Метод. Декларация на метод. Синтаксис. Формални параметри. Фактически параметри. Стек. Стекова архитектура. Предаване на параметри чрез стек. Замяна на фактически с формални параметри. Предаване на параметри по стойност и по име (препратка).Тип на резултат на метод. Локални променливи. Припокриване областта на действие на локални променливи. Оператор за възврат **return**. Стратегия "Разделяй и владей".

**Методът** е функция, описваща последователност от действия над дадени поменливи (формални параметри или полета на класа), които водят до конкретен резултат – процедурен или функционален. Методите са част от класа и описват функционалността (операциите) над неговите екземплярни променливи (полетата).

**Декларация на метод:**

<декларация-на-метод>::=<тип–на–върнат–резултат> <име–на–метода> <списък-от-формални-параметри>
“{“
 <последователност-от-оператори>

“}”

 <списък-от-формални-параметри> ::= “(“[ <деклрация-на-формален-параметър> {“,”<деклрация-на-формален-параметър> } ] “)”

Както се вижда при декларацията на метод не е задължително методът да има формални параметри. Същота така методът може да изпълнява само определени процедури и да не връща стойностен резултат.

Последователността от оператори на методът се загражда винаги с фигурални скоби и представялва тяло на метода.

Методите могат да се декларират навсякъде в класа и всички конструктури и методи на този клас има достъп до тях.

Параметрите биват 2 вида – **формални и фактически.** Първите се декларират в метода и нямат конкретна стойност. Те я придобиват при извикване на метода и са видими само в него, т.е. те са локални променливи за конкретния метод. Формалните параметри се записват на върха на стековата памет и след изпълнението на метода се изтриват от стека.

Фактическите параметри имат конкретни стойности и се използват при извикване на метода. Фактическият параметър може да бъде променлива или някакъв аритметичен или булев израз. Тъпът на фактическият и формалният параметър трябва да съвпадат.

<извикване-на-метод>::=<име-на-метод> <списък-фактически-параметри>

<списък-фактически-параметри>::=”(“[ <фактически-параметър> {“,” <фактически-параметър> } ] “)”

<фактически-параметър> ::= <булев-израз>|<аритметичен-израз>|<променлива>|<константа>

В езиците за програмиране съществуват два начина за замяна на фактически на формален параметър – по адрес и по стойност. При замяната по адрес фактическията и формалният параметър ползват една и съща клетка памет за съхраняването на определена стойност и при промяна формалния параметър се променя и фактическия параметър, което често води до програмни грешки, т.е. 2-те променливи не са отделени една от друга. Замяната по стойност заделя нова памет в стека за формалния параметър и копира стойността на фактическия. Така 2-те променливи са физически независими, но това води до ползване на по-голямо количчество памет. В Java замяната на фактически с формален параметър става по стойност. В C/C++ и други езици това става по адрес.

Стек – вид линейна организация на паметта, при която полетата са подредени едно над друго и добавянето и премахването на нови такива става само на върха на стека. За стека е валидно твърдението последния влязъл е първият излязъл (LIFO – last in first out). При стековата архитектура премятането на изрази става по обратен полски запис:

a+b = +(a,b)

Методите могат да имат за пълноправни параметри както нисши типове така и обекти. Също така резултатът от методът може да бъде пак обект.

В Java не могат да бъдат пропускани параметри, т.е. ако методът има декларирани 3 формални параметъра не може да го извикаме само с 2 фактически.

Операторът return се използва за върщане на стойност на метода. Когато методът е процедурен този оператор може да се изпусне. Във всички останали случаи е задължителен и след него следва израз, чиято стойност по тип отговаря на типа на метода. Този израз се присвоява като резултат на метода. Операторът return прекратява по нататъчното изпълнение на метода.

int sum(int a, int b)

{
 return a+b;
}

int sum(int a, double b)

{
 return (int)(a+b); //неявно преобразуване на а от int в доубле //явно преобразуване на аритметичен израз от double в int
}

int sum(double a, double b)

{
 return (int)(a+b); //явно преобразуване на аритметичен израз от double в int
}

void print(int a)
{
System.out.println(a); //резултатът е процедурен

return; //след return няма нищо; return може да се пропусне
}

Често срещано явление в програмирането е **претоварването на имена** – многократното ползване на едни и същи означения за имена на променливи, методи и конструктури.

Например в Java могат да съществуват методи с едно и също име, които обаче трябва да се различават по брой или тип формални параметри. В такъв случай изборът кой метод да се изпълни се прави по типът и броят на фактическите параметри при извикването на метода.

Например горният метод sum съществува в два варианта:

Int sum( int int)

Int sum (int double)

Int sum (double double)

При извикване на sum по следния начин:

sum(4, 10.4);

компилаторът ще изпълни 2-рият метод.

При извикване на sum (4.3, 2); това е извикване на метод с параметри (double, int) които не е описан. В такъв случай компилаторът преобразува неявно int в сложен тип double и се обръща към метода int sum(double double).

Друг пример за претоварване на имена е припокриването на областна на действие на локални и глобални променливи. Например класът има поле с име “a” (глобална пром.), методът има формален параметър (лок. Пром.) със същото име като и 2-те се ползват в метода. Тогава имаме припокриване на области на действие на 2 променливи и в такъв случай по-близко дефинирата е определяща.

Class test

{

Int a=10;

Int sum(int a, int b)

{

 Return a+b; //а е стойността на формалния параметър

}

Int sum (int a, int b)

{

 Return this.a + a +b;

}

Чрез операторът this се обръщаме към екземплярната променлива на класа, която е глобална за метода.

**Стратегията „Разделяй и владей”** е алгоритмична техника при която някакъв проблем се разделня на няколко по-малки проблема (най-често 2) и след това чрез обединение на техните решения се получава решение на цялостния проблем. Такива алгоритми протичат в 3 стъпки: разделяне на проблема на по-малки; решаване на малките проблеми; обединяване на малките проблеми. Пример за алгоритми ползващи такава страгегия са бърза сортирова, сортировка със сливане, Ханойски кули и много др.