Л Е К Ц И Я №2

Основи на програмния език Java

Обектно ориентирано програмиране

I. ВЪВЕДЕНИЕ В ОБЕКТНО-ОРИЕНТИРАНОТО ПРОГРАМИРАНЕ

Основната част от съвременните програмни езици поддържат обектно ориентирано програмиране (ООП). Трябва да се има предвид, че повечето програмни езици са проектирани да решават добре конкретни проблеми. Например C е програмен език с общо приложение, който след компилиране генерира компактен изпълним код и има добро бързодействие. Поради тази причина той е процедурен език, а не обектно-ориентиран. Основен проблем на повечето процедурни езици е, че при нарастване на програмния код той става по-нестабилен. Обектно ориентираните езици са подходящи за създаване на средни до големи по размер програмни проекти. В този случай, за да е стабилен програмния код не трябва да се разчита на опита на програмистите, а трябва самият програмен език да не позволява създаване на нестабилен код и да има добра поддръжка на обслужване на грешки. Тези грешки са от тип *изключения* – могат да бъдат отстранени програмно. Програмният език Java е обектно ориентиран, проектиран е да генерира стабилен код при работа в среда, където възникването на изключения е много вероятно. Тъй като силата на Java е при работа в мрежова среда, този език е платформено независим и има много добра система за обработка на изключения. Други езици са специализирани основно за математически изчисления (Matlab), за обработка на статистическа информация (R), оптимизирани за функционално програмиране (Haskel) и др. Най-използваните на този етап програмни езици най-често са хибридни – обектно-ориентирани с поддръжка на функционално програмиране (Java, C#, Python).

ООП се базира на четири основни *концепции*:

- *Абстракция;*
- *Капсулиране*;
- *Наследяване*;
- *Полиморфизъм.*

Абстракцията има за цел сложни неща да се реализира чрез прости решения. Чрез използване на програмни класове, обекти и променливи могат да се опишат програмно произволно сложни системи. Това описание може да се използва многократно.

Капсулирането позволява защита на данните на обектите. За целта, за тези данни се указва те да бъдат частни. Ако се налага достъп до тях, той се реализира чрез публични методи. Капсулирането ни позволява да използваме функционалността на обекта, без да застрашаваме сигурността. Тази концепция помага на програмистите да пестят време. Например, може да създадем част от код, който изисква конкретни данни от база данни. Може да е полезно да използвате този код с други бази данни. Капсулирането ни позволява да направим това, като запазим нашите оригинални данни частни. Чрез капсулацията става възможно едно и също програмно име да има различни значения в различни контексти (пространство на имената).

Наследяването е концепция, която позволява на програмистите да създават нови класове, които споделят всички или избрани свойства на съществуващи класове. Класът, който се наследява се нарича *родителски*, а наследникът – *дъщерен* клас.

Полиморфизмът е термин, който буквално означава „много форми". В програмирането полиморфизмът се свежда до способността на различни обекти да реагират на едно и също послание по различни начини. Например, ако опишем програмно чат бот, той трябва да може да говори. За целта всеки обект-бот ще има един и същи метод за синтез на говор, но различните ботове ще говорят на различни езици, в зависимост от конкретния клиент. Полиморфизмът се базира на възможността за предекларирane и предефиниране на методи. Полиморфизмът има още много форми. Например, той се проявява и при множествено наследяване или имплементиране на множество програмни интерфейси.

II. ПРОГРАМНИ КЛАСОВЕ И ОБЕКТИ

2.1. Програмни обекти

Програмните обекти симулират състоянието и действието на реални или виртуални обекти. Състоянието на обекта се описва чрез *член-променливи* (*полета* в Java), а действията, които обектът може да реализира, чрез *член-функции* (*методи* в Java).

> **Програмният обект** е специфична комбинация от *променливи*, описващи състоянието му, и *методи* чрез които се реализира неговата функционалност.

Например, ако обектът е мобилен телефон, състоянието му може да се опише чрез променливи като: тип (смартфон), размери (в/ш/д), цвят, капацитет на батерията, мрежови характеристики и много др. Функционалността на мобилния терминал може да се реализира чрез методи, като: регистриране в мрежата, провеждане на разговор, изпращане/получаване на SMS, край на работа с мрежата и др. Програмният обект реално заема оперативна памет. Тя е необходима за променливите и методите, които го описват. За разлика от C++, при който обектите могат да се създават в стека и в динамичната памет (хийп), при Java обектите се създават *само в хийпа* чрез запазената дума `new`. Причината за това ограничение е да се гарантира сигурността и надеждността на Java приложенията. Тъй като в даден момент в паметта могат да съществуват множество копия на даден обект, те трябва да се разграничават по някакъв начин от JVM. За целта компилаторът генерира уникално число, наречено *инстанция* на обекта. За променливите от примитивен тип при Java не се създават обекти в хийпа чрез new. Достъпът до тях се реализира по стойност, която се записва в стека. От гледна точка на бързодействието и прегледността на изходния код, това решение е по-ефективно. Променливите и методите на всеки обект са изолирани от тези на останалите обекти. На практика се налага обектите да комуникират помежду си и следователно пълната изолация няма практически смисъл. За целта се задават права за достъп до обектите чрез т.нар. *модификатори на достъпа*, които могат да се назначават както за променливи, така и за методи и класове.

2.2. Програмни класове

За да може да се създаде даден обект е необходимо операционната система (JVM при Java) да знае колко памет е необходима за неговите компоненти (променливи и методи). Описанието на тези компоненти се реализира чрез *класове*.

Класът може да се разглежда като шаблон чрез който могат да се генерират (чрез ключовата дума `new`) толкова обекти, колкото са необходими. Самият клас не генерира програмен код - той съдържа само описанието на компонентите на бъдещите обекти.

Примерната структура на Java клас е показана на Фиг.2.1.

Модификатор на достъпа	**class**	ИмеНаКласа	**extends** ИмеНаКлас	**implements** ИмеНаИнтерфейс_1,..., ИмеНаИнтерфейс_N
public	**class**	MyFirstClass		

{

// Константи и променливи, достъпни за всички методи от класа:

Модификатор на достъпа	Тип на променливата	Име на променливата	= стойност
`final static`	`int`	`NUM_OF_ITEMS`	`= 10;`
`private`	`int`	`mySum;`	

// Конструктор на класа:

`public`		`MyFirstClass()`

```
{
    mySum = 0;
}
```

// Методи:

Модификатор на достъпа	Тип на връщаната стойност	Име на метода	(Тип на атрибут 1	Име на атрибут 1	, ...)
`private`	`int`	`sum`	`( int[]`	`args`	`)`

```
{
    int sum = 0;
    int N = args.length;
    for (int i=0; i<N; i++) {
          sum += args[i];
    }
    return sum;
}
```

Тяло на класа

}

Фиг. 2.1. Примерна структура на Java клас

Полетата на сив фон са *задължителни*. При деклариране на нов клас се използва ключовата дума class. ***Конструкторът*** в Java е специален *метод* с име, съвпадащо с името на класа в който е дефиниран. Той се вика от JVM веднага след като от класа се създаде обект. Затова конструкторът се използва най-често с цел инициализация на състоянието на обекта и отпускане на необходимите за неговото функциониране ресурси. От един клас могат да се създадат колкото желаем на брой обекти чрез ключовата дума new. Синтаксисът е показан на Фиг.2.2. Java позволява в един клас да се дефинират *няколко* конструктора. Разликата в конструкторите е само в броя и/или типа на предаваните атрибути. По този начин от един клас могат да се създават обекти чрез викане на различните му конструктори. Конструкторът, както всеки метод, може да връща стойност и тя най-често е индикатор за статуса на новосъздадения обект.

```
// деклариране на обект:
Име_на_класа        име_на_обекта;
 MyFirstClass       myFirstObject;
// създаване на обект:
име_на_обекта= new Име_на_конструктора (списък от атрибути);
 myFirstObject = new MyFirstClass();
// едновременно деклариране и създаване на обект:
 MyFirstClass myFirstObject = new MyFirstClass();
```

Фиг. 2.2. Деклариране и създаване на обект

В Java *не се използват указатели*, когато се предават параметри към даден метод. Те се предават *само по стойност*, независимо дали са от примитивен тип или обекти. Методите работят с копия на предаваните им променливи и обекти. Следователно, промяната на стойността на обект в тялото на метод, не променя реалния обект (промяната е в сила само за копието на обекта, чието време на живот е в рамките на тялото на метода). Структурата на класовете в Java и C++ се различава минимално. Основната разлика е в липсата на *деструктори* в Java. Докато конструкторът има за задача да инициализира бъдещия обект, който се създава от класа, то деструкторът трябва да освободи всички заети от обекта ресурси, преди да се прекрати неговото използване. Тази задача в Java (аналогично е и при C#) се изпълнява от специален модул от JVM, наречен **Garbage Collector** (GC). Този модул анализира в "реално време" кои обекти трябва да бъдат унищожени, най-често чрез анализ на референциите към тях. Това има своите предимства, но и недостатъци. Основното предимство е, че програмистът вече не се интересува от освобождаване на заетата от обектите памет. Досадното и характерно за C++ "изтичане на памет", поради забравяне от страна на програмиста да освободи заета памет, при Java е невъзможно. За сметка на това GC намалява бързодействието на приложенията, тъй като работи "паралелно" на кода на приложението. Понякога изчистването на паметта може да изисква до стотина ms и това трябва да се има предвид. Все пак, ако се налага, програмистът може принудително да активира GC като извика метод gc от клас System:

```
System.gc().
```

2.3. Наследяване

Мощността на обектно-ориентираното програмиране се свежда основно до възможността за *наследяване* на функционалност. Наследяването предоставя възможността даден клас, който ще наричаме *дъщерен*, да получи цялата или част от функциалността на друг(и) клас(ове), който(които) ще наричаме *родителски*. Всеки Java клас може да бъде едновременно дъщерен за определени класове и родителски за други. В Java има един базов супер-клас Object от пакета java.lang. Реално, всички класове са наследници на този супер-клас. Наследяването се реализира чрез използване на ключовата дума **extends**, след която следва името на наследявания родителски клас (виж Фиг .2.1). За разлика от други програмни езици като C++, Java не поддържа *множествено наследяване*. Следователно един Java клас може да наследи *само един* родителски клас. Това на пръв поглед ограничение, което Java налага, има своя практически смисъл - множественото наследяване е мощно средство в обектно-ориентираното програмиране, но предполага възможни проблеми с новите свойства на получения клас, особено при начинаещите

програмисти (diamond problem). Следователно, без множествено наследяване се повишава стабилността на програмния код.

Java позволява да се симулира множествено наследяване чрез използване на *програмни интерфейси*. Всеки Java клас може да наследи само един родителски клас, но може да реализира множество интерфейси. За целта се използва ключовата дума **implements**. Интерфейсите са специфичен Java тип, подобно на класовете. За разлика от класовете, интерфейсите не могат да бъдат инстанциирани, а методите от интерфейсите се декларират без тяло. Това означава, че функционалността на методите трябва да се реализира в тялото на класа, който използва интерфейса. Така един програмист може да контролира правилното използване на неговия код от други програмисти.

От Java 8 е възможно в тялото на интерфейс да се декларират методи, които съдържат програмен код. За целта се използва ключова дума default при деклариране метода.

2.4. Предекларирane и предефиниране на методи

Предекларирането (***overloading***) предоставя на програмиста възможността да използва методи с едно и също име, но с различен брой или тип на предаваните аргументи. Типът на връщаната от метода стойност *не може* да се променя. Предекларирането може се използва за методите в рамките на един клас, или в рамките на дъщерния клас, ако е използвано наследяване. Следва пример при който метод getSum има две декларации: в първият вариант на метода се сумират стойностите на два атрибута от тип int, а във втория вариант – стойностите (тип int) на елементите от масив.

```
// ---------------------- вариант 1
private int getSum(int a, int b) {
    return (a+b);
}
// ---------------------- вариант 2
private int getSum(int[] a) {
    int sum=0;
    for (int i=0;i<a.length;i++)
         sum += a[i];
    return sum;
}
```

Пример за overloading на метод

Предефинирането (***overriding***) се използва когато е необходимо да се промени (частично или напълно) логиката на работа на даден метод. За целта в дъщерния клас се разписва кода от тялото на метода, като задължително се запазват броя и типа на предаваните атрибути и типа на връщаната стойност. Предефинирането позволява по елегантен начин да се промени функционалността на дъщерните класове. Задачата на програмиста е да реши кои методи да запази и кои да предефинира. Разбира се, той може да декларира и нови методи в дъщерния клас и така да формира желаната от него функционалност.

2.5. Модификатори на достъпа

Програмните обектите по подразбиране са изолирани един от друг. Следователно, когато се налага адресиране на полетата на даден обект е необходимо изрично да се декларира за кого това е разрешено. За целта за всеки клас, метод и поле е възможно явно да се зададат правата за достъп до тях чрез ключови думи - *модификаторите на достъпа*. При Java е позволено да се пропусне модификатора на достъп. В този случай JVM назначава модификатор на достъп по подразбиране.

Модификаторите на достъпа при Java са ключови думи, които определят *достъпността* на данните, методите и самите класове. Основните модификатори за достъп са: public, protected, private. Достъпът се управлява и чрез ключови думи abstract, final и static.

В Табл. 2.1 са описани правата за достъп до класове, данни и методи при изброените модификатори.

Табл. 2.1. Ключови думи в Java, свързани с достъпа до програмния код

Модификатор	Клас	Метод	Променлива
public	Достъпен за всички класове, независимо от пакета.	Достъпен за всички класове, независимо от пакета.	Достъпна за всички класове, независимо от пакета.
protected	*Не се използва*	Достъпен за всички класове, както и за всички наследници на класа.	Достъпна за всички класове в рамките на пакета и за всички наследници на класа.
private	Класът е недостъпен (използва се за създаване на	Достъпен в рамките на класа	Достъпна в рамките на класа.
abstract	Невъзможно е получаването на инстанция на класа. За да се използва абстрактен клас, той се наследява, а след това се дефинират абстрактните методи, които съдържа.	Ако даден метод е абстрактен, то и класа, които го съдържа, е абстрактен. Абстрактните методи нямат тяло.	*Не се използва*
final	Класът не може да бъде наследяван.	Не се поддържа overriding (Кодът на метода не може да се променя при наследяване)	Променливата не може да бъде променяна, става константа.
static	Може да се използва само при *вложени* програмни класове.	Не могат да се адресират чрез this (текущата инстанция на класа) и следователно нямат достъп до нестатични методи и променливи в рамките на класа.	Статичната променлива принадлежи на родителския клас, а не на всяка инстанция на класа.

III. ВЪПРОСИ И ЗАДАЧИ ЗА ИЗПЪЛНЕНИЕ

1. Какви са основните концепции на които се основава ООП?

2. Защо Java не поддържа множествено наследяване? Потърсете в Интернет информация за „diamond problem" в ООП.

3. Колко програмни интерфейса може да имплементира един Java клас? Възможни ли е един програмен интерфейс да наследи функциалността на други интерфейси?

4. Да се напише Java конзолно приложение, което описва домашни животни, например прасе и крава. За всяко животно трябва да може да се задава: какво е *теглото* му в килограми (цяло число); *идентификационния му код* (низ); какъв звук издава животното (низ), метод чрез който се показва какъв *звук* издава животното и метод чрез който се получава на конзолата *обща информация* за животното.

Тъй като се описват различни домашни животни може да се използва наследяване. *Кога трябва да използвате наследяване*? Когато връзката между класовете е чрез Е, например: „прасето Е домашно животно". Следователно, родителският клас ще бъде „домашно животно", а дъщерните „прасе". *Каква информация трябва да има в базовия клас*? В базовия клас трябва да бъдат всички свойства и методи, които са общи за всички наследници. В конкретния пример: тегло и id код. Всяко едно от животните издава звук, но различен. Това означава, че родителският клас трябва да има метод за издаване на звук, който обаче трябва задължително да се пренапише във всеки дъщерен клас. Това налага този метод да е абстрактен. Ако един клас съдържа абстрактни методи, то и класът трябва да е абстрактен. За всяко животно трябва да може да се връща информация, но тя също е различна. Следователно и този метод може да се декларира в родителския клас като абстрактен.

На Фиг. 2.3 е показано примерно решение на Задача 4. Извън тялото на базовия клас (този, който съдържа метод main) декларирайте класове ДомашноЖивотно, Прасе и Крава. Тъй като входната информация за всеки тип дъщерен клас е еднаква (тегло, id и звук) тя се задава чрез конструктора на базовия клас - ДомашноЖивотно. Забележете как се прави разлика между променливите, които се предават чрез конструктора и променливите, които са член-данни на класа. Тъй като имат еднакви имена, член-данните се адресират чрез ключова дума this (тази инстанция). Тъй като методи звук() и информация() печатат различна информация, то те са декларирани като абстрактни. Забележете, че абстрактните класове *нямат тяло*.

Да разгледаме и кода на клас Прасе. В конструктора му е използван метод super чрез който се вика конструктора на родителския клас. Следователно, при създаване на обект от клас Прасе ще се извика конструкторът му, а super ще извика конструктора на ДомашноЖивотно. Кодът от него ще запомни всички входни данни за животното в съответните член-данни. *Внимание!* Метод super трябва да е първият метод от тялото на канструктура (ако има и други методи).

От тялото на метод main се „създава" едно прасе и една крава чрез ключова дума new. След тази ключова дума следва името на конструктора на който се предават необходимите параметри. Викането на метод се реализира като инстанцията на обекта (прасе, крава) и името на метода се свързват със точка:

```
прасе.информация();
```

При изпълнение на този код на конзолата се разпечатва следното:

```
Прасе с id 1111111 тежи 100 кг. и казва грух.
Звук на крава: мууу
Крава с id 2222222 тежи 220 кг. и казва мууу.
```

```
abstract class ДомашноЖивотно {
    int тегло;
    String id;
    String звук;
    public ДомашноЖивотно(int тегло, String id, String звук) {
        this.тегло = тегло;
        this.id = id;
        this.звук = звук;
    }
    abstract void звук();
    abstract void информация();
}
// ---------------------------------------------------------------------------
class Прасе extends ДомашноЖивотно {
    public Прасе(int тегло, String id, String звук) {
        super(тегло, id, звук);
    }
    @Override
    void звук() {
        System.out.println("Звук на прасе: " + звук);
    }
    @Override
    void информация() {
        String info = String.format("Прасе с id %s тежи %d кг. и казва %s.",
                id, тегло, звук);
        System.out.println(info);
    }
}
```

а) Програмни класове (декларирайте клас Крава, подобно на клас Прасе)

```
public static void main(String[] args) {
    ДомашноЖивотно прасе = new Прасе(100,"1111111", "грух");
    прасе.информация();
    ДомашноЖивотно крава = new Крава(220,"2222222", "мууу");
    крава.звук();
    крава.информация();
}
```

б) Метод main – създаване на обекти и викане на техните методи

Фиг. 2.3. Примерно решение на Задача 4 (вариант 1)

Програмният код работи коректно, но има следния недостатък: Методи звук() и информация() се различават само по името на животното. Ако е възможно името да се извлече програмно, то тези методи могат да бъдат декларирани само в тялото на базовия клас.

Името на животното съвпада с името на програмния клас, който го описва. В Java името на класа може да се извлече програмно по следния начин:

```
String name = this.getClass().getSimpleName();
```

Подобреният вариант на решението на Задача 4 е показан на Фиг. 2.4. Написан е нов метод задайИме() чрез който се получава името на животното.

```
abstract class ДомашноЖивотно {
    int тегло;
    String id, звук, име;
    public ДомашноЖивотно(int тегло, String id, String звук) {
        this.тегло = тегло;
        this.id = id;
        this.звук = звук;
    }
    void задайИме(String име) {
        this.име =име;
    }
    void информация() {
        String info = String.format("%s с id=%s тежи %d кг. и казва \"%s\".",
                име, id, тегло, звук);
        System.out.println(info);
    }
    void звук() {
        System.out.println(String.format("Звук от %s: %s", име, звук));
    }
}
// ------------------------------------------------------------------------
class Прасе extends ДомашноЖивотно {
    public Прасе(int тегло, String id, String звук) {
        super(тегло, id, звук);
        задайИме(this.getClass().getSimpleName());
    }
}
```

Фиг. 2.4. Примерно решение на Задача 4 (вариант 2)

При изпълнение на този код на конзолата се разпечатва следното:

```
Прасе с id=1111111 тежи 100 кг. и казва "грух".
Звук от Крава: мууу
Крава с id=2222222 тежи 220 кг. и казва "мууу".
```

5. Да се напише Java конзолно приложение, което описва програмно двумерни геометрични фигури (квадрат, правоъгълник, прост полигон и окръжност). Да се предостави възможност за изчисляване на периметъра и площта на всяка от фигурите.

Задачата ще бъде решена на Семинарно упражнение №2.