

ТЕМА 6

МНОГОВАРИАНТЕН ИЗБОР

Оператор за многовариантен избор

Съществуват средства, които подпомагат програмната реализация на разклонените алгоритми. Специално внимание ще обърнем на случаите, в които на определен етап от вземането на решение трябва да се избира между повече от два варианта. За целта ще разгледаме задача, която се свежда до подобен многовариантен избор на решение.

Задача1: Съставете програма, която въвежда последователно: първо реално число, знак за операция (един от '+', '-', '*', '/'), второ реално число и изчислява и извежда стойността на съответната операция. В случай, че е въведен недопустим знак за операция, програмата трябва да дава съобщение "Грешна операция".

Алгоритъм за решаване на задача 1 включва многовариантен избор на вземане на решение, защото

пресмятането може да се извърши по един от четири възможни варианта (в зависимост от това дали е въведен знак '+', '-', '*' или '/'). Програмирането на този етап включва използването на няколко условни оператора.

Пр. 1 Избор между четири възможни варианта, реализиран с условни оператори.

Ако стойностите на променливите number1 и number2 са двете въведени реални числа, а стойността на променливата operation е въведеният знак за операция, то пресмятането може да стане така:

```
if (operation == '*') result = number1 * number2;
else if (operation == '/') result = number1 / number2;
    else if (operation == '+') result = number1 + number2;
        else if (operation == '-') result = number1 - number2;
            else InvalidOperation = true;
```

За такива случаи е предвиден специален оператор (оператор за многовариантен избор), който улеснява програмирането на случаи, когато се налага да се прави избор между повече възможни варианти на действие. Операторът за многовариантен избор представя вземането на решения при наличието на сложни условия и прави програмата много по-ясна и четлива.

В езика за програмиране Java за многовариантен избор се използва оператора **switch**. Действията, между които се избира, са взаимноизключващи се (изпълнява се не повече от едно действие). Взаимноизключващите се действия ще наричаме **варианти**. Ето как изглежда реализацията на оператора **switch**:

```
switch ( <ключ> ) {
    case <вариант1>:<оператори>;
    ...
    case <вариантN>:<оператори>;
    default: <оператори>;
}
```

Стойността на ключа след **switch**, определя кой от вариантите трябва да бъде изпълнен т.е. всеки *вариант* се сравнява с **ключа** и ако съвпадне, се изпълняват *операторите след него*. За да изпълни само тях (и да не продължи с изпълнението на другите) след всеки вариант трябва да се запише служебната дума **break**;

Ролята на операторите след **default** е да посочат какво да изпълни програмата, ако случаят, който е въведен не съвпада с нито един от вариантите, посочени за многовариантния оператор.

Пр. 2 Оператор switch

1. Избор (между 4 варианта), определен от стойността на целочислена променлива I:

```
switch (I)
{
    case 1: y = x*x*x; break;
    case 2: y = Math.abs(x); break;
    case 3: y = 0; break;
    case 4: y = Math.pow(x,4); break;
    default: y = -1;
}
```

При изпълнение на този оператор , ако I има стойност 1, на y ще бъде присвоена стойността на израза $x*x*x$, ако I е 2, y ще получи стойността на $\text{Math.abs}(x)$ и аналогично – за стойност на I=3 или 4, ще се премине към изпълнението на оператора, следващ case. Ако е зададена друга стойност то на y ще се присвои стойност -1.

2. Избор (между четири варианта), определен от стойността на променлива grade от знаков тип:

```
switch (grade)
{
    case 'A': System.out.println ("Congratulations!");
    break;
```

```

    case 'B':System.out.println ("Not bad, B level is OK");
    break;
    case 'C':System.out.println ("C level is only average");
    break;
    case 'D':System.out.println ("D level is terrible");
    break;
    default:System.out.println("No excuses!Study harder!");
}

```

Примерна програма

Нека като пример за приложение на оператор да съставим програмат от задача 1.

Пр. 3 Програма за решаване на **задача 1**

Проектиране

Според условието на задача 1 аргументите и резултатът на операциите са реални числа, а знакът на операцията е от знаков тип. Ето защо в програмата се декларират променливите:

```

char operation;
float number1, number2, result;

```

Знакът на операцията може да бъде един от четирите '-', '*', '/', '+', като се избира между четири варианта. В този случай логично е да използваме оператора за многовариантен избор като най-удобно средство за програмиране:

```

switch (operation){
    case '*': result = number1 * number2; break;
    case '/': result = number1 / number2; break;
    case '+': result = number1 + number2; break;
    case '-': result = number1 - number2; break;
    default: InvalidOperation = true;
}

```

}

```
import java.io.*;
class ex_oper {
    public static void main(String[] args) {
        char oper='+';
        double number1=10, number2=20, result=0;
        boolean InvalidOperation = false;
        System.out.print("Въведените две числа и операцията са:");
        System.out.println(number1+", "+number2+", "+oper);
        switch (oper) {
            case '*': result = number1 * number2; break;
            case '/': result = number1 / number2; break;
            case '+': result = number1 + number2; break;
            case '-': result = number1 - number2; break;
            default: InvalidOperation = true;
        }
        if (InvalidOperation) System.out.println("Грешна операция !!!");
        else System.out.println("Резултатът
e: "+number1+oper+number2+"="+result);
    }
}
```

ТЕМА 7

ПОВТАРЯЩИ СЕ ДЕЙСТВИЯ

Циклични алгоритми

Един алгоритъм е цикличен, когато при неговото изпълнение група от елементарни действия се повтаря многократно. В много случаи се налага организиране на повтарящи се действия. Такава например е задачата за намиране на сумата на редица от числа. Сумиране на голям брой числа може да се наложи, когато искате да проверите сметката си в магазина или пък когато класният ръководител определя средния успех на учениците в класа.

Пр. 1 Алгоритъм за определяне на сума

Предложеният алгоритъм решава задачата за намиране на сумата на числа, въвеждани последователно, като сумирането

завършва, когато се въведе числото 0. сумата на числата се натрупва последователно (след всяко въвеждане) в параметър s. Следващото словесно описание представя типичен цикличен алгоритъм:

1. $s = 0$.
2. Въведете поредното число x.
3. Ако $x \neq 0$, то $s = s + x$, в противен случай изпълнете стъпка 5.
4. Изпълнете стъпка 2.
5. Съобщете стойността на s (като резултат).
6. Прекратете работа.

Групата от повтарящи се действия в цикличните алгоритми се наричат **тяло** на цикъла. В примера тялото на цикъла представляват действията от стъпки 2, 3 и 4. Всяко изпълнение (повторение) на тялото на цикъла се нарича **итерация**. Задължително изискване е повторенията да са краен брой. Кога ще завърши цикълът се определя от т.нар. **условие за край** на повторенията, което в горния алгоритъм е $x = 0$ (проверява се на стъпка 3).

За да сме сигурни, че цикълът наистина ще завърши (условието за край ще настъпи), е необходимо някоя от стъпките на цикъла да включва действие, което евентуално ще предизвика удовлетворяване на условието за край. Това действие се нарича **управление на повторенията**. В нашия пример то е определено със стъпка 2, защото нововъведената стойност на x може да е 0 и да предизвика приключване на изпълнението на алгоритъма.

Стъпка 1 от горния алгоритъм е много важна за правилно му изпълнение, защото ако на s не беше дадена предварително стойност 0, а например 1, тогава s, получено на стъпка 5, няма да е сумата на въведените числа. Такива действия, които се извършват преди началото на цикъла с цел да се подготви неговото правилно изпълнение, се наричат подготовка или **инициализация** на цикъла.

Средства, необходими за организация на цикли в програмите се наричат *оператори за цикъл*. С тяхна помощ можем да

накараме компютъра да изпълни голям брой повтарящи се действия, без да е необходимо да записваме всяко действие отделно.

1. Четирите основни елемента на всеки цикъл, въведени по горе – инициализация, тяло, управление на повторенията и условие за край, трябва внимателно да бъдат обмислени и проектирани, независимо по какъв начин ще се записва цикличният алгоритъм – словесно или чрез оператор за цикъл.
2. Забележете, че когато чрез цикъл в една променлива се натрупва сума, то тази променлива първоначално се нулира.

В езика Java има три циклични оператора. В този урок ще разгледаме два от тях, които се наричат съответно **цикъл с предусловие** и **цикъл със следусловие**. Двата оператора се различават помежду си по това дали условието за край на цикъла се проверява преди или след изпълнение на неговото тяло. Третият вид оператор за цикъл – с параметър, ще разгледаме в следващия урок.

Оператор за повторение с предусловие

Оператор за цикъл с предусловие **while** има следният вид:

```
while (<условие>) {  
    <оператор>;  
    [оператор2; ... операторN;]  
}
```

, където условието е от логически тип и определя условието за край на цикъл, а операторът (кой да е оператор от езика Java) представлява тялото на цикъла. Т.е. оператора се изпълнява докато условието от логически тип има стойност **true**, ако приеме стойност **false**, то оператора не се изпълнява повече.

Пр. 2 Оператор за цикъл с предусловие

Предполага се, че S и I са декларирани като целочислени променливи:

```
S = 0; I = 1;           // инициализация
while (I <= 4) {         // проверка за край
    S = S+I*I;           // тяло
    I = ++I;             // управление на повторенията
}
```

Смисълът на оператора **while** може много лесно да бъде разбран, ако го преведем и прочетем като свързано изречение – “**докато** <условие> има стойност истина, изпълнявай <оператор>”. Иначе казано: преди всяка итерация се изчислява стойността на логическото условие; ако тя е **true**, се изпълнява оператора, а ако е **false** – цикълът се прекратява.

Да разгледаме по-подробно изпълнението на оператора за цикъла от пример 2:

- Преди влизане в цикъла променливите S и I приемат начални стойност 0 и 1 (**инициализация**);
- При първата итерация се проверява дали стойността на I е по-малка или равна на 4, т.е. $1 \leq 4$, и понеже това е вярно, се изпълняват операторите от тялото на цикъла ($S = 0 + 1 * 1$, т.е. $S = 1$ и $I = 1 + 1$, т.е. $I = 2$);
- При втората итерация, понеже отново е вярно, че $2 \leq 4$, се изпълнява тялото ($S = 1 + 2 * 2$, т.е. $S = 5$ и $I = 2 + 1$, т.е. $I = 3$);
- Трета итерация: Понеже $3 \leq 4$, следва че: $S = 5 + 3 * 3$ ($= 14$), а $I = 3 + 1$ ($= 4$);
- Четвърта итерация: Проверката дали $4 \leq 4$ дава стойност истина и (за последен път) се изпълнява тялото на цикъла $S = 14 + 4 * 4$, т.е. $S = 30$ и $I = 4 + 1$ ($= 5$);
- При проверка дали $5 \leq 4$ се получава лъже и цикълът се прекратява.

1. Тялото на цикъла може да не се изпълни нито веднъж!
2. Основните елементи на цикъла от пример 1 са посочени с помощта на коментари.

За да помните колко е важно управлението на повторенията, разгледайте цикъла от пример 3, който довежда до т.нар. **зацикляне** (цикълът не завършва).

Пр. 3 Зацикляне

```
while (true) {System.out.println("Безкраен цикъл");}
```

Друг пример за циклична програма и използване на оператора **while** е даден в следващия пример:

Пр. 4 Основна част на програма за намиране на НОД по алгоритъма на Евклид

```
M = 23;           // инициализация
N = 42;
while (M != N)      // проверка на условието
{
    if (M > N) M = (M-N); // тяло
    else N = (N-M);
}
System.out.print("НОД на числата е: "+M);           // извеждане на резултат
```

Оператор за повторение със следусловие

Операторът за цикъл със следусловие **do...while** има следната синтактична диаграма:

```
do {
    <оператор>;
    [<оператор1>; [<оператор2>;...]];
} while (<условие>);
```

, където условието е от логически тип и определя условието за край на цикъла, а операторите между **do** и **while** представляват тялото на цикъла.

Пр. 5 Основна част на програма - цикъл със следусловие (алгоритъм на Евклид)

```
M = 56;           // инициализация
N = 33;
if (M != N) {
    do {           // тяло
        if (M > N) M = M - N; // управление на повторенията
        else N = N - M;
    }
    while (M == N); // проверка на условието за край
}
System.out.print(M + " "); // извеждане на резултат
```

Начинът на изпълнение на оператора за цикъл със следусловие се определя от следните две стъпки:

1. Изпълняват се последователно операторите между **do** и **while**.
2. Определя се стойността на логическото условие. Ако тя е **false**, се повтарят отново стъпки 1 и 2, а ако е **true** изпълнението на цикъла се прекратява.

Смисълът на оператора **do...while...** може да се изрази с изречението: “**Изпълнявай** операторите, **докато** стойността на условието стане истина.”

1. При цикъла със следусловие също може да се получи зацикляне, ако в тялото му няма оператор, който влияе върху условието за край.
2. Ако сравните двата начина за програмиране на алгоритъма на Евклид (пример 4 и пример 5), ще забележите, че в този

случай е по-удобно цикълът да бъде записан чрез оператор `while` (защо?).

3. Операторът **`do...while...`** много често се използва в програмите (вж. пример 6) за проверка на това дали въведените от потребителя данни са коректни (по стойност, по тип и т.н.).
4. Тялото на оператора **`do...while...`** се изпълнява поне един път.

ТЕМА 8

ПОВТАРЯЩИ СЕ ДЕЙСТВИЯ (ПРОДЪЛЖЕНИЕ)

Почти всички примери за цикли, разгледани в предишния урок, имат едно общо свойство: колко пъти ще се повтори изпълнението на тялото зависи от обработваните данни, т.е. при съставяне на програмата този брой е неизвестен.

Изключение в това отношение прави само цикълът от пример 2, тялото на който се изпълнява винаги четири пъти. За по-ефективно записване на цикли, за които предварително е известно колко пъти ще се изпълнят, в повечето езици за програмиране има специален оператор, наречен **цикъл с параметър**.

Тук по-конкретно ще разгледаме как изглежда този цикъл в езика за програмиране Java.

Оператор за повторение `for`

При оператора за цикъл с параметър тялото се изпълнява по веднъж за всяка една стойност от предварително определена редица от стойности на даден параметър.

Операторът с параметър има следният вид:

```
for(<израз1>; <израз2>; <израз3>) {  
    <оператор1>; [[оператор2]; ...; [операторN];] //тяло  
}
```

Където <израз1> е оператор за инициализация, <израз2> е логически израз, а <израз3> е оператор, изпълняващ се след тялото на цикъла. Всъщност <израз1> ни задава началната стойност на параметъра (трябва да е от дискретен тип), <израз2> - крайната стойност, а <израз3> - стъпката с която се изменя параметъра. Цикълът се изпълнява, като започва от началната стойност на променливата и докато логическият израз има стойност **true**, като параметъра се изменя със зададената стъпка в <израз3>.

Когато тялото съдържа само един оператор, то може да се опише без фигурни скоби, но ако са повече, то фигурните скоби са задължителни.

Пр. 1 Цикъл с параметър

Управляващата променлива е от целочислен тип а тялото на цикъла се състои от два оператора:

```
for (int I=0; I<=15; I++) {//докато I е по-малко или равно на 15  
    J= I*I; //на J присвоява стойност I*I, отпечатва  
тази  
        System.out.print(J+", "); //стойност и увеличава I с  
I.  
    }
```

1. Проверката за край на цикъла се извършва преди изпълнението на тялото, което означава, че цикълът може да не се изпълни нито веднъж т.е. оператор за цикъл с параметър е оператор с условие.
2. Началната и крайната стойност на параметъра се пресмятат еднократно в началото на цикъла.

Сравнение на операторите за повторение

За да не допускате грешки при програмирането на цикли, ще изброим някои прилики и разлики в записването и използването

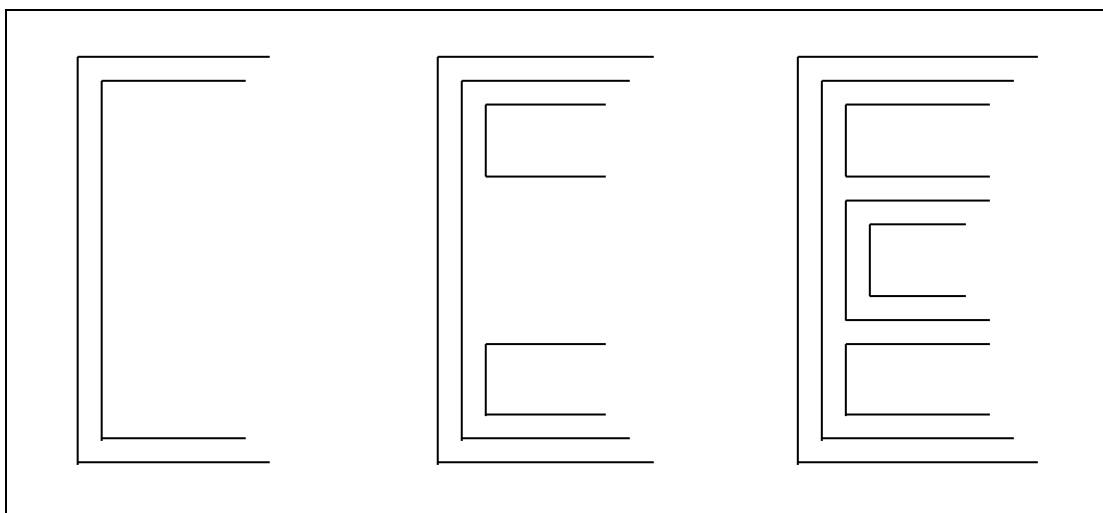
на различните оператори за цикъл, разгледани в предишните два урока:

1. Ако тялото на цикъл от типовете, изброени в предишните уроци, включва повече от един оператор, те трябва да се запишат във фигурни скоби.
2. Тялото на циклите **while** и **for** може да не се изпълни нито веднъж, докато тялото на **do...while...** се изпълнява поне веднъж.
3. Тялото на цикъл **while** се изпълнява дотогава, докато условието за край има стойност **true**, на **do...while...** – докато условието има стойност **false**, а на **for** – докато управляващата променлива не е надхвърлила крайната стойност.
4. При **while** условието за край се инициализира преди влизане в цикъла.
5. При **for** броят на повторенията на тялото е предварително известен, докато при операторите с условие (**while** и **do...while..**) той зависи от това кога ще се изпълни условието за край.

Вложени цикли

Ако тялото на един оператор за цикъл включва друг оператор за цикъл, се получава конструкцията вложени цикли. Синтаксисът на всички оператори за цикъл в езика Java допуска влагане не само на два, но и на повече цикли. Фигура 1 дава представа как изглеждат вложени един в друг оператори за цикъл.

Фигура 1.



Изпълнението на вложените цикли е следното: при всяка итерация на външен цикъл се изпълнява целият вложен непосредствено в него вътрешен цикъл; изпълнението приключва, когато се изпълнят всички итерации на най-външния цикъл.

Програма с три вложени цикъла е дадена в следващия пример.

Пр. 2 Програма за намиране на броя на “щастливите комбинации”

Шестцифрена комбинация от цифри се нарича “щастлива”, ако сумата от първите 3 и последните 3 цифри на комбинацията е равна на 13.

В програмата първоначално се преброяват всички трицифрени комбинации (броят им се натрупва в променливата Nhappy) със сумата на цифрите (C1, C2 и C3), равна на 13. след това броят на “щастливите” комбинации се получава, като този брой (Nhappy) се повдигне на квадрат (защо?).

За да получим всички трицифрени комбинации, е удобно да използваме три вложени цикъла – съответно за генериране на първата, втората и третата цифра. За всяка така получена комбинация, ако сумата от трите цифри е 13 увеличаваме брой на намерените до момента “щастливи” комбинации с единица ($Nhappy = ++Nhappy$).

```
import java.io.*;
class Happy {
    public static void main(String[] args) {
        byte C1, C2, C3;
        int Nhappy;
        Nhappy = 0;
        for (C1=0; C1 <= 9; C1++)
            for (C2=0; C2 <= 9; C2++)
                for (C3=0; C3 <= 9; C3++)
                    if (C1+C2+C3==13) Nhappy= ++Nhappy;
        Nhappy *= Nhappy;
        System.out.println("Щастливи са "+Nhappy+" комбинации от цифри");
    }
}
```

В програмата се използва оператор **for**, защото броят на повторение на всеки цикъл е известен предварително – той е равен на 10. Ще забележим, че тялото на най-вътрешния от трите вложени цикъла **for** ще се изпълни 1000 пъти.

1. С помощта на вложени цикли с малък брой оператори могат да се програмират значителен брой действия.
2. При влагане на цикли тялото на вътрешния цикъл трябва да се съдържа изцяло в тялото на външния.

ТЕМА 9

ОБОБОЩЕНИЕ: ОСНОВНИ ТИПОВЕ ДАННИ И ОПЕРАТОРИ

Алгоритми, ЕП и програми

Всеки **алгоритъм** задава начин за решаване на съответна задача. Алгоритмите се различават по възможните множества от входни данни, получаваните резултати, правилата за достигане на резултата и др. Алгоритмите се характеризират (и отличават помежду си) по седем параметри.

В зависимост от типа на съставлящите ги команди алгоритмите се разделят на три основни вида – **последователни (линейни), разклонени и циклични.**

Основните начини за **представяне на алгоритми** са чрез използване на естествени езици (словесно описание), графични изображения (блок-схемни описания) и различни специализирани езици.

ЕП (езици за програмиране) са специализирани езици, предназначени за описание на алгоритми, които ще бъдат изпълнявани от компютър.

Програмата представя алгоритъма във форма и вид, които могат да се възприемат и изпълняват от компютърна система.

Най-общо програмата се състои от две части – част за описание на данните (определя типа на данните, техните имена и вид – константи или променливи) и част за описание на тяхната обработка (състои се от оператори, представящи обработката на данните).

Типове данни

Всеки тип данни може да се разглежда като съвкупност от следните **три множества**:

- множество от **стойности** (константи на типа), които могат да се присвояват на променливите от типа;
- множество от **операции**, в които могат да участват величините (константи и променливи) от типа;
- множество от **отношения** (релации) между величините (константи и променливи) от типа.

В уроци 1-8 разгледахме така наречените **прости типове данни**. Стойностите на тези типове са неделими (за разлика от стойностите на структурните типове данни). Най-използваните прости типове данни (цял, реален, знаков и логически)

обикновено са вградени в универсалните ЕП и могат да се използват непосредствено в програмите. Такива типове се наричат **стандартни типове данни**. Освен това в ЕП на програмиста се предоставят и средства за създаване на собствени прости типове данни на базата на стандартни типове данни.

Стандартните прости типове в езика Java са реален (**float**; **double**), целочислен (**byte**; **short**; **int**; **long**), знаков (**char**) и логически (**boolean**).

Ограниченият тип в Java може да се определя от програмиста.

Величините от прост тип имат стойности, които са неделими. За разлика от тях стойностите на величините от структурен тип могат да се разглеждат като съставени от повече стойности, наречени компоненти или елементи на съответната съставна стойност. Всяка от компонентите на съставната стойност принадлежи на друг, предварително определен тип.

Възможно е всички компоненти на един структурен тип да са от един и същи тип, но има и структурни типове, чиито компоненти могат да принадлежат на различни типове.

Изрази и приоритет на операциите

Изразът в програмата е правило за определяне на стойност от определен тип чрез комбиниране по подходящ начин на величини от различни типове, допустими операции с тях и кръгли скоби. В изразите като аргумент могат да участват и стойности на функции. Пример за такъв израз е: `a + Math.pow(x,2)`.

При срещане на израз в програмата неговата стойност се намира с отчитане на приоритета на срещаните операции. Приоритетът на операциите в езика Java (в намаляващ ред) е, както следва:

- постфиксни – `<израз>++`, `<израз>--`;
- унарни - `++<израз>`, `--<израз>`, `+<израз>`, `-<израз>`, `!` `<израз>`;
- мултипликативни - `*`, `/`, `%`;

- адитивни - +, -;
- отношения - <, <=, >, >=, instanceof;
- еднквост - ==, !=;
- логическо “И” - &&;
- логическо “ИЛИ” - ||;
- присвояване - =, +=, -=, /=, %=;

Оператори

Операторите в ЕП задават определена обработка на данните. Същевременно някои от тях дават възможност да се определи редът на изпълнение на други оператори.

При съставянето на линейни програми обикновено се използват следните оператори: оператор за **въвеждане** на данни, оператор за **извеждане** на резултати и оператор за **присвояване**. В ЕП Java съответните оператори са System.in.read(), System.out.print() и “=” и са представители на т.нар **елементарни (прости) оператори**.

Структурните оператори, за разлика от простите, могат да включват в тялото си и други оператори.

За да се обедини **последователност** от оператори в Java, се използват фигурни скоби “{...}”.

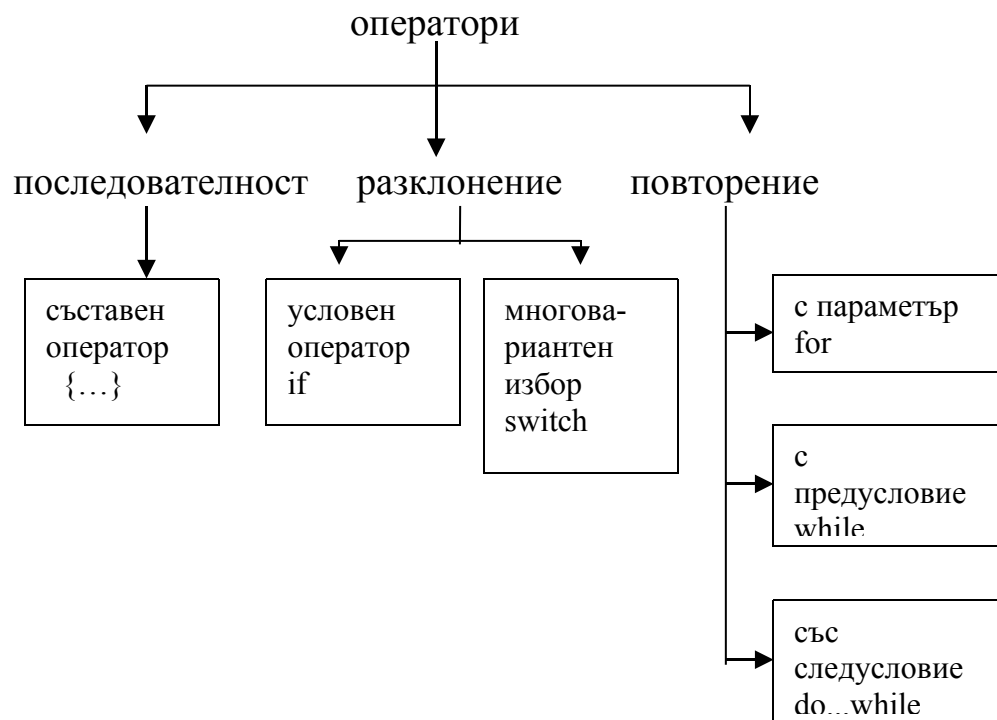
За реализация на разклонени алгоритми в ЕП са предвидени различни **оператори за разклонение**. В Java има два оператора – условен оператор **if** (за избор между два възможни варианта) и оператор **switch** (за избор между произволен брой варианти).

ЕП предлагат удобни средства за организация на **цикли**. Това са т.нар. оператори за цикъл, с помощта на които можем да накараме компютъра да изпълни голям брой повтарящи се действия, без да е необходимо да записваме всички извършващи се действия.

В езика за програмиране Java има три оператора за цикъла – цикъл с предусловие **while**, цикъл със следусловие **do...while** и цикъл с параметър **for**.

На фигура 1 са представени основните оператори в Java:

Фиг. 1:



За да можете да използвате един оператор в програма, е необходимо да знаете две неща – как се записва (синтаксис) и как се изпълнява (семантика).

Следващите два примера включват почти всички изучени до момента елементи на език за програзиране Java.

Пр. 1 Програма реализираща разклонен алгоритъм

Да се състави програма на Java, която да пресмята обиколката (периметъра) и лицето на триъгълник по зададени дължини на страните *a*, *b* и *c* и да определя вида на триъгълника – разностранен, равнобедрен, равностранен.

Проектиране

Програмата изисква три променливи *a*, *b* и *c* от реален тип, в които ще се съхраняват дължините на страните на триъгълника. Въвеждаме и логическа променлива *Flag*, стойността на която ще се определя в зависимост от това дали съществува триъгълник с дължини на страните равни на въведените стойности *a*, *b* и *c*.

Променливата p е предназначена за съхраняване стойността на полупериметъра, а S – на лицето на триъгълника. S и p са от реален тип.

Ще използваме известните формули от математиката за изчисляване на полупериметъра и лицето на триъгълника по Херонова формула:

$$p = \frac{a + b + c}{2}$$

$$S = \sqrt{p(p-a)(p-b)(p-c)}$$

След въвеждане на стойностите на a , b и c се извършва проверка съществува ли триъгълник с дължини на страните a , b и c . Видът на триъгълника се определя чрез сравняване на дължините на неговите страни.

Програмиране

```

import java.io.*;
import java.awt.*;
class triangle {
    public static void main(String[] args) {
        double a=12.5, b=13.5, c=17, p=0, S=0;
        boolean Flag;
        System.out.println("Въведена стойност за a="+a);
        System.out.println("Въведена стойност за b="+b);
        System.out.println("Въведена стойност за c="+c);
        Flag = ((a>0)&&(b>0)&&(c>0)&&((a+b)>c)&&((a+c)>b)&&((b+c)>a));
        if (Flag)
        {
            p = (a+b+c)/2;
            S = Math.sqrt(p*(p-a)*(p-b)*(p-c));
            System.out.println("Периметърът е: "+ 2*p);
            System.out.println("Лицето е: " + S);
            if ((b==c)&&(a==b)) System.out.println("Триъгълникът е равноностранен");
            else if ((a==b)||((b==c)||((c==a)))System.out.println("Триъгълникът е
равнобедрен");
                else System.out.println("Триъгълникът е разностранен");
            }
        else System.out.println("Не съществува триъгълник със страни:"+a+b+c);
        }
    }
}

```

Пр. 2 Програма, реализираща цикличен алгоритъм

Да се отпечата на екран таблицата за умножение на числата от 10 до 20.

Проектиране

Ще използваме два вложени цикъла, за да можем да изведем на екран всички възможни произведения. Също така, ще ни бъдат необходими две целочислени променливи I и J, които ще бъдат броячи.

Програмиране

```
import java.io.*;
class pr_2 {
    public static void main (String [] args) {
        for (int I=10; I<=20; I++)
            for (int J=10; J<=20; J++) {
                System.out.println(I+"*"+J+"="+"I*J");
            }
    }
}
```

ЧАСТ 2

ПРОГРАМИРАНЕ НА ОСНОВНИ АЛГОРИТМИ

Тема 10: Наследяване

Тема 11: Интерфейс

Тема 12: Подалгоритми и подпрограми

Тема 13: Тип масив

Тема 14: Изключителна ситуация

Тема 15: Работа с текстове

Тема 16: Рекурсивни алгоритми

Тема 17: Обработване и съхранение на данни

ТЕМА 10

НАСЛЕДЯВАНЕ

В езика Java (както в останалите обектно-ориентирани езици за програмиране) класовете могат да се наследяват от други класове. Така веднъж създадени, класовете могат да се използват като база за дефиниране на нови класове, които слабо се отличават от първите. Новите класове, наследяват елементите на съществуващите и могат да променят поведението им с дефиниране на методи със същите имена.

Работа с повече от един клас

Реалните програми използват повече от един клас за решаване на определена задача. Възможни са два подхода за вмъкване на класове – всеки клас да бъде в нов файл или няколко класа да са в един файл. Двата подхода имат следните различия:

1. Няколко класа в един файл:

В този случай само един от класовете може да има име, съвпадащо с името на файла. Именно този клас ще бъде видим от другите файлове и се нарича **основен**, а останалите класове са **помощни**. За да са видими, те трябва да бъдат декларирани със служебната дума **public**.

Пр. 1

// начало на Example.java

```
class Example {  
    int a;  
    void main (String[] arg) {  
        Extra e = new Extra();  
        ...  
    }  
}  
class Extra {  
    int x;  
    ...  
}
```

// край на Example.java

Въпреки че класът Extra е дефиниран след основния клас Example, той може да бъде използван в него.

2. Класове в различни файлове:

Възможно е всеки клас от една програма да се разполага в отделен файл с име, съвпадащо с името на класа. Така класовете по подразбиране са видими.

Наследяване на класовете

Един Java клас може да се обяви като наследник на някакъв друг клас. Това става чрез служебната дума **extends**.

Пр. 2

```
class Base {  
    ... //определяне на данни и методи  
}  
class New extends Base {  
    ... //определяне на данни и методи  
}
```

В новия клас New се съдържат всички данни и методи, определени в класа Base и още някои собствени декларации. Така новият клас се явява разширение на класа Base. Предимството на пораждање на класове един от друг е очевидно – възможност за многократно използване на вече въведена програма.

Пр. 3

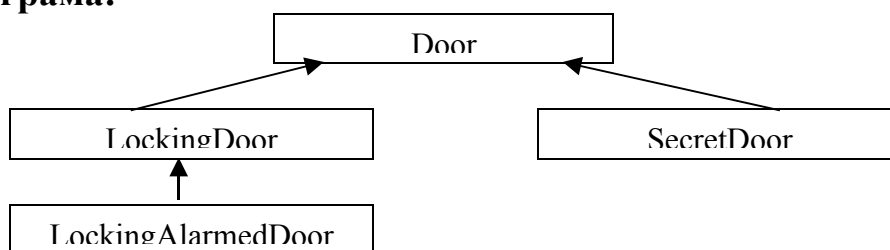
```
class Door { // описание на обикновена врата  
    boolean stateOpen;  
    void open() {stateOpen = true; }  
    void close() {StateOpen = false;}  
}  
class LockingDoor extends Door // описание на врата, която  
//се заключва с ключ. Автоматично се наследяват  
(включват)  
    // stateOpen, open() и close()  
    boolean stateLocked;  
    void lock() { stateLocked = true;}  
    void unlock() { stateLocked = false;}  
class SecretDoor extends Door { //описание на врата, която се  
    //заключва с код. Автоматично се наследяват (включват)  
    //stateOpen, open и close
```

```

...
}
class LockingAlarmedDoor extends LockingDoor {//описание на
    //врата ...Автоматично се наследяват (включват)
    // stateOpen, open, close, stateLocked, lock, и unlock
    ...
}

```

Наследяването от примера е онагледено на следната диаграма:



Суперклас и подклас

Когато се дефинира наследяване един клас се явява наследник на друг. Както казахме, в този случай първият включва в себе си всички елементи на втория. Вторият се нарича **суперклас** или базов, а първият – **подклас** или извлечен. Наследяването се задава при декларация на подкласа със служебната дума **extends**.

Пр. 3

```

class Base {
    //елементи на базовия клас (суперклас)
}
class New extends Base {
    //елементи на извлечения клас (подклас)
}

```

При наследяването се добавят нови членове данни и/или методи, като се запазват и всички данни и методи на суперкласа. Методите и данните на суперкласа се наследяват в подкласа.

Елементите на суперкласа се използват в подкласа директно със задаване на тяхното име. Когато има елементи на класа и подкласа с едно и също име, за достъп до елементите на класа се използва служебната дума **super**.

Пр. 4

```
class Base {
    int x, y;
}
class New extends Base {
    int z, y;
    void example() {
        x = 1;           // x от суперклас Base
        y = 1;           // y от извлечен клас New
        super.y = 1; // достъп до елемент y на суперклас
                      //Base
        z = 1;           // от извлечен клас New
    }
}
```

Достъп

Пред членовете данни и подпрограмите на класа може да се използва модификатор за достъп. Съответно:

- **Скрит достъп: private.** С този модификатор един елемент се обявява за скрит за всички методи, които са извън класа, в който е елемента;

Пр. 5

```
class Alfa {
    private int privateDatum;
    private void privateMethod() {
        System.out.println("Method from Alfa");
        ...
    }
}
```

```

class Beta {
    void example() {
        Alfa e = new alfa();
        e.privateDatum = 1; // невъзможно!
        e.privateMethod(); // невъзможно!
        ...
    }
}

```

- Видим достъп: **public**. С този модификатор елемента се обявява за видим и достъпен от всички външни програми. Видимите променливи могат да бъдат четени и да се променя тяхната стойност. Поради тази причина много рядко променливите имат такъв модификатор.

Пр. 6

```

class Alfa {
    private int privateDatum;
    private void privateMethod() {
        System.out.println("Method from Alfa");
    }
    public int publicDatum;
    public void publicMethod() {
        System.out.println("Method from Alfa");
    }
    ...
}

class LetterA {
    void example() {
        Alfa e = new Alfa();
        e.privateDatum = 1; // невъзможно!
        e.privateMethod(); // невъзможно!
        e. publicDatum = 2; // възможно
        e. publicMethod(); // възможно
        ...
    }
}

```

```

    }
}

```

- **Защитен достъп: `protected`.** Това е комбинация между скрития и видимия достъп. Всеки елемент, който е защитен, е видим в собствения си клас и във всички негови преки или косвени подкласове, а е скрит за всички останали части на програмата.

Пр. 7

```

class Alfa {
    private int privateDatum;
    private void privateMethod() {
        System.out.println("Method from Alfa");
    }
    protected int protectedDatum;
    protected void protectedMethod() {
        System.out.println("Method from Alfa");
    }
    ...
}

class LetterA extends Alfa {
    void example() {
        Alfa e = new Alfa();
        e.privateDatum = 1;    // невъзможно!
        e.privateMethod();    // невъзможно!
        e.protectedDatum = 2; // възможно
        e.protectedMethod();  // възможно
        ...
    }
}

```

ТЕМА 11

ИНТЕРФЕЙС

Освен затварянето на данните и програмите в клас и наследяване на класовете, третата характеристика на обектно-ориентираното програмиране е полиморфизмът. Това е използване на един и същ програмен код за еднакви или подобни действия върху различни данни. Той се реализира при обръщение към метод от обект и в зависимост от типа на параметрите и самия обект се изпълнява един или друг програмен код.

Полиморфизъм

Съществуването на две или повече функции с еднакви имена, които извършват еднакви (подобни) действия се нарича полиморфизъм. Той е свързан с предефиниране на функцията. Предефинирането може да стане статично, в рамките на един клас (статичен полиморфизъм) или динамично, при наследяване (динамичен полиморфизъм).

Статичен полиморфизъм – да се предефинира (статично) един метод (overload) означава да се предостави негова друга реализация с използване на различен брой или тип на параметрите. В един клас могат да се дефинират методи с едно и също име, но сигнатурата им трябва да е различна. За да се считат за един метод, реализиращ статичен полиморфизъм, методите с еднакви имена трябва да връщат един и същ тип.

Пр. 1

```
class Equality(){
    static boolean intEqual(int i, int j){return i==j;}
    static boolean intEqual(int i, double d){return i==(int)d;}
    static boolean intEqual(double d, int j){return (int)d==j;}
    static boolean intEqual(double d, double e){return (int)d==(int) e;}
    static boolean intEqual(int i, char c){return i==(int)c;}
    static boolean intEqual(char c, int j){return (int)c==j;}
```

```

static boolean intEqual(char c, char d){return (int)c==(int)d;}
}
//използване
if(Equality.intEqual(x, y)){//в зависимост от типа на x и y ще
//се извиква метод
...
}

```

Динамичен полиморфизъм – да се предефинира (динамично) или преопредели един метод (override) означава да се предостави нова реализация на дадения метод, различна от наследената от суперкласа реализация. Новата реализация в подкласа има същото име, същия брой и тип на параметрите и връща същия резултат като реализацията на метода в подкласа.

Пр. 2

```

import java.io.*;
class Base {
    int a = 1;
    void printing() {
        System.out.println("Печата в клас Base");
    }
    void basing(){
        System.out.println("Метод на класа Base");
        printing();
        System.out.println("Стойността на променливата е:" +a);
    }
}
class New extends Base{
    int a = 2;
    void printing(){
        System.out.println("Печата в клас New");
    }
}

```

```

class OverrideTest {
    public static void main(String[]) throws IOException {
        System.out.println("Използваме обект на клас Base...");
        Base b = new Base();
        b.printing();
        b.basing();
        System.out.println("Използваме обект на клас Newt...");
        New n = new New();
        n.printing();
        n.basing();
    }
}

```

Един клас може да бъде дефиниран като последен със служебната дума **final**. Такъв клас не може да бъде наследяван, т.е. не може да има подкласове. Създаването на класове **final** позволява защита на информацията. Методите от тип последен също се дефинират със служебната дума **final** и не могат да се преопределят в подкласовете.

Абстрактни методи и класове – това са методи, които са обявени в един клас, но не са реализирани в него. Такъв клас се нарича абстрактен, явявайки се всъщност само шаблон. Действиетелната реализация на метода се предоставя на подкласовете посредством преопределяне. В Java за указване на абстрактен метод се използва ключовата дума **abstract**.

Пр. 3

```

abstract class Shape {
    //абстрактния метод getArea() се реализира от
    //подкласа
    public abstract double getArea();
    ...
}

```

Класът Shape е обявен с ключовата дума **abstract**, показваща, че даденият клас включва в себе си един абстрактен (или нереализиран) метод. Това означава, че в действителност не може да бъде създаден обект от класа Shape, а само обекти от подкласовете, които осигуряват реализацията на метода `getArea()`.

Класът, съдържащ абстрактни методи се нарича абстрактен, в противен случай – функционален.

Интерфейс – това е начин на задължаване на класовете да имат определени свойства. Той по структура прилича на класа, като включва полета и методи, но се различава от него по това, че не може да се създават негови екземпляри. Методите в интерфейса не трябва да се реализират, а полетата могат да бъдат само константи.

На интерфейса може да се гледа като на абстрактен клас, съдържащ само абстрактни методи и константи.

Пр. 4

```
interface Helpable {  
    public void help();  
}
```

Интерфейсът в Java се използва, когато се създава клас, който го реализира. Реализацията означава включване на реализации на всички методи на интерфейса в класа. За да се укаже, че дадения Java клас реализира методите на определения интерфейс, се използва служебната дума **implements**. Всеки клас може да реализира неограничен брой интерфейси.

Пр. 5

```
class Topic implements Helpable {  
    String strNameTopic;  
    public void help() {  
        System.out.println("Помощ за "+strNameTopic);  
    }  
}
```

```
}  
}
```

Интерфейсите могат да се наследяват и синтаксисът на интерфейса в общия случай е следния:

```
[public] interface <име> [extends <интерфейс>...] {  
    //тяло, съдържащо заглавни редове на методи и константи  
}
```

Където тялото съдържа **само** заглавни редове на методи и константи, а модификаторът за достъп `public` се използва, ако интерфейсът ще бъде използван извън пакета. За разлика от класа, всеки интерфейс може да наследява много интерфейси. (Класът може да наследява един клас, но да реализира много интерфейси.)

В интерфейсът може да се използва в качеството на тип данни. Може да се обявяват променливи от тип интерфейс. Всеки обект от клас, реализиращ интерфейса, може да се присвои на присвои на променлива от типа на интерфейса.

ТЕМА 12

ПОДАЛГОРИТМИ И ПОДПРОГРАМИ

Отделните части на един алгоритъм могат да бъдат отделени и представени като подалгоритми. В случаите, когато се налага група от едни и същи действия да се изпълнява на различни места в алгоритъма, тяхното оформяне като подалгоритъм може да съкрати в голяма степен усилията по съставянето на този алгоритъм и да съкрати записването му. Един алгоритъм се разделя на подалгоритми в два случая – когато е сложен и разделянето улеснява неговото съставяне и програмиране, или

когато разполагате с готови подалгоритми, които можете да използвате.

Пр. 1 Подреждане на три реални числа в намаляващ ред

Алгоритъмът за решаване на задачата е следният:

1. Въведете три реални числа и ги запомнете (в променливи a , b и c).
2. Ако $a < b$, разменете стойностите на a и b .
3. Ако $b < c$, разменете стойностите на b и c .
4. Ако $a < b$, разменете стойностите на a и b .
5. Изведете стойностите на a , b и c (вече подредени).
6. Край на работата.

В алгоритъма се повтарят многократно две действия – сравняване и размяна на стойностите на две променливи. Следователно, ако предварително съставим тези подалгоритми, можем да ги използваме за описанието му.

Коя е основната разлика между един подалгоритъм и алгоритъма, който го използва? В повечето случаи основният алгоритъм получава от потребителя стойности за началните данни и съобщава крайните резултати отново на потребителя. Докато подалгоритъмът обикновено получава началните си данни от основния алгоритъм и му връща крайните си резултати. Например подалгоритъмът за сравняване на две числа получава стойностите, които се сравняват, от основния алгоритъм, а резултатът от сравнението се връща отново към основния алгоритъм, за да се използва в следващите му действия.

Подпрограми

Програмата, реализираща основния алгоритъм за решаването на определена задача, се нарича главна (или основна) програма.

Всяка подпрограма, съдържаща се в главната програма, реализира подалгоритъм на основния алгоритъм. Изпълнението на подпрограмата се предизвиква с указване на нейното име и списък

от съответни параметри. Подпрограмата може да се изпълни за различни стойности на входните данни (зададени чрез параметри от списъка) и да върне различни резултати (отново зададени в същия списък).

Например ако подалгоритъмът за размяна на стойностите на две променливи е реализиран като подпрограма, то неговите параметри трябва да са два – променливите, чиито стойности се разменят.

Основната програма обикновено си взаимодейства с потребителя за въвеждането и извеждането на данни, докато подпрограмите ползват данни от главната програма и връщат в нейната среда получените резултати.

- Възможно е някои подпрограми, подобно на основната програма, да ползват описани в тях други подпрограми.
- Вече създадена и проверена, една подпрограма може да се използва в различни програми.

Подпрограми в Java

В езика Java се допускат два вида подпрограми, наречени съответно **процедури** и **функции** т.е. подпрограмите, това са всъщност **методите**, реализиращи съответния алгоритъм.

Подпрограмите-функции служат за описание на подалгоритми, при изпълнението на които се получава един единствен резултат, който се присвоява на функцията.

Подпрограмите-процедури могат да се използват за получаване на произволен брой резултати, но самата процедура не приема стойност.

Методите на класа имат следният синтаксис:

```
<служебна дума> тип <име> ([параметър1];[параметър2] ;...])  
{  
    тяло;  
}
```

Служебната дума може да бъде една или повече.

В следващата таблица ще обясним приложението на служебните думи:

Когато се посочва тип се реализира **подпрограма-функция**, а типът се нарича тип на функцията (т.е. типът на резултата, който ще бъде върнат). За да бъде върнат резултат на функцията, то в тялото ѝ, трябва да зададем **return(<израз>)**. Когато връщаме число, като стойност на функцията, не е необходимо да пишем скоби.

Когато вместо тип се използва служебната дума **void** – се реализира **подпрограма-процедура**. Служебна дума **void**, показва че на самият метод не се присвоява стойност т.е. това е подпрограма-процедура.

Структурата на подпрограмата е аналогична на структурата

Служебна дума	Приложение
public	Показва, че метода е достъпен навсякъде, където е достъпен класа, в който е описан.
private	Показва, че метода не е достъпен за подкласовете.
protected	Показва, че метода е достъпен само за класа, неговите подкласове и пакета.
private protected	Показва, че метода е достъпен само за класа и неговите подкласове.
final	Показва, че метода не може да се предефинира в подклас.
abstract	Показва, че метода няма тяло (реализация).
native	Показва, че реализацията на метода е направена на език, различен от Java.
synchronized	Показва, че в един момент само една програма може да изпълнява дадения метод.

на основната програма.

Параметрите в скобите се наричат **формални параметри**. Те служат за задаване на входните (началните) и изходните данни

(резултати) на подпрограмите. Поради тази причина тези параметри се наричат формални. Обикновено резултатът на функцията е само една стойност. За име на тази стойност се счита името на самата функция, а списъкът от формални параметри на функцията определя само входните данни.

Параметрите, които дефинираме в тялото на подпрограмата се наричат **локални параметри**, а тези, дефинирани в тялото на класа – **глобални**.

Ако в подпрограма искаме да използваме някой от глобалните параметри, то трябва да укажем това с помощта на служебната дума **this**.

Пр. 2 Нека е даден следният фрагмент от програма:

```
class Fragment {
    int x; // глобални параметри
    int y;
    int z;
    ...
    void Exampl (int x; int a) {
        //локален параметър с име като на
        // глобален и др. локален параметър

        x=2; //присвоява стойност на локален
            //параметър
        a=3;      //присвоява стойност на локален
                  // параметър
        this.x=7; //присвоява стойност на глобален
                  // параметър
        this.y = 1; //присвоява стойност на глобален
                    //параметър
        this.z = 0; //присвоява стойност на глобален
                    //параметър

        ...
    }
}
```

```

    }
    ...
}

```

Пр. 3 Деклариране на процедура и функция:

1. Деклариране на **процедура**, която разменя стойностите на две променливи a и b:

```

void change_a_b (float a, float b) {
    float buf;
    buf = a;
    a = b;
    b = buf;
}

```

, но смяната на тези два локални параметъра не засяга глобалните параметри, от които те са взели стойностите си.

2. Деклариране на **функция**, която сравнява две реални числа x и y, и ако $x < y$ връща стойност **true**, а в противен случай - **false**:

```

boolean Little (float x, float y) {
    return (x < y);
}

```

Допустими са подпрограми и без параметри. Например процедура, която отпечатва на екран текст “Добър ден” няма нужда от входни данни и не връща резултат, който да се използва от основната програма.

Изпълнение на подпрограми

Дефинирането на една подпрограма определя какво трябва да е нейното действие, но не и кога (при изпълнение на главната програма) да се изпълни това действие.

Изпълнението на подпрограма става чрез нейното извикване от главната програма. Това става чрез името на подпрограмата и списък от **фактически параметри**, за които тя трябва да бъде изпълнена.

Фактическите параметри, при извикване на подпрограмата, трябва да са толкова на брой, колкото са формалните параметри при нейното описание, като се запазва съответствието между типовете в реда на изброяването им.

Ако в програмата е указано активиране на подпрограма, то на това място се изпълняват операторите от блока на подпрограмата, като формалните параметри се заместват със зададените фактически параметри, т.е. ролята на фактическите параметри е да получат конкретните стойности на входните и изходни данни на подпрограмата.

Пр. 4 Активиране на процедури и функции:

```
1. void main() {  
    boolean b;  
    float x = 23.45;  
    float y = 23.455;  
    ...  
    b=Little(x, y); //извикване (активиране) на функция  
    //Little и присвояване на нейната стойност на променлива b  
    ...  
}
```

2. Ако искаме да разменим местата на две променливи T и P, то активирането на процедурата ще бъде:

```
change_a_b(T, P);
```

3. Ако искаме да отпечатаме резултата от сравнението на две числа 2*L и K:

```
System.out.println(Little(2*L, K));
```

Заместването на формалните параметри на подпрограмите с различни фактически параметри при активиране на подпрограмата

се нарича *механизъм за предаване на параметри*. Механизмът за предаване на параметри позволява подпрограмата да се изпълнява за различни входни данни и да получава различни изходни резултати.

Пр. 5 Примерна програма за подреждане на три реални числа в намаляващ ред:

```
import java.io.*;
class order {
    boolean Little (float x, float y) {
        return (x<y);
    }
    public static void main(String[] args) {
        float a=15, b=8, c=11;
        System.out.println("Въведените 3 реални числа са : "+a+", "+b+", "+c);
        order NewObj=new order();
        if (NewObj.Little(a, b)) {
            float buf;
            buf = a;
            a = b;
            b = buf;
        }
        if (NewObj.Little(b, c)) {
            float buf;
            buf = b;
            b = c;
            c = buf;
        }
        if (NewObj.Little(a, b)) {
            float buf;
            buf = a;
            a = b;
            b = buf;
        }
        System.out.println("Числата, подредени в намаляващ ред
са: "+a+", "+b+", "+c);
    }
}
```

Масивът е един от най-използваните в програмирането типове данни. Почти винаги, когато в програмата се появява необходимост от работа с редица еднотипни стойности се въвежда и съответен масив.

Пр. 1 Необходимост от обработване на редица от еднотипни данни

В програма се обработва средният успех на учениците от един клас (например с цел намиране на средния успех на целия клас или определяне на ученика с най-висок среден успех). Масивът е удобно средство за представяне на редицата от данни за средния успех, а именно – всеки елемент на масива ще съответства на средния успех на един ученик. Ще отбележим следните три характеристики на масива: а) елементите на масива са **краен брой** (в случая - равен на броя на учениците в класа); б) елементите на масива са от **един и същи тип** (в разглеждания пример средният успех на всеки ученик е реално число) и в) елементите на масива са **подредени** по някакъв признак (тук например може да считаме, че елементите са подредени според номерата на учениците).

Масивът е структурен тип данни, чиито стойности се представят с крайна редица от еднотипни стойности.

На всеки елемент на масива може да се съпостави неговият пореден номер в установената наредба между елементите, наречен **индекс** на елемента. Индексът определя мястото на елемента в масива. В случая с учениците на всеки среден успех се съпоставя поредният номер на ученика (таблица 1).

Номер в класа (индекс)	1	2	...
Среден успех	4,66	6.00	...

Възможно е елементите на масив да са подредени по два и повече признака. Така например оценките на всеки ученик в класа

могат да се подредят по номера на ученика и по изучавания предмет (таблица 2). Броят на признаците, по които са подредени елементите на масива, се нарича **размерност** на масив.

При двумерните масиви мястото на елемента в масива се определя от два индекса. Например в таблица 2 оценката 3.00 съответства на двойката индекси **1** и **Английски език**.

Таблица 2:

Номер в класа	Учебен предмет			
	Английски език	Биология	Информатика	...
1	3,00	5,00	6,00	...
2	5,00	6,00	6,00	...
3	4,00	5,00	5,00	...
...

Едномерните и двумерните масиви са най-използваните структури от еднотипни данни в ЕП.

Масивът в Java е променлива и като такава, преди използването му трябва да бъде деклариран. Декларацията може да се извърши по два начина – тип, следван от [] или тип и име, следвано от []. Тази декларация не заделя памет за масива. Заделянето се извършва с помощта на оператора **new** т.е. масивът в Java е обект.

Стъпките за създаване на масив са три: деклариране, заделяне на памет и нейната инициализация.

Пр. 1

```
int [] ArrInteger = new int[5];
```

Ако се извърши само декларация на името на масива, в паметта се заделя място, което се инициализира по-късно с оператора **new**. Декларирането на променливата на масива и заделянето на памет могат да се извършат с отделни оператори.

Пр. 2

```
int [] ArrInteger;      //деклариране на променливата масив  
ArrInteger = new int[5]; //заделяне на памет за 5 елемента
```

За достъп до елемент на масива се използва индекс, като индексирването започва от 0. Следователно последният елемент има индекс, равен на размера на масива минус единица. Тогава последният елемент на масива от Пр. 2 ще има индекс 4. За да се обърнем към елемент на масива, се изписва името на масива, последвано от квадратни скоби с индекса на елемента, към който искаме достъп.

Пр. 3

```
ArrInteger[2];
```

Инициализацията на масива може да се извърши по два начина:

- Инициализация, чрез присвояване на начални стойности на всеки елемент на масива поотделно:

```
ArrInteger[0]=10;  
ArrInteger[1]=15;  
ArrInteger[2]=20;  
ArrInteger[3]=21;  
ArrInteger[4]=13;
```

- Инициализиране при деклариране на масива, без да се използва оператора new:

```
int ArrInteger [] = {10, 15, 20, 21, 13};  
boolean array2 [] = {true, false, true};
```

в този случай компилаторът сам определя брой на елементите (размера на масива), заделя необходимата памет и я инициализира със зададените стойности.

Елементите на масива може да са от всеки тип, позволен за езика. Те може дори да са също от тип масив.

Масивът поддържа свойство **length**, което връща дължината му и е удобно за определяне на горната граница при индексирание. Долната е винаги 0.

Пр. 4

```
for (int I=0; I<ArrInteger.length;I++)  
    ArrInteger[I]=0; // нулиране на масива от горния пример
```

Многомерни масиви

Двумерният масив в езика Java се декларира с използване на две размерности. **По същество той се явява едномерен масив от скрити указатели към едномерни масиви.**

Пр. 5

```
int [][] matrix = new int [10][100];  
//матрица с размерност 10x100
```

Съответно достъпът до елемент от двумерен масив се извършва чрез посочване на два индекса:

Пр. 6

```
matrix [2][35];
```

Инициализиране на многомерни масиви:

- a) `int myarrayX [][] = {{5, 6, 3},
 {7, 2, 1},
 {4, 0, 9}};`
- б) `int myarrayY [][] = {{1},
 {2, 3},
 {4, 5, 6},
 {7, 8, 9}};`

При обработка на масиви, когато се налага обхождане на всички техни елементи, обикновено се използва операторът за цикъл **for**.

Пр. 7

- **Извеждане на едномерен масив на екран**

```
int myarray [] = {1, 2, 3, 4, 5};
for (int I=0; I<myarray.length; I++)
{
    System.out.println(myarray[I]);
}
```

- **Извеждане на многомерен масив на екран**

```
int myarray2 [][]={{1, 2, 3},
                   {4, 5, 6}};
for (int I=0; I<myarray2.length;I++)
{
    for (int K=0; K<myarray2.length; K++)
    {
        System.out.println(myarray[I][K]);
    }
}
```

Примерна програма

Пр. 8 Съставете програма, която по дадени стойности x_1, x_2, \dots, x_{12} , отговарящи на месечния оборот (в левове) на фирма за всеки от дванадесетте месеца на изминалата година, определя средномесечната стойност на оборота.

Проектиране: Програмата въвежда и обработва еднотипно 12 стойности от реален тип, които е най-добре да се представят като реален масив (например с име `arr`) с 12 елемента. Средномесечната стойност на оборота се определя по формулата за намиране на средно аритметично:

$$S = \frac{x_1 + x_2 + \dots + x_{12}}{12}$$

Изчисляването по формулата може да се програмира с помощта на оператор за цикъл с параметър, защото предварително е известно, че трябва да се съберат 12 числа.

```
public class oborot {
    public static void main(String[] args) {
        float[] oborot=
{1200,2345,345,1222,900,4567,7890,6000,8787,9876,8909,9999};
        float S=0;
        for (int i=0; i<oborot.length; i++)
            S+=oborot[i];
        S/=oborot.length;
        System.out.println("Средната стойност на оборота е:"+S);
    }
}
```

ТЕМА 14

ИЗКЛЮЧИТЕЛНА СИТУАЦИЯ

При изпълнението на програма може да възникнат грешки (при излизане извън обема на масив; при опит на променлива от даден тип да бъде присвоена стойност от друг тип и т.н.). Когато възникне такава грешка (или има вероятност да възникне), казваме че е налице т.нар. **изключителна ситуация**. Компиляторът следи за обработката на изключителните ситуации и програмистът е длъжен да осигури тяхната обработка (т.е. как да постъпи компютъра, ако възникне съответната изключителна ситуация). По този начин обработката на изключителни ситуации в Java е част от създаването на програмата.

Съществуват два типа изключителни ситуации – **системни и определени от програмиста**. При работа с изключителни ситуации се използват следните понятия – **генериране на ситуация, прихващане на ситуация и обработка на ситуация**.

Прихващане и обработка

Прихващането на ситуация и нейната обработка се извършва с помощта на конструкцията – **try-catch-finally**, включваща три програмни блока. Последният блок **finally** се използва рядко. Типичната конструкция се състои от първите два блока и има следният вид:

```
try {  
    //програмен код, в който настъпва изключителна ситуация  
} catch(<изключителна ситуация><обект>) {//прихващане на  
    //ситуация  
  
    //програмен код за обработка на изключителна ситуация  
}
```

Възможно е да съществуват няколко блока **catch**, всеки за прихващане и обработка на различна ситуация. Тези блокове прихващат обект от клас *<изключителна ситуация>*, генериран от блока **try**. Класът *<изключителна ситуация>* наследява класа *Exception* и може да съдържа произволни данни и методи, необходими и полезни за обработката на изключителната ситуация.

Пр. 1

Опитайте се да компилирате следната проста програма:

```
import java.io.*;  
class ExceptionTest {  
    public static void main(String[] args) {  
        System.in.read();  
    }  
}
```

В резултат на компиляцията ще бъде открита грешка, че изключителната ситуация "IOException" не е прихваната:

"ExceptionTest.java": Error #: 360 : unreported exception: java.io.IOException; must be caught or declared to be thrown at line 17, column 17

Възможни са две решения. Първото е програмният код да прихване и обработи изключителната ситуация (с конструкцията try-catch). Второто е методът main да се освободи от отговорността за нея, като я прехвърли на извикващия го. За тази цел в заглавието на метода се поставя служебната дума **throws**, следвана от името на изключителната ситуация.

Първо решение:

```
import java.io.*;
class ExceptionTest {
    public static void main(String[] args) {
        try {
            System.in.read();
        } catch (IOException e) {
            // код за обработка
        }
    }
}
```

Второ решение:

```
import java.io.*;
class ExceptionTest {
    public static void main(String[] args) throws IOException {
        System.in.read();
    }
}
```

Блокът **catch** е организиран като функция, получаваща обект от тип изключителна ситуация. Областта на видимост на обекта е

само в блока **catch**. Блокът **catch** се изпълнява само, ако настъпи съответната ситуация, т.е. в кода на **try** се генерира обект от тип, съвпадащ с типа на параметъра в **catch**. В противен случай (при нормална ситуация) след последния оператор в блока **try** се изпълнява първият оператор след конструкция **try-catch**.

В случай на един оператор е допустимо изпускане на фигурните скоби.

Пр. 2

```
try System.in.read();  
catch (IOException e); //празен оператор в частта catch
```

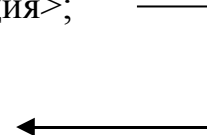
Името на параметъра в блока catch не може да съвпада с никое от имената на локалните променливи!

Генериране на изключителна ситуация

Генерирането на системна изключителна ситуация се извършва от системата, а на потребителска – с оператора **throw**. Този оператор се поставя в програмния код на блока **try**, на мястото, където не може да продължи нормалното изпълнение на програмата. При използването му с оператора **new** се генерира обект от тип <изключителна ситуация>.

Пр. 3

```
try {  
    //действия  
    if (<условие за ненормална ситуация>)  
        throw new <тип-изключителна-ситуация>;  
    //нормални действия  
} catch (<тип-изключителна-ситуация> e) {  
    //обработка на изключителна ситуация  
}  
...  
class <тип-изключителна-ситуация> extends Throwable {  
    ...
```



```
}
```

В класа изключителна ситуация може да се дефинират конструктори и при настъпване на ситуация да се предават параметри на обекта.

Задача 1 Да се организира клас, представляващ масив от цели числа, така че да се обработват следните изключителни ситуации: грешен размер на масива при неговата дефиниция и индексирание извън границите на масива.

Проектиране:

За да се обработват тези две ситуации, необходимо е за всяка от тях да се дефинира клас, разширяващ класа Exception. При втората ситуация е добре да се предава и индексът, при който тя е настъпила. За тази цел трябва да се въведат два класа, единият от които е тривиален.

Програмиране:

```
class Size extends Exception {}
class Range extends exception {
    private int index;
    public Range( int anIndex ) {index = anIndex;}
    public int getIndex() { return index; }
}
//основния клас е:
class ArrayOfInteger {
    private int [] theArray;
    private int theSize;
    public ArrayOfInteger( int size ) throws Size {
        if ( size <= 0)
            throw new Size();
        theSize = size;
    }
}
```

```

        theArray = new int [size];
        for( int index = 0; index < size; index++)
            theArray[index]=index;
    }
    public int get( int index ) throws Range {
        if ( index <= 0 || index > theSize)
            throw new Range ( index );
        return (theArray [index-1]);
    }
}
class Example {
    // example обработва Range, но прехвърля Size
    void example(int size) throws Size {
        try {
            ArrayOfInteger array = new ArrayOfInteger(size);
            int x = array.get(0); //get(1)
            System.out.println("Okey");
        } catch(Range r) {
            System.out.println("Index out of range:" +
r.getIndex());
        }
    }
    public static void main(String [] args) {
        try {
            Example e = new Example();
            e.example(-1); // e.example(10)
        } catch(Size s) {
            System.out.println("Invalid size.");
        }
    }
}

```

Разучете програмата. Забележете, че методът example на класа Example обработва ситуацията Range, но се освобождава и

прехвърля на извикващия го метод `main` ситуацията `Size`. Методите `ArrayOfInteger` и `get`, в които настъпват ситуациите (генерират се обектите) се освобождават от тях. В случай, че не се зададе ключовата дума `throws` след параметрите, компилаторът ще сигнализира за грешка.

Прехвърлянето на обработката на една изключителна ситуация на извикващия метод от извиквания става с поставяне на името ѝ в заглавния ред на метод:

```
<тип> <име-на-метод>([<параметър>,...]) throws <изключ.
ситуация>,... {
    ... throw new <>;...
}
```

Въведете и стартирайте горната програма. След това променете и пак стартирайте програмата последователно, както следва: първо, обръщението `e.example(-1)` на `e.example(10)` в метода `main` и второ `int x = array.get(0)` в метода `example` на `int x = array.get(1)`.

Ако е необходимо двете ситуации да се обработват в `example`, програмният код на класа `Example` ще изглежда така:

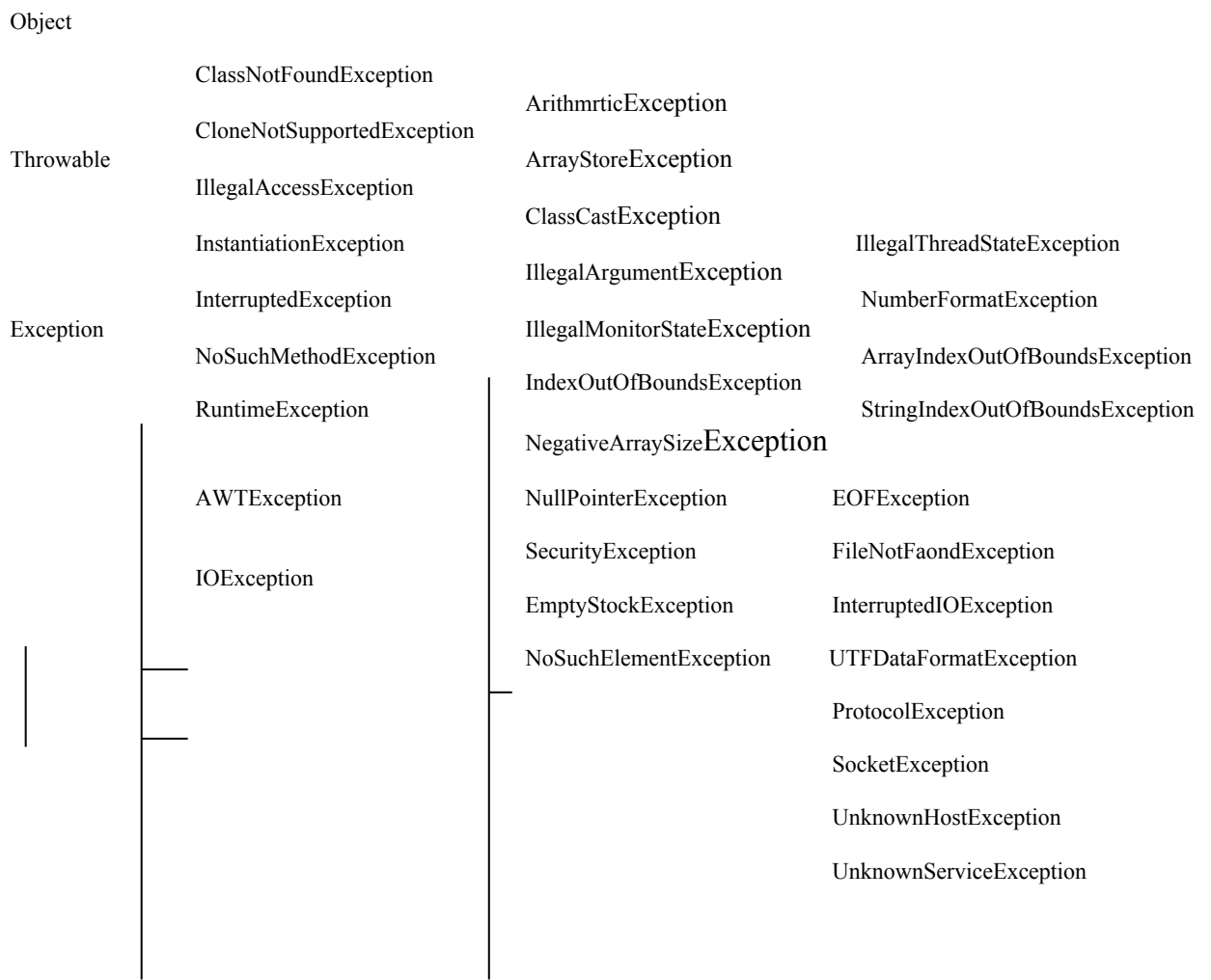
```
class Example {
    //example Range Size
    void example(int size) {
        try {
            ArrayOfInteger array = new ArrayOfInteger(size);
            int x = array.get(0); // get(1);
            System.out.println("Okey");
        } catch( Range r ) {
            System.out.println("Index out of range:" +
r.getIndex());
        } catch( Size s ) {
            System.out.println("Invalid size.");
        }
    }
}
```

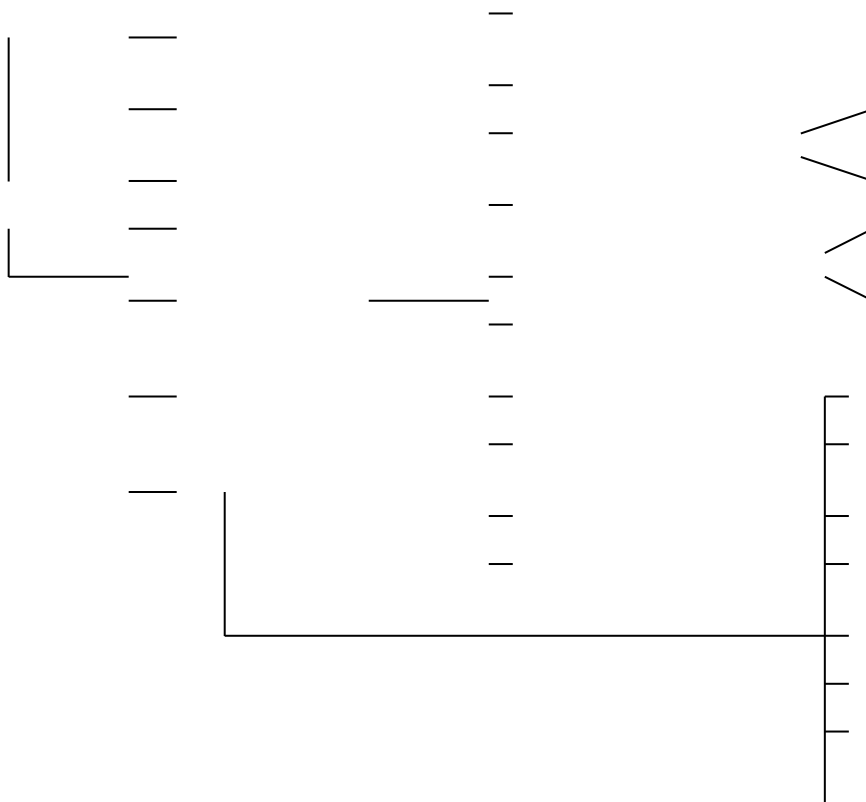
```

    }
    public static void main( String [] args) {
        Example e = new Example();
        e.example(-1); //e.example(10);
    }
}

```

На метода main не се прехвърля изключителна ситуация и поради тази причина в него няма конструкция try-catch. В метода example има два блока catch.





- Изключение се нарича неочаквано събитие (грешка) в програмата
- В рамките на програмата вие определяте изключенията като класове
- За да започне търсене на изключения се използва оператора **try**
- За да прихване конкретно изключение, програмата използва оператора **catch**
- За да предизвика изключение в момента на възникване на грешка, програмата използва оператора **throw**
- Когато програмата прихване изключение, тя го **обработва**

ТЕМА 15
РАБОТА С ТЕКСТОВЕ

Компютрите не биха имали толкова широко приложение, ако с тяхна помощ се обработваха само числови данни. В ежедневните си дейности човек най-често си служи с текстове. Компютрите са незаменими помощници в този вид дейност, особено когато се налага обработка на големи по обем текстове. Например при търсене на литературни източници в електронни библиотечни каталози или търсене на думи в електронни речници, при превод на текстове (романи, научни издания, новини) от един естествен език на други др.

Тип знаков низ

Низът е подредена съвкупност от краен брой знакове.

Броят на елементите на даден низ се нарича дължина на низа. Важно свойство на низовете е, че броят на знаковете, т.е. дължината на низа, може да се променя по време на обработката на структурата в програмата.

Основните операции за работа със знакови низове са конкатенация (слепване), сравнение, намиране на подниз, вмъкване на низ в друг, изтриване на подниз, намиране на дължината на низ, преобразуване на елементите (знаците) на низа и анализ на знаците.

а) **конкатенация** – операция между два знакови низа (отбелязва се с +), в резултат на която се получава нов низ, съдържащ елементите на първия, следвани от елементите на втория.

б) **сравнение** – сравнението се извършва на лексикографичен принцип. То се извършва знак по знак, като се започне от първите. Ако съответните знаци не са равни, за по-малък се взема този с по-малък код. В случай на съвпадение, се преминава на следващия. Ако при това елементите на един низ се изчерпят, той е подниз на другия и е по-малък.

в) операции за **промяна** на съдържанието на низ – съдържанието на низ може да се промени с вмъкване на низ или изтриване на подниз.

Особености на представянето на знаковите низове в Java

Знаковите низове в Java се представят с обекти. Съществуват два вида знакови низа – класовете String и StringBuffer. Първият се използва за константни знакови низове и веднъж създадени, те не могат да се променят. Вторият осигурява възможност за работа със знакови низове, променящи стойността и дължината си по време на изпълнение на програмата.

Всички обекти в Java трябва да се създават явно с помощта на оператора new. За създаване на два типа обекти, обаче има особен синтаксис. Единият бе разгледан в тема 13 – това е масивът. Другият е знаковият низ String. Когато в програма се срещне серия от знаци – текст, заградени в кавички, компилаторът създава автоматично обект от тип String и го инициализира с текста. Последното е еквивалентно на създаване на обект с new.

Пр. 1:

“Пловдив” // е еквивалентно на
`new String(“Пловдив”);`

Операторът “+” предизвиква конкатенация на знакови низове, когато такива се явяват левият и десният операнд. Ако единият операнд на оператора “+” е низ, то другият автоматично се преобразува в знаков низ и тогава се прави сливането. Вградените типове данни в Java автоматично се преобразуват в низове, докато за обектите това е невъзможно. Във всеки клас е реализиран методът toString(), който се използва при необходимост за преобразуване на обекта в знаков низ.

Знаков низ с фиксирана дължина – клас String

След като се създаде един обект от тип String той не може да се променя. Той представлява икономично представяне на знаков низ с фиксирана дължина.

Типът String е последен (final) и не може да се препокрива. Методите му позволяват обработка на отделни знаци, сравнение, търсене, извличане на подниз, създаване на копие и други операции, извършвани без модификация на оригинала.

Конструктори

Типът поддържа множество конструктори, позволяващи създаването на знаков низ от друг обект от тип String и StringBuffer или от масив от знаци.

Конструктор	Създава обект от:
<code>public String(String value)</code>	друг обект от клас
<code>public String(StringBuffer buffer)</code>	обект от клас
<code>public String(char[] value)</code>	масив от знаци
<code>public String(char[] value, int offset, int count)</code>	от област в масив от знаци

Общи методи

- `public String toString();` *//връща като резултат самия обект*
- `public char[] toCharArray();` *//връща масив от знаци, съответстващи на знаците от обекта*
- `public int length();` *//връща дължината на знаковия низ т.е. броя знаци*
- `public char charAt(int index);` *// връща знак, разположен на позиция index*

Методи за сравняване

- `public boolean equals(Object anObject);` *//връща true, ако обектът е знаков низ със същите знаци*
- `public boolean equalsIgnoreCase(String anString);` *//връща true, ако двата низа са еднакви, без отчитане на разликата в големи и малки букви*
- `public int compareTo(String anString);` *//лексико-графично сравняване на двата низа, като ако са еднакви връща 0, ако се различават връща разликата*

// между първите два различни знака

Методи за намиране на позиция на данни в знаков низ:

Следните методи връщат първата позиция на срещане на зададения знак или знаков низ, като се започне търсене от началото на низа. Ако не се срещне, методите връщат минус 1. когато се използва с допълнителен параметър fromIndex, търсенето започва не отначало, а от зададената позиция:

- public int **indexOf**(int ch); *//търси знак от началото*
- public int **indexOf**(int ch, int fromIndex); *// търси знак от
//зададена позиция*
- public int **indexOf**(String str); *//търси знаков низ от
//началото*
- public int **indexOf**(String str, int fromIndex);
//търси знаков низ от зададена позиция

Следните методи връщат първата позиция на срещане на зададения знак или знаков низ като се започне търсене от края към началото на низа. Ако не се срещне, методите връщат минус единица (-1). Когато се използва с допълнителен параметър fromIndex, търсенето започва не от края, а от зададената позиция:

- public int **lastIndexOf**(int ch); *// търси знак от края*
- public int **lastIndexOf**(int ch, int fromIndex); *// търси знак
//от зададена позиция*
- public int **lastIndexOf**(String str); *//търси знаков низ от
//края*
- public int **lastIndexOf**(String str, int fromIndex); *//търси
// знаков низ от зададена
позиция*

Методи за конструиране на нов обект от:

1. Обект

- public String **substring**(int beginIndex); *//създаване на
//обект, който е подниз*

- `public String substring(int bIndex,int eIndex);`
*//създаване на обект, подниз на обекта със зададени
//начална и крайна позиция*
- `public String concat(String str);` *//създаване на обект,
//който е конкатенация на два обекта*
- `public String toLowerCase();` *//създаване на нов
обект // от съответстващите малки букви на
обекта*
- `public String toUpperCase();` *//създаване на нов
обект // от съответстващите големи букви на
обекта*

2. Други данни

- `public static String valueOf(Object obj);`
//от произволен обект
- `public static String valueOf(char[] data);`
//от масив от знаци
- `public static String valueOf(char[] data, int offset, int
count);` *// от област в масив от знаци*
- `public static String valueOf(boolean b);`
//създава обект с "true" или "false"
- `public static String valueOf(char c);` *//знаков низ от
//един знак*
- `public static String valueOf(int i);` *//знаково(текстово)
// представяне на цяло число*
- `public static String valueOf(long l);` *//знаково //
(текстово) представяне на цяло число*
- `public static String valueOf(float f);` *//знаково
//(текстово) представяне на реално число*
- `public static String valueOf(double d);` *//знаково
// (текстово) представяне на реално число*

Извикване на метод

Знаковите низови константи се представят с обекти. Поради това е възможно директното прилагане върху тях на методи и

получаване на техния резултат. Обикновено това се използва за сравняване:

Пр. 2

```
public int compare(String x) {  
    if ("add".equals(x)) {  
        ...  
    else if ("mov".equals(x)) {  
        ...  
    }  
}
```

Изразът "add".equals(x) е еквивалентен на x.equals("add").

Знаков низ с променлива дължина – клас StringBuffer

Обектите от тип StringBuffer са като обектите от тип String, но могат да бъдат изменяни. Промяната може да бъде както в съдържанието на низа, така и в неговата дължина. Този тип се нарича знаков низ с променлива дължина. Неговото представяне заема повече ресурси в сравнение с типа String.

Методите на тип StringBuffer позволяват добавяне и вмъкване на знаци в него, както и промяна на дължината на буфера. StringBuffer се характеризира с две величини – дължина на буфера и дължина на низа.

Конструктори

Съществуват три конструктора - за създаване на празен буфер, за създаване на буфер със зададена дължина и за създаване на буфер от знаков низ с фиксирана дължина:

- public **StringBuffer**(); *//създаване на буфер с празен
// низ и дължина 16*
- public **StringBuffer**(int length); *//създаване на буфер с
//празен низ и зададена дължина*
- public **StringBuffer**(String str); *//създаване на буфер,*

*//съдържащ зададения низ и дължина равна на
//дължината на низа плюс 16*

Пр. 3

```
StringBuffer strEmpty = new StringBuffer();  
//празен знаков низ (с нулева дължина)  
StringBuffer strSize = new StringBuffer(7);  
//знаков низ с дължина 7  
StringBuffer strString = new StringBuffer("Пловдив");  
//низ Пловдив
```

Метод за генериране на знаков низ с фиксирана дължина

public String toString();

Създава нов обект знаков низ с фиксирана дължина и последващите изменения на низа от буфера не променят създадения знаков низ.

Методи за работа с дължината на низа и големината на буфера

- **public int length();** *//връща дължината на низа*
- **public void setLength(int newLength);** *//променя
//дължината на низа с отсичане или добавяне на
// знака null '\u000'*
- **public int capacity();** *//връща големината на буфера*
- **public void ensureCapacity(int minimumCapacity);**
*// когато големината на буфера е по-малка от
//параметъра, се създава нов вътрешен буфер с
по-//голяма големина, равна на по-голямото число от //
minimumCapacity и удвоената стара големина плюс 2*

Методи за работа със знаци на низа

- `public char charAt(int index);` *//върща знака на
//зададената позиция*
- `public void setCharAt(int index, char ch);` *//зададеният
//знак ch се записва на зададената
позиция*

Метод за добавяне в края на низа

Параметърът на този метод се преобразува до знаков низ (подобно на `String.valueOf`) и знаците от този низ се добавят в края на низа от обекта `StringBuffer`. Всички методи връщат самия обект като резултат, което е удобно за създаване на съставни операции.

- `public StringBuffer append (Object obj);`
- `public StringBuffer append (String str);`
- `public StringBuffer append (char[] str);`
- `public StringBuffer append (char[] str, int offset, int len);`
- `public StringBuffer append (boolean b);`
- `public StringBuffer append (char c);`
- `public StringBuffer append (int i);`
- `public StringBuffer append (long l);`
- `public StringBuffer append (float f);`
- `public StringBuffer append (double d);`

Реализация на конкатенацията (добавянето, слепването)

Този тип се използва за реализация на конкатенацията при създаване на нов знаков низ.

Пр. 4

`String a = "Стара";`

`String b = "Загора";`

`//изразът`

`a+" "+b`

`//е еквивалентен на израза`

`new StringBuffer().append(a).append(" ").append(b).toString()`

Метод за вмъкване на определена позиция в низ

Параметърът на този метод се преобразува до знаков низ (подобно на `String.valueOf`) и знаците от този низ се вмъкват в зададената позиция в `StringBuffer`. Всички методи връщат самия обект като резултат, което е удобно за създаване на съставни операции.

- `public StringBuffer insert(int offset, Object obj);`
- `public StringBuffer insert(int offset, String str);`
- `public StringBuffer insert(int offset, char[] str);`
- `public StringBuffer insert(int offset, boolean b);`
- `public StringBuffer insert(int offset, char c);`
- `public StringBuffer insert(int offset, int i);`
- `public StringBuffer insert(int offset, long l);`
- `public StringBuffer insert(int offset, float f);`
- `public StringBuffer insert(int offset, double d);`

Пр. 5

```
StringBuffer sbExample;
```

```
...
```

```
// следните две операции са еквивалентни
```

```
sbExample.append(data)
```

```
sbExample.insert(sbExample.length(), data)
```

Метод за “огледално” обръщане на знаковия низ

```
public StringBuffer reverse();
```

Огледалният знаков низ на един низ съдържа знаците на оригинала в обратен ред. Например, оригинал “Пловдив” има огледален низ “видволП”.

Пр. 6

```
StringBuffer x = new StringBuffer(“СтараЗагора”);
```

```
StringBuffer y = new StringBuffer(“Пловдив”);
```

```
...
```

```
//вмъкване на знак в знаков низ
```

```

x.insert(5, ' '); //x е "Стара Загора"
//вмъкване на един знаков низ в друг
y.insert(4, " - "); //y е "Плов - див"
//добавяне към знаков низ
x.append(", Хасковска област"); //x е "Стара Загора,
                                //Хасковска област"

//промяна на дължината на буфера
y.setLength(20); // y има дължина 20

// дължина и капацитет
int k = y.length(); //обикновено буферът е с 16 байта по-дълъг
int m = y.capacity(); //от инициализиращият го знаков низ

```

Въвеждане на знаков низ от клавиатурата

В показания по-долу клас `readString` се извършва въвеждане на знаков низ от клавиатурата до натискане на клавиша `Enter`, при което знаците последователно се записват в обект от тип `StringBuffer`. След това той се преобразува до знаков низ от тип `String`.

```

import java.io.*;
class readString {
    public static String read() {
        StringBuffer sb = new StringBuffer(0);
        char c;
        while((c = (char) Sistem.in.read()) != 10)
            sb.append(c);
        return sb.toString();
    }
}

```

Изключителни ситуации

При използване на методите на обектите от тип знаков низ е възможно да възникнат изключителни ситуации. Необходимо е програмистите да осигурят обработката на тези ситуации:

1. `IndexOutOfBoundsException` – при опит да се прочете (запише) знак на позиция, която не съществува т.е. тази позиция е по-голяма от дължината на низа или е зададено отрицателно число.
2. `NegativeArraySizeException` – при опит да се създаде буфер с отрицателна големина.
3. `NullPointerException` – при опит да се промени или използва неинициализиран обект.

ТЕМА 16

РЕКУРСИВНИ АЛГОРИТМИ

Рекурсия

Навярно помните детската приказка: “Имало едно време един поп, той си имал кученце, попът умрял и на гроба му написали: имало едно време един поп...”. Така думите в приказката се повтарят отново и отново, и така детето може да я разказва с часове, до безкрайност. Приказката е пример за явлението рекурсия. В най-общ смисъл рекурсията е явление, при което едно и също нещо се повтаря, случва се отново.

В ежедневието също се наблюдават рекурсивни явления: например, ако предмет се намира между две огледала, поставени едно срещу друго, то и в двете огледала отражението (в случая рекурсивно изображение) се съдържа в себе си отново и отново.

Рекурсията се използва често и в математиката за дефиниране на математически понятия и функции.

Пр. 1 Рекурсия в математиката

1. Рекурсивната дефиниция на понятието естествено число се състои от три части:

- а) 1 е естествено число;
- б) ако N е естествено число, то и $N+1$ е естествено число;
- в) няма други естествени числа, освен определените с а) и б).

Според б) следва, че $2=1+1$ е естествено число; аналогично и $3=2+1$ е естествено число и т.н

2. Стойността на функцията “факториел” за аргумент – естественото число N се определя с равенството $N!=1.2.3...N$. Нейната рекурсивна дефиниция е:

- а) $1!=1$;
- б) $N!=N.(N-1)!$, за всяко $N>1$.

Рекурсивни методи

В информатиката рекурсията се използва при описание и обработка на данни. Обикновено рекурсивните данни се описват с помощта на обекти от същия тип, но отличаващи се с по-проста структура или съставени от по-малък брой елементи.

В този урок ще разгледаме как могат да се съставят рекурсивни алгоритми и ще изучим средствата, предлагани от ЕП, за тяхната реализация. За програмирането на рекурсивни алгоритми се използват така наречените рекурсивни методи.

Рекурсивният метод е метод, който при своето изпълнение директно или косвено активира (извиква) себе си.

Активирането е директно (когато в метода има оператор за активиране на същия метод) или **косвено** (когато метода се активира при изпълнение на метод, включен в тялото на първия). В последния случай двата метода се наричат **взаимно рекурсивни методи**.

Съставяне на рекурсивни алгоритми и методи

Да се състави рекурсивна подпрограма (метод) не е трудно, ако съответните обекти или действия предварително са дефинирани рекурсивно, както е в случая например с функцията “факториел” от Пр. 1.2.

Пр. 3 Програма за намиране на $N!$ по дадено естествено число $N \geq 1$ с използване на рекурсивна подпрограма-функция:

```
import java.io.*;
class DemoRecursion {

    public int factorial(int value) {
        if (value == 1)
            return(1);
        else
```

```

        return(value*factorial(value-1));
        //рекурсивно извикване
    }
    public static void main(String [] args){
        N = 21;
        System.out.println(N+"!="+factorial(N));
    }
}

```

Основен **проблем** при съставянето на рекурсивни алгоритми е да се осигури тяхната крайност, т.е. съответната подпрограма да се изпълнява краен брой пъти.

Да разгледаме отново задачата за намиране на $N!$: съответната рекурсивна дефиниция зависи от един параметър – естествено число N ; точка б) от дефиницията в пример 1.2 свежда решаването на задачата до по-проста подзадача, като $N!$ се пресмята чрез $(N-1)!$; наличието на точка а) от дефиницията осигурява алгоритъмът да е краен, тъй като при $N=1$ стойността на $N!$ се намира непосредствено – без рекурсия.

По аналогия ще формулираме три **обща правила**, които трябва да се спазват при съставянето на рекурсивен алгоритъм за решаване на дадена задача:

- Определят се параметрите, които определят решението на задачата (**параметризация**);
- Намира се рекурсивна връзка между параметрите на дадената задача и тези на подзадача(и) от същия вид (**редукция**);
- Определя се подзадача, чието решение не е рекурсивно, т.е. води до завършване на алгоритъма, като се определят стойностите на параметрите, от които тази подзадача зависи (**условие за край**).

Основната трудност при съставяне на рекурсивни алгоритми се заключава в параметризацията. Намирането на параметри, подходящи за редукция, е творческа задача. Обикновено определянето на условието за край не представлява трудност.

Пр. 4 Алгоритъм и програма, която отпечатва двоичното представяне на дадено естествено число N , записано в десетична бройна система.

Проектиране

Начинът за преобразуване на числа от десетична бройна система в двоична вече ви е известен: даденото десетично число N да се дели на две, а след това и всяко получено частно, докато не се получи частно 0; редицата от остатъци на извършените деления, записана в обратен ред, е търсеното двоично представяне на N . По този начин например е намерено двоичното представяне 110 на десетичното число 6 (фиг. 1).

$6 : 2 = 3$		0	
$3 : 2 = 1$		1	$6_{(10)} = 110_{(2)}$
$1 : 2 = 0$		1	

При съставяне на рекурсивния алгоритъм ще спазваме общите правила, формулирани по-горе:

1. Очевидно решението на задачата зависи от един параметър – естествено число N .
2. Ако се знае двоичното представяне на частното $\text{Math.floor}(N/2)$ и ако то вече е изведено на дисплея, за да се получи двоичното представяне на N , остава да се изведе и остатъкът $(N \% 2)$.
3. Целочислените деления приключват, когато се получи частно 0 (т.е. рекурсията завършва при стойност на параметъра $N = 0$).

- В подпрограмата отпечатването на поредния остатък се извършва след рекурсивното извикване, което дава възможност остатъците да се извеждат на дисплея в обратен ред и да формират двоичното представяне.
- Използването на рекурсивни процедури и функции е удобно, защото съставянето им обикновено следва директно от условието на самата задача. Реализацията без рекурсия обаче обикновено е по-икономична по отношение на необходимите за изпълнение време и памет. Илюстрация на това е следващият пример, в който се изчислява $N!$, като се използва нерекурсивна формула за намирането му ($N! = 1*2*3*...*N$).

Пр. 5 Нерекурсивна функция за пресмятане на $N!$

```
public int factorial (int value) {
    int F=1;
    for (int I=2; I <= value; I++) F = F*I;
    factorial = F;
}
```

ТЕМА 17

ОБРАБОТКА И СЪХРАНЕНИЕ НА ДАННИ

Поток

Потокът е абстрактен източник или получател на данни, осигуряващи на програмиста удобен интерфейс под формата на функции и подпрограми за въвеждане или извеждане на информация. Потоците са логически устройства и са независими от конкретната файлова и входно-изходна системи.

Потокът данни е последователност от байтове, която може да бъде свързана с физически файл, с временен файл, с област от паметта на програмата, обикновена поредица байтове или с физическо устройство, като клавиатура, принтер и дисплей.

Временните файлове са буфери в операционната система или в програмата. В тях една програма записва, а друга чете. Входен и изходен поток могат да бъдат организирани с последователност от байтове в оперативната памет. **Въвеждането на информация от входен поток в този случай е свързано с четене от паметта, а извеждането на изходен поток – със запис в паметта.**

Потоци InputStream и OutputStream

Библиотеката за вход/изход (io) на Java предоставя множество класове за работа с потоци. Всички потоци, работещи с входа, се явяват подкласове на абстрактния клас `InputStream`, а всички, работещи с изхода – подкласове на абстрактния клас `OutputStream`. Единственото изключение е класът **`RandomAccessFile`**. В обект от типа `RandomAccessFile` може както да се пише, така и да се чете т.е. осъществява се и вход и изход.

`InputStream` и `OutputStream` осигуряват минималния интерфейс и вътрешната реализация на потока. Те поддържат методи за четене и запис на байт или последователност от байтове. Класът за четене поддържа още методи за прескачане на определен брой байтове, установяване на указателя на текуща позиция в началото на файла и др. Потоците се отварят автоматично при създаването на обект и се затварят явно с метода **`close()`** или неявно при освобождаване на заетата памет.

Методи на класа InputStream

В класа `InputStream` са дефинирани няколко метода, осигуряващи четене от поток данни. Тези методи са предефинирани в наследяващите класове.

1. Методът **`read()`** има три версии:

- Първата е без параметри – извършва се четене на един байт и се връща цяло число в диапазона от 0 до 255. Ако в потока няма данни, се връща минус 1.
`read ();`
- Втората е с един параметър - масив от байтове. В този случай се въвеждат определен брой байтове, който не може да е по-голям от големината на масива

(параметъра). Методът връща броя на въведените байтове или -1, ако не е въведен нито един байт.

read (byte [] <масив>);

- Третата версия на метода има три параметъра. Първият е масив, в който ще се въвежда информацията, вторият е позицията в масива, от където ще започне да се въвежда, а третият е броят на символите, които трябва да се прочетат от потока.

read (byte [] <масив>, int<позиция>, int<брой>);

Пр.1

// четене на знаков низ от поток, записан в Паскал-стил

```
public void readPascalString(InputStream isExample, byte[] bString) {  
    int iNumber = isExample.read(); // вариант 1  
    int iRealBytes = isExample.read(bString, 0, iNumber); // вариант 2  
    if (iNumber != iReadBytes) {  
        // генериране на изключителна ситуация  
        // неправилен формат на входните данни  
    }  
}
```

2. Методът **skip(<брой>)** пропуска зададен брой байтове от входния поток.

3. Методът **available()** връща броя на байтовете които могат да се прочетат от потока. Ако потока е закрит, връща -1.

Методи на класа OutputStream

В класа OutputStream са дефинирани няколко метода, осигуряващи запис на данни в поток. Тези методи са предефинирани в наследяващите класове.

1. Основен метод е **write()**. Той има един параметър във формата на цяло число. Като резултат от изпълнение на метода този байт се добавя към потока. Както и при read() се поддържат и останалите две версии – с масив и с масив и диапазон (начало и брой байтове). Техният синтаксис е както следва:

- write(int <байт>);
- write(byte[] <масив>);
- write(byte[] <масив>, int <позиция>, int <брой>);

2. Методът **flush()** се използва за принудително записване на данните на носителя. Този метод е от особено значение за буферираните потоци.

Извлечени методи

От класовете `InputStream` и `OutputStream` са извлечени няколко класа за въвеждане и извеждане на байтове. Те са описани в следната таблица:

Потоци	Описание
<code>FileInputStream</code> <code>FileOutputStream</code>	За въвеждане от и извеждане на двоичен файл
<code>ByteArrayInputStream</code> <code>ByteArrayOutputStream</code>	За въвеждане от и извеждане в/от масив в оперативната памет
<code>PipedInputStream</code> <code>PipedOutputStream</code>	За организиране на временни файлове, които се свързват по двойки за формиране на канал между секции в една програма
<code>SockedInputStream</code> <code>SockedOutputStream</code>	За организиране на временни файлове, които се свързват по двойки за формиране на канал между програми в мрежа
<code>SequenceInputStream</code>	За обединение на няколко входни потока в един
<code>StringBufferInputStream</code>	За въвеждане от <code>StringBuffer</code>

Филтриращи потоци

Филтриращите потоци са група от класове, които наследяват абстрактните класове `FileInputStream` и `FileOutputStream`. Последните път от своя страна наследяват съответно `InputStream` и `OutputStream`. Те дефинират интерфейс към филтри, които се използват автоматично за обработка на данни при тяхното въвеждане и извеждане. Отварянето на филтриращите потоци се извършва при създаване на обект от този клас. На конструкция на обекта се задава съответно обект от тип `InputStream` или `OutputStream`. Към филтриращите класове спадат следните потоци:

Потоци	Описание
--------	----------

DataInputStream DataOutputStream	Съдържат съответно филтри за въвеждане и извеждане на примитивни типове в машинно-независим формат
BufferedInputStream BufferedOutputStream	Буферират данните при въвеждане и извеждане с цел намаляване брой на обръщенията към носителя
PushbackInputStream	Входен поток с възможност за връщане на един байт
LineNumberInputStream	Поддържа бройч на редовете по време на четенето

При използване на FileInputStream и FileOutputStream се прави **точно двоично копие на данните от паметта**. При DataInputStream и DataOutputStream се прави **двоично копие в машинно-независим формат**.

В класовете DataInputStream и DataOutputStream са реализирани различни методи, позволяващи въвеждане и извеждане на примитивни типове. Тези класове са полезни за директно извеждане и въвеждане на данни. Методите и съответните данни са следните:

- readBoolean/ writeBoolean – прехвърля се един байт, винаги 0 – за неистина, а за истина се счита всичко останало при четене и 1 при запис;
- readByte/ writeByte – прехвърля се един байт, без да се преобразува към цял тип;
- readChar/ writeChar – прехвърля се един символ (2 байта), първо се прехвърля старшият байт;
- readInt/ writeInt – прехвърля се 32-битово знаково число;
- readShort/ writeShort – прехвърля се 16-битово число със знак, но при четене то се преобразува до 32-битово;
- readLong/ writeLong – прехвърля се 64-битово число със знак;
- readFloat/ writeFloat – прехвърля се 32-битово реално число;
- readDouble/ writeDouble – прехвърля се 64-битово реално число.

```

FileOutputStream s = new FileOutputStream("test.dat");
DataOutputStream f = new DataOutputStream (s); //обект от
                                                // mun OutputStream

f.writeInt(12);
f.writeLong(50091909);
f.writeDouble(3.14159);
f.writeBytes("Plovdiv");

```

Буферирано въвеждане и извеждане

Буферираното въвеждане и извеждане позволява намаляване на броя на операциите по записване и четене на операционната система. За целта се използва вътрешен буфер. Записването се извършва, когато се изпълнят методите `flush()` и `close()` или се запълни вътрешният буфер. Четенето се извършва при отваряне на потока или изчерпване на съдържанието на вътрешния буфер.

Пр. 3

```

FileOutputStream s = new FileOutputStream("test.dat");
BufferedOutputStream b = new BufferedOutputStream(s);
... b.write(...); ... // буферирано извеждане
b.close();
s.close();

```

Обработка на файлове с директен достъп

В езика Java се поддържа клас **RandomAccessFile**, който включва интерфейсите за вход и изход. При създаване на обект от този клас може да се зададе и начинът на използването му – само за четене или за четене и запис.

С методите от вида **read<тип>()** и **write<тип>()** може да се извършват операции за четене и запис на примитивни данни. С помощта на наследяване, препокриване и дефиниране на нови методи от вида **read<тип>()** и **write<тип>()** могат да се дефинират филтри за достъп.

Класът поддържа методи за директна работа с указателя на текущата позиция. Това са **skipByte()** – премества указателя с указан брой байтове напред, **seek()** - позиционира указателя на зададена позиция и **getFilePointer()** – връща текущата позиция на указателя. При отваряне на файла указателят на текущата позиция се позиционира в началото и при четене или запис се премества с дължината на прочетените или записани данни. С помощта на метода **seek()** тази позиция може да бъде променяна. Със **seek(0)**

указателят се позиционира в началото на файла, а с **<обект>.seek(<обект>.length())** – в края на файла (ако е необходимо да се добавя във файла).

Пр.4

```
import java.io.*;
class RandAccessTest {
    public static void main(String[] args) throws IOException{
        RandomAccessFile f=new RandomAccessFile("test.dat",
"rw");
        f.writeInt(12);
        f.writeLong(50091909);
        f.writeDouble(3.14159);
        f.writeBytes("Plovdiv");
        f.seek(0);
        int I = f.readInt();
        long l = f.readLong();
        double d = f.readDouble();
        String s = f.readLine();
        f.close();
        System.out.println("From file " +I+", "+l+", "+d+", "+s);
        System.in.read();
    }
}
```

В този пример се използва файл с име test.dat. Той се обработва с помощта на обект от тип RandomAccessFile. При създаване на този обект се задава името на файла и режимът на отваряне. Този режим може да бъде "r" - само за четене или "rw" – за четене и запис.

В примерната програма се записват четири типа данни, след което файлът се позиционира в началото и се прочита записаната информация. Накрая тази информация се отпечатва.

Клас File

За да се получи информация за файл или директория може да се използва класът **File**. В конструктора се задава името на файла при създаване на обект от този клас. С метода **exists()** може да се провери дали файл или директория с такова име съществува, а с **isDirectory()** и с **isFile()** може да се провери дали това е директория или файл. Методите **getName()** и **getPath()** връщат

името и пътя на файла, а неговата големина може да се получи с **length()**. Всички атрибути на файла или директорията могат да бъдат проверени или получени с множество от методи като **canRead()**, **canWrite()**, **lastModified()** и др.

Файлът може да бъде изтрит или преименуван с **delete()** и **renameTo()**.

Пр. 5

```
File f = new File("test.dat");
if (f.exists()) {
    System.out.println("File is "+f.length()+"bytes long.");
    if (f.isDirectory()) {
        ...
    }
}
```

1. Проектирайте клас с два метода – **reading** и **writing**. Първият метод да отваря един файл за четене и запис, да записва в него данни за един студент – име, номер и група и да го затваря. Вторият метод да отваря същия файл само за четене, да прочита информацията от него и да я извежда на екран. За тестване на проектирания клас създайте нов клас със статичен метод **main**, който да използва двата метода.

Решение:

```
import java.io.*;
class StudentFile {
    String fname;
    void reading() {
        RandomAccessFile f=null;
        try {
            f = new RandomAccessFile (fname, "rw");
        } catch (FileNotFoundException e) { System.out.print("Errorr: "+e);}
        try {
            f.writeUTF("Ivan Ivanov");
            f.writeInt(2018312);
            f.writeShort(3);
        } catch(IOException e) { System.out.print("Errorr: "+ e); }
    }
    void writing() {
        RandomAccessFile f=null;
        String s = null;
        int I =0;
        short k = 0;
        try {
            f = new RandomAccessFile (fname, "r");
            s = f.readUTF();
            I = f.readInt();
            k= f.readShort();
            f.close();
            f.seek(0);
        } catch (IOException e) { System.out.print("Errorr: "+e);}
        System.out.println("From file > name: "+s);
        System.out.println("number: " +I);
        System.out.println("groupe: " +k);
    }
    public StudentFile(String fname) {
        this.fname=fname;
    }
    public StudentFile() {
        //this.fname="Stu.dat";
    }
}
public class TestStudentFile extends StudentFile {
    static void main(String[] args) {
        StudentFile SF = new StudentFile("Stu.dat");
        SF.reading ();
        SF.writing ();
    }
}
```

ПРИЛОЖЕНИЕ 1

ДОПЪЛНИТЕЛНИ ЗАДАЧИ

ТЕМА 4:

1. Напишете програма за пресмятане корените на квадратното уравнение $x^2 - 5x + 6 = 0$. Кой е най-големият недостатък на програмата?

Решение:

```
import java.io.*;
import java.math.*;
class square_1 {
    public static void main(String[] args) {
        double x=0, y=0;
        x = 5*5 - 4*6;
        y = (5 + Math.sqrt(x))/2;
        System.out.println("Решенията на уравнението са:");
        System.out.println("x1=" + y);
        y = (5 - Math.sqrt(x))/2;
        System.out.println("x2=" + y);
    }
}
```

2. Опитайте се да определите какви видове оператори (с какво действие) са необходими, за да можем да напишем програмата от предходната задача така, че тя да изследва и в случай че е възможно, да намира корените на произволно квадратно уравнение.

ТЕМА 5:

1. Зависимостта между величините x и y е следната:

$$y = \begin{cases} x^2 - 3x + 5 & , \quad \text{ако } x \leq -4 \\ 2x + |x| & , \quad \text{ако } -4 \leq x \leq 5 \\ x - 10 & , \quad \text{ако } x > 5 \end{cases}$$

Да се състави програмата, която при въведена стойност на x намира y .

Решение:

```
import java.io.*;
import java.math.*;
class x_y {
    public static void main(String[] args) {
        double x=12.5, y=0;
        System.out.println("При въведена стойност за x:"+x);
        if (x <= -4) y = x*x - 3*x + 5;
        else if ((x >= -4) && (x <= 5)) y = 2*x + Math.abs(x);
        else y = x - 10;
        System.out.println ("y="+y);
    }
}
```

ТЕМА 6:

1. Напишете програма с използване на оператор, която:

- Определя дали въведеното цяло число е четно или не;

```
import java.io.*;
class even {
    public static void main(String [] args) {
        int x=6;
        if ((x%2)==0)
            System.out.print("Числото "+x+"е четно");
        else System.out.print("Числото "+x+"е нечетно");
    }
}
```

- След въвеждане на дневната температура (цяло число), отпечатва едно от съобщенията “студено”, “хладно”, “приятно” или “горещо” в зависимост от това дали температурата е съответно в интервала $[-3, 5]$, $[12, 18]$, $[19, 25]$ или $[26, 32]$;

```

import java.io.*;
class Temp {
    public static void main(String [] args) {
        int x=21;
        switch(x) {
            case -3: case -2: case -1: case 0: case 1: case 2: case 3: case 4: case 5:
                System.out.print("студено"); break;
            case 12: case 13: case 14: case 15: case 16: case 17: case 18:
                System.out.print("хладно"); break;
            case 19: case 20: case 21: case 22: case 23: case 24: case 25:
                System.out.print("приятно"); break;
            case 26: case 27: case 28: case 29: case 30: case 31: case 32:
                System.out.print("горещо"); break;
            default: System.out.print("не е в зададен интервал");
        }
    }
}

```

ТЕМА 7:

1. Съставете програма по алгоритъма за сумиране на редица от числа. Определете кои са основните елементи на цикъла, който участва в програмата, която изчислява и отпечатва произведението на първите 7 естествени числа (т.е. стойността на 7!, което се чете "седем факториел").

```

import java.io.*;
class sum_a {
    public static void main (String [] args) {
        int S = 0;
        int x = 0;
        do {
            x += 4;
            S += x;
            System.out.print(S+" ");
        } while (x < 26);
    }
}

```

```

import java.io.*;
class sev_fact {
    public static void main (String [] args) {
        int I=1;
        int P=1;
        while (I<=7) {
            P*=I;
            I++;
        }
        System.out.print("Произведението е: "+P);
    }
}

```

ТЕМА 8:

1. Напишете програма, която показва на дисплея таблицата за умножение на числата от 1 до 10.

```

import java.io.*;
class table {
    public static void main (String [] args) {
        for(int I=1; I<=10; I++)
            for(int j=1; j<=10; j++)
                System.out.println(I*j);
    }
}

```

2. Да се състави програма на Java, която определя броя на трицифрените естествени числа, сумата от цифрите на които е равна на дадено естествено число s (s е между 1 и 27).

Решение:

```
import java.io.*;
class Sum_c {
    public static void main (String [] args) {
        byte S=25;
        boolean B = ((S>=1)&&(S<=27));
        if (B)
            for (int I = 0;I <= 999;I++)
                for (int J = 0;J <= 999;J++)
                    for (int K = 0; K<=999;K++)
                        if ((I+J+K)== S)
                            System.out.println("Сумата от цифрите на
"+I+", "+J+", "+K+" е равна на "+S);
    }
}
```

ТЕМА 9:

1. Да се състави програма за решаване на линейни неравенства от вида: а) $ax > b$ и б) $ax < b$.

а)

```
import java.io.*;
class ex_a {
    public static void main(String[] args) {
        float a=12, b=14, x;
        System.out.println("Стойност на коефициент a:"+a);
        System.out.println("Стойност на коефициент b:"+b);
        if ((a==0)&&(b>=0)) System.out.println("Неравенството няма решение");
        if ((a==0)&&(b<0)) System.out.println("Всяко x е решение");
        if (a<0) {x=b/a; System.out.println("Решенията са в интервала (-∞;"+x+"");}
        if (a>0) {x=b/a; System.out.println("Решенията са в интервала (" +x+";+ ∞)");}
    }
}
```

б)

```
import java.io.*;
class ex_b {
    public static void main(String[] args) {
        float a=12, b=14, x;
        System.out.println("Стойност на коефициент a:"+a);
        System.out.println("Стойност на коефициент b:"+b);
        if ((a==0)&&(b<=0)) System.out.println("Неравенството няма решение");
        if ((a==0)&&(b>0)) System.out.println("Всяко x е решение");
        if (a>0) {x=b/a; System.out.println("Решенията са в интервала (-∞;"+x+"");}
        if (a<0) {x=b/a; System.out.println("Решенията са в интервала (" +x+";+ ∞)");}
    }
}
```

2. Да се състави програма за решаване на биквадратното уравнение $ax^4+bx^2+c=0$.

```
import java.io.*;
class bisquare{
    public static void main(String[] args) {
        double a=11, b=12 ,c=34, x1, x2, x3, x4, y1, y2, D;
        System.out.println("Стойност на коефициент a:"+a);
        System.out.println("Стойност на коефициент b:"+b);
        System.out.println("Стойност на коефициент c:"+c);
        D=b*b-4*a*c;
        if (D>0) {
            y1=(-b+Math.sqrt(D))/2*a;
            y2=(-b-Math.sqrt(D)) /2*a;
            if ((y1>=0)&& (y2>=0)){
                x1=Math.sqrt(y1); x2=-Math.sqrt(y1); x3= Math.sqrt(y2); x4=-Math.sqrt(y2);
                System.out.println("Корените на уравнението
ca:"+x1+", "+x2+", "+x3+", "+x4);
            }
            if ((y1>=0)&& (y2<0)){
                x1=Math.sqrt(y1); x2=-Math.sqrt(y1);
                System.out.println("Уравнението има два корена:"+x1+", "+x2);
            }
            if ((y1<0)&& (y2>=0)){
                x3= Math.sqrt(y2); x4=-Math.sqrt(y2);
                System.out.println("Уравнението има два корена:"+x3+", "+x4);
            }
            if ((y1<0)&& (y2<0)) System.out.println("Уравнението няма реални корени");
        }
        else if (D==0) {
            y1=-b/2*a;
            if (y1>=0){
                x1=Math.sqrt(y1); x2=-Math.sqrt(y1);
                System.out.println("Уравнението има корени:"+x1+", "+x2);
            }
            else System.out.println("Уравнението няма реални корени");
        }
        else System.out.println("Уравнението няма реални корени");
    }
}
```

ТЕМА 12:

1. Да се състави програма за изчисляване стойността на функцията:

$$f(x) = \begin{cases} z(x) + z^2(x+1), & \text{за } x < 0 \\ 6, & \text{за } x = 0 \\ z(x) + \frac{1}{z^2(x-1) - 1}, & \text{за } x > 0 \end{cases}$$

, където $z(x) = x^2 - 2x + 12$.

Упътване: в програмата декларирайте и използвайте подпрограма за изчисляване стойността на $z(x)$.

```
import java.io.*;
class func_Z {
    float func_z (float x) {
        return (x*x - 2*x + 12);
    }
    float func_f (float x) {
        if (x<0) return (func_z(x)+ func_z(x+1)* func_z(x+1));
        else if (x==0) return(6);
        return (func_z(x) + 1/ (func_z(x-1)*func_z(x-1) - 1));
    }
    static void main(String[] args) {
        float x=5;
        func_Z y = new func_Z();
        y.func_z(x);
        System.out.print(y.func_f(x));
    }
}
```

2. Да се състави подпрограма-функция, която връща резултат **true**, ако стойностите на реалните променливи **a**, **b** и **c** могат да бъдат дължини на страни на триъгълник и **false** – в противен случай.

```

boolean strTri (float a, float b, float c) {
    if (((a+b)>c)||((a+c)>b)||((b+c)>a)) return(true);
    else return(false);
}

```

ТЕМА 13:

1. Нека a_1, a_2, \dots, a_{20} и b_1, b_2, \dots, b_{20} са две редици от по 20 цели числа. Съставете програмата, която пресмята сумата:

$$S = a_1b_1 + a_2b_2 + \dots + a_{20}b_{20} .$$

```

public class red {
    public static void main(String[] args) {
        int red1[]={1,2,3,4,5,6,7,8,9,10};
        int red2[]={10,9,8,7,6,5,4,3,2,1};
        int s=0;
        for (int I=0; I<red1.length; I++)
            s+=red1[I]*red2[I];
        System.out.println("сумата е:"+s);
    }
}

```

2. Съставете програмата, която запълва масив от цели числа със стойностите от таблицата за умножение на числата от 1 до 10. Предварително определете какви типове индекси и каква размерност е удобно да има този масив.

```

public class tabl_umn {
    public static void main(String[] args) {
        int [][] arr1=new int[10][10];
        int arr2 [] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        int arr3 [] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        for (int I=0; I<arr2.length; I++)
            for (int J=0; J<arr3.length; J++) {
                arr1[I][J]=arr2[I]*arr3[J];
                System.out.println("arr1["+I+"]"+"["+J+"]="+arr1[I]
[J]);
            }
    }
}

```

РАЗШИРЕНИЕ КЪМ ТЕМА 13:

Алгоритми за търсене и броене

Решаването на много интересни практически задачи води до съставянето на програми, в които участват масиви. Често те се основават на сходни алгоритми за обработка на масиви, различаващи се само по типа или броя на своите елементи. Такива например са задачите за определяне на средния успех на учениците от един клас, намирането на ученика с най-висок среден успех или най-високия месечен оборот от фирма. Най-често срещаните алгоритми с масиви са добре изучени и разработени. Ще разгледаме основни алгоритми за търсене и броене в масив, както и примери на тяхната реализация.

Алгоритми за търсене в масив

Често срещана практическа задача е задачата за търсене в множество от определени данни. Може например да се търси фирмата, която предлага дадена стока на най-ниска цена, ученикът с най-висок успех в училище и т.н.

Задачата за търсене в общия случай изисква да се намери (отдели) един или няколко елемента в съвкупност от елементи така, че те да отговарят на предварително зададено условие, наречено **критерий на търсене**. В горните примери тези критерии съответно са “най-ниска цена” и “най-висок успех”.

Когато множествата от елементи, между които се търси, са еднотипни, при програмното решаване на задачата за търсене е удобно това множество да се представи като масив.

Задача 1 Задача за търсене в масив A , състоящ се от N елемента, на елемент X , отговарящ на даден критерий.

Алгоритъмът за последователно търсене в масив се основава на следната идея: последователно, от първия към последния елемент на масива, се обхождат всички елементи на масива A ; всеки елемент се проверява дали отговаря на поставения критерий

за търсене, като резултат елементът X може да бъде намерен или не. Този алгоритъм се нарича **обикновено последователно търсене**.

В пример 1 е представена програма, реализираща алгоритъма на последователно търсене, като търсенето спира при първото срещане на елемент със стойност value (т.е. търсим елемент с дадена стойност value).

Пр. 1 Обикновено последователно търсене

Проектиране

Нека приемем, че масивът A се състои от реални числа. Алгоритъмът за последователно търсене е цикличен и ще го представим с оператора за цикъл for.

```
import java.io.*;
class subseq_search {
    public static void main (String [] args) {
        float RealArr [] = {1,79,89,56,76}; //масив с 5 реални елемента
        float R=7;                          //търсеният елемент
        byte I=0;                            //индекс
        while ((I!=RealArr.length-1) && (RealArr[I]!=R))
            I++;
        if (RealArr[I]==R)
        {
            System.out.println("търсеният елемент е с индекс "+ I);
        }
        else System.out.println("елементът не е намерен!");
    }
}
```

Дихотомно (двоично) търсене

Дихотомното търсене е много удобно да се използва, когато масива е сортиран (подреден). То се основава на следната **идея**: масива се разделя на все по-малки части, докато се намери търсеният елемент. Като начало, масива се разделя на две и тогава се решава в коя от двете части да се търси елемента (имаме в

предвид, че масива е подреден). След това половината от масива отново се разделя на две и т.н. докато се открие търсения елемент.

Пр. 2 Програмна реализация на алгоритъм за дихотомно търсене

```
public class arrBinary {
    public static int bsearch (int array [], int value) {
        boolean found = false;
        int high = array.length - 1; // последният елемент на масива
        int low = 0;                  // първият елемент на масива
        int cnt = 0;                  // брояч
        int mid = (high + low)/2;     // за разделяне масива на все по-малки части
        System.out.println ("търсим елемент: " + value);
        while (!found && (high >= low)) {
            System.out.print("Low " + low + "Mid " + mid);
            System.out.println(" High " + high);
            if (value == array[mid]) {
                found = true;
            }
            else if (value < array[mid])
                high = mid - 1;
            else    low = mid + 1;
            mid = (high + low)/2;
            cnt++;
        }
        System.out.println("Steps " + cnt);
        return ((found) ? mid: -1);
    }
    public static void main (String [] args) {
        int array [ ] = new int [100];
        for (int I=0; I<array.length; I++) {
            array[I] = I;
            System.out.println("Result " + Lsearch(array, 67));
            System.out.println("Result " + Lsearch(array, 33));
            System.out.println("Result " + Lsearch(array, 1));
            System.out.println("Result " + Lsearch(array, 1001));
        }
    }
}
```

Алгоритъм за търсене на най-голяма стойност

Друга често срещана задача, при програмното решаване на която е удобно данните да се представят като масив, е задачата за търсене на най-големия по стойност елемент измежду множество

от елементи. Алгоритъмът се основава на следната **идея**: последователно се преглеждат елементите на масива, като най-голямата до момента стойност се помни (например в променлива MAX); всеки прегледан елемент се сравнява с MAX (рекорда до момента) и ако е по-голям, стойността на MAX се променя и става равна на стойността на този елемент. По този начин на всяка стъпка MAX ще съдържа най-голямата стойност, открита до тази стъпка. След като се обхождат всички елементи на масива MAX ще има стойността на най-големия елемент.

За начална стойност на MAX (инициализация) обикновено се избира първият елемент на масива, но би могло да се избере и стойност, за която е известно, че е по-малка от всички елементи от масива (защо?).

Подобен алгоритъм може да се приложи и ако се търси най-малкият елемент на масив. Разликата в този случай е, че на всяка стъпка се проверява дали поредният елемент е по-малък от намерената до момента рекордно малка стойност.

Програмата в пример 3 реализира разгледания алгоритъм за намиране на най-големия и най-малкия елемент в масив от реални числа.

Пр. 3 Програма, която при зададени стойности на месечния оборот на фирма за всеки от дванадесетте месеца на годината определя най-високия и най-ниския месечен оборот

```

import java.io.*;
class turnor {
    public static void main (String [] args) {
        float oborot [] =
{1111,2222,3333,222,33333,222222,3333,444,5555,666,7777,5434};
        float MIN, MAX;
        MAX = oborot[0];
        MIN = oborot[0];
        for (int I=1; I<oborot.length; I++)
        {
            if (oborot[I] > MAX) MAX = oborot[I];
            if (oborot[I] < MIN) MIN = oborot[I];
        }
        System.out.println("Максималният оборот е "+ MAX);
        System.out.println("Минималният оборот е "+ MIN);
    }
}

```

Алгоритми за броене

Много близки до задачите за търсене са задачите, които изискват преброяване на елементите на масив, имащи определени свойства или отговарящи на някакъв критерий. Например намиране броя на:

- положителните и отрицателните числа в масив от числа;
- елементите, които са равни на дадена стойност или пък са по-малки от нея;
- елементите, равни на максималния елемент на масив и т.н.

Алгоритъмът, по който се решават задачите за броене, е много близък до разгледаните по-горе. Стъпките на алгоритъма са следните: предварително на променлива от цял тип *Вг* се присвоява стойност 0 (в края тя ще съдържа броя на елементите, отговарящи на поставеното условие); последователно се обхождат всички елементи, като за всеки от тях се проверява дали отговаря на предварително зададения критерий; на всяка стъпка ако елементът удовлетворява зададения критерий, то стойността на

променливата *Vg* се увеличава с 1. По този начин променливата *Vg* ще съдържа точния брой на елементите, които до тази стъпка отговарят на поставеното условие, като в края на обхождането ще съдържа търсения общ брой.

Въпроси и задачи:

1. Каква е разликата между обикновеното и дихотомното търсене?
2. Съставете програма за намиране на най-малкия елемент в масив от 100 цели числа.
3. В едномерния масив *score* са съхранени резултатите (време на спускане в секунди) на участниците в ски-състезание (не повече от 80 на брой). Съставете програма, която намира най-доброто и най-лошото постижение, както и средно аритметичното време за спускане на състезателите.

Решение:

```
import java.io.*;
class compet {
    public static void main (String [] args) {
        float score [] = {80,100,200,30};
        float MIN, MAX;
        float s=0;
        MAX = score [0];
        MIN = score [0];
        for (int I=1; I<score.length; I++)
        {
            if (score[I] > MAX) MAX = score [I];
            if (score[I] < MIN) MIN = score [I];
            s+=score[I];
        }
        System.out.println("Максималното постижение е "+ MAX);
        System.out.println("Минималното постижение е "+ MIN);
        s/=score.length;
        System.out.println("Средното време за спускане е:" + s);
    }
}
```

4. Даден е едномерен масив от 6 елемента. Да се състави програма, която намира най-малкия и най-големия елемент на масива и разменя местата им.

Решение:

```
import java.io.*;
class arrMinMax {
    public static void main (String [] args) {
        float arr4 [] = {1,4,7,9,0,2};
        int MIN=0,MAX=0;
        float buf;
        MAX = 0;
        MIN = 0;
        for (int I=1; I<arr4.length; I++)
        {
            if (arr4[I] > arr4[MAX]) MAX = I;
            if (arr4[I] < arr4[MIN]) MIN = I;
        }
        buf = arr4[MAX]; arr4[MAX] = arr4[MIN]; arr4[MIN]= buf;
        for (int I=0; I<arr4.length; I++)
            System.out.println(arr4 [I]);
    }
}
```

5. Съставете програма за намиране на броя на четните числа в масив от 10 елемента от целочислен тип.

```
import java.io.*;
class ten {
    public static void main (String [] args) {
        int Arr5 [] = {12,34,56,78,6,5,5,45,4,3};
        int br = 0;
        for (int I=0; I < Arr5.length; I++)
            if ((Arr5[I]%2) == 0) br++;
        System.out.println("Броят на четните елементи в масива е: " + br);
    }
}
```

6. В едномерния масив klas са съхранени оценките на учениците от 9^a клас по математика (не повече от 35 на брой). Да се състави програма, която намира най-високата и най-ниската оценка в класа и броя на

учениците, които имат успех над средния успех на класа.

```
import java.io.*;
class ks{
    public static void main (String [] args) {
        int br = 0;
        float klas [] = {5,6,5,4,3,4,5,5,4,6,6};
        float MIN, MAX;
        float s=0;
        MAX = klas [0];
        MIN = klas [0];
        for (int I=1; I<klas.length; I++)
            { if(klas[I] > MAX) MAX = klas [I];
              if (klas[I] < MIN) MIN = klas[I];
              s+=klas[I];
            }
        System.out.println("Максималната оценка е "+ MAX);
        System.out.println("Минималната оценка е "+ MIN);
        s/=klas.length;
        System.out.println("Средният успех на класа е:" + s);
        for (int I=0; I < klas.length; I++)
            {
                if (klas[I] > s) br++;
            }
        System.out.println("Броят на учениците с успех над средния е " + br);
    }
}
```

7. Опитайте се да съставите поне две задачи, които имат практическа стойност и за решаването на които се използват някои от изучените в урока алгоритми. Напишете програми за решаване на тези задачи.
8. Потърсете информация и за други алгоритми за броене и търсене. Опитайте се да съставите съответни програми на Java.

Сортировка

На хората ежедневно се налага да използват информация, подредена по определен начин. Например по азбучен реда са подредени имената на абонатите в телефонния указател и думите в различните речници; на спортните страници на вестниците

отборите се класират по брой на постигнатите точки или други показатели; при игра на карти играчите също подреждат своите карти, за да могат по-лесно да определят как да анонсират.

Използването на речници, каталози, предметни и телефонни указатели показва, че хората предварително подреждат необходимите им данни, за да може по-лесно да ги преглеждат, обработват и съхраняват. Същият подход се използва и при компютърната обработка на големи информационни масиви – почти винаги те се подреждат по определен признак, за да се ускорят следващите операции с тях.

В компютърната информатика вместо термина подреждане се използва терминът сортиране.

Задачата за сортиране на масив може да бъде формулирана така: да се подредят елементите на масива по нарастване (или намаляване) на някоя тяхна характеристика.

Предложени са различни алгоритми за решаване на задачата за сортиране. В информатиката са добре изучени и описани десетки алгоритми за сортиране.

Ще разгледаме два от най-простите алгоритми за сортиране.

Алгоритъм за сортиране “Метод на мехурчето”

Основната **идея** на алгоритъма е: масивът се преглежда толкова пъти, колкото е последния му индекс, като при всяко преглеждане един от елементите на масива се поставя в позицията, която трябва да заема в подредения масив. Всеки елемент се сравнява с другите и ако е по-голям, то си сменят местата. Аналогично масивът може да се сортира и намаляващо т.е. от най-големия елемент към най-малкия.

Словесно описание на алгоритъма:

- На променлива от цял тип се присвоява дължината на масива: `int I = array.length;`

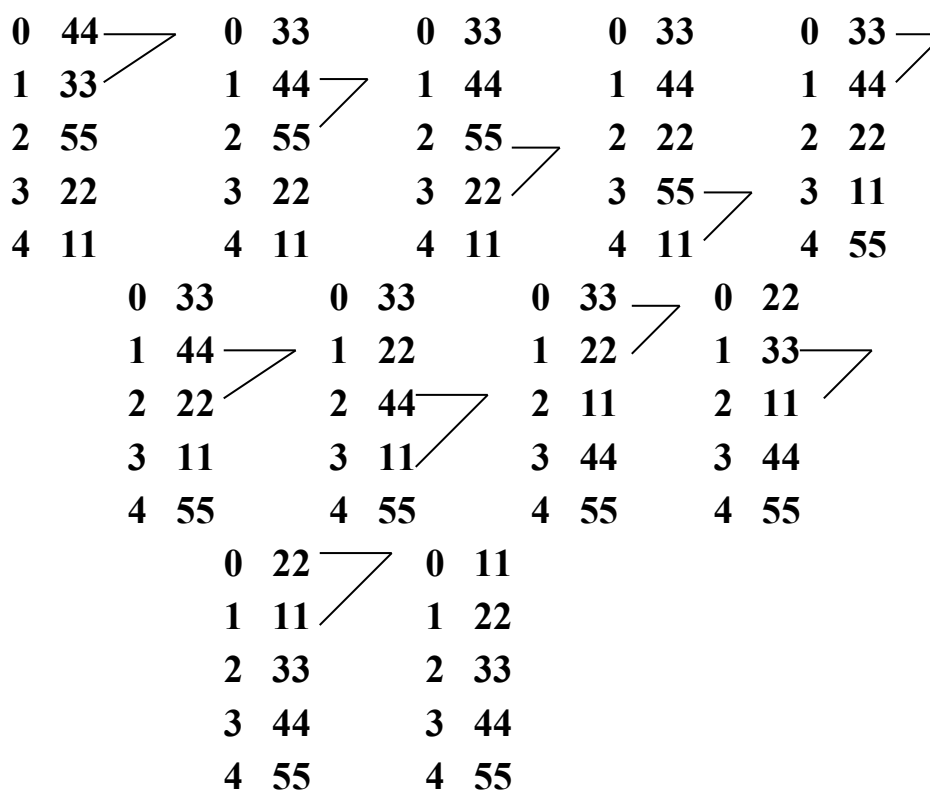
- Описваме цикъл, в който променливата I намалява, докато получи стойност 0, като на всяка стъпка, всеки елемент на масива се сравнява със следващият.
- Ако е по-голям, то местата им се разменят (използваме променлива temp);
- Край на алгоритъма.

Пр. 1 Програма “Сортиране чрез метод на мехурчето”

```
import java.io.*;
public class arrBubble {
    public static void main (String [] args) {
        int array[] = new int[20];
        int I = array.length;
        System.out.println("Before sort:");
        for (int k = 0; k < array.length; k++)
            array[k] = (int)(Math.random()*20.0);
        // запълва масива с произволни цели числа
        while (--I >= 0 )
        {
            for (int j=0; j < I; j++)
            {
                if (array[j] > array[j+1])
                {
                    int temp = array[j];
                    array[j] = array[j+1];
                    array[j+1] = temp;
                }
            }
        }
        System.out.println("After sort:");
        for (int k=0; k < array.length; k++)
        {
            System.out.println(array[k]);
        }
    }
}
```

Илюстрация на метода на мехурчето

Нека е даден масив от пет елемента, тогава действието на алгоритъма е следното:



Алгоритъм за бърза сортировка

За повечето масиви е необходим по-ефективен метод за сортировка. Едни от най-ефективните методи е методът за бърза сортировка. Основната идея на алгоритъма е следната: методът избира средният елемент и разделя масива на две части; едната половина ще съдържа елементи по-големи от средния, а другата, съответно по-малки елементи от средния; този процес се повтаря, докато в крайния масив се съдържа един елемент. В резултат масива ще бъде сортиран в правилен ред, както ще илюстрираме по-долу.

Програма “Бърза сортировка”

```

import java.io.*;
class Qsort {
    public static void main (String [] args) {
        int array [] = new int[1000];
        System.out.println("Before sort:");
        for (int I = 0; I < array.length; I++){
            array [I] = (int) (Math.random()*1000.0);
            System.out.println(array[I]);
        }
        qsort (array, 0, array.length-1);
        System.out.println("After sort:");
        for (int I=0; I < array.length; I++){
            System.out.println(array[I]);
        }
    }

    static void qsort (int array [], int first, int last) {
        int low = first;
        int high = last;
        int mid = array[(first+last)/2];
        do{
            while (array[low] < mid)
                low++;
            while (array[high] > mid)
                high--;
            if (low <= high) {
                int temp = array[low];
                array[low++] = array [high];
                array[high--] = temp;
            }
        } while (low <= high);
        if( first<high)
            qsort (array, first, high);
        if (low<last)
            qsort (array, low, last);
    }
}

```

За да се смени посоката на сортиране (т.е. да намалява: от по-голям елемент към по-малък) е достатъчно в цикъла while да променим знака на неравенството:

```

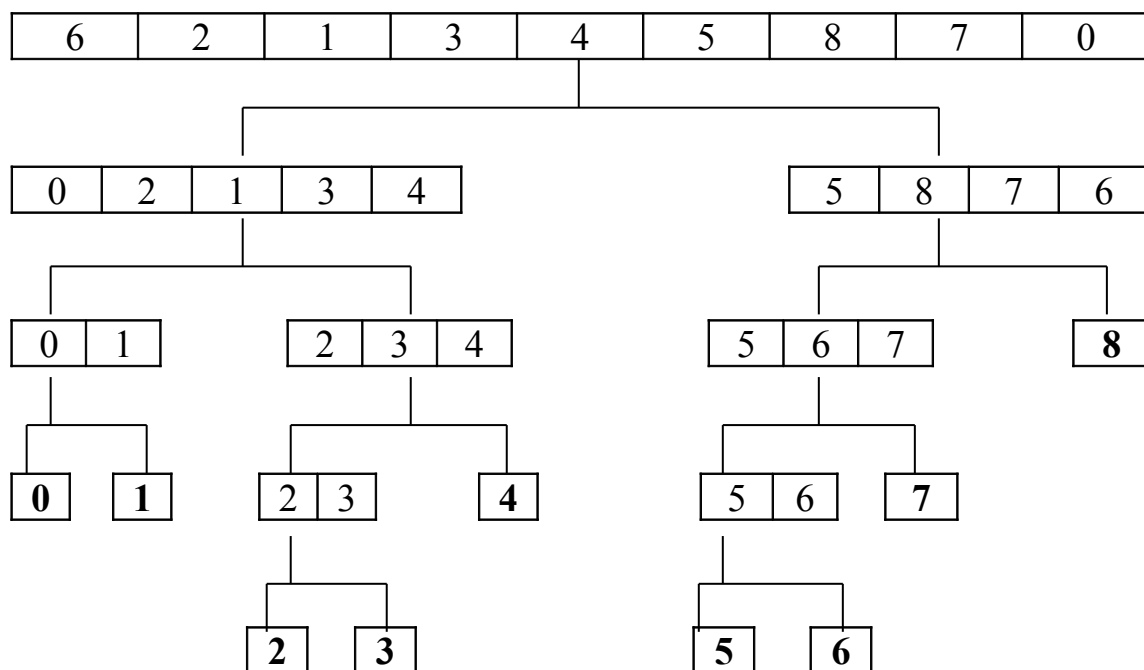
while (array[low] > mid)
    low++;

```

```
while (array[high] < mid)
    high--;
```

Илюстрация на метода за бърза сортировка

Нека е даден масив от девет елемента, тогава действието на алгоритъма е следното:



Въпроси и задачи:

1. Съставете описание на алгоритъма за сортиране по метода на мехурчето, ако масивът се сортира в намаляващ ред.
2. Посочете поне три примера от практиката, при които данните са подредени по определен признак.

3. Даден е масив с име bal, състоящ се от реални числа. Да се състави програма, която въвежда като елементи на масива средния годишен успех на всеки от

```
import java.io.*;
class bal {
    public static void main (String [] args) {
        double array []={4.5, 5.75, 5.5, 3.87, 4.5, 4.75, 6, 5.25, 4.83, 5.45};
        System.out.println("Before sort:");
        for (int I = 0; I < array.length; I++){
            System.out.println(array[I]);
        }
        qsort (array, 0, array.length-1);
        System.out.println("After sort:");
        for (int I=0; I < array.length; I++){
            System.out.println(array[I]);
        }
    }
    static void qsort (double array [], int first, int last) {
        int low = first;
        int high = last;
        double mid = array[(first+last)/2];
        do {
            while (array[low] < mid)
                low++;
            while (array[high] > mid)
                high--;
            if (low <= high) {
                double temp = array[low];
                array[low++] = array [high];
                array[high--] = temp;
            }
        } while (low <= high);
        if( first<high)
            qsort (array, first, high);
        if (low<last)
            qsort (array, low, last);
    }
}
```

учениците от 9^a клас и ги подрежда в низходящ ред.

4. Опишете словесно метода на бързото сортиране.

5. Потърсете и се запознайте и с други алгоритми за сортиране. Съставете описание на тези алгоритми, както и програми, които ги реализират.

ТЕМА 15:

1. Да се състави рекурсивен метод, който по зададено естествено число N пресмята произведението на естествените числа между 1 и N със стъпка 4. Например при $N=11$ стойността на произведението е 45 (1.5.9).
2. Редица на Фибоначи се нарича редицата от числа $f_0, f_1, f_2, f_3, \dots$, в която $f_0=f_1=1$ и $f_{k+1} = f_{k-1} + f_k$ за $k=1, 2, 3, \dots$. Да се състави рекурсивна метод, който за дадено n отпечатва f_n .
3. Даден е едномерен масив A от цели числа с не повече от 100 елемента $a_0, a_1, a_2, \dots, a_n$. Съставете рекурсивна подпрограма, която за зададена стойност на реалната променлива x , пресмята стойността на дробта:

$$a_0 + \frac{1}{a_1 + \frac{x}{a_2 + \frac{x^2}{a_3 + \frac{x^3}{\dots a_{n-1} + \frac{x^{n-1}}{a_n}}}}}$$