Д И П Л О М Н А   Р А Б О Т А

На тема:

Учебно помагало „Обектно-ориентирано програмиране с Java“

Дипломант: Владислав Руменов Гоцов

Фак.номер: 0701181043

Научен ръководител: доц. Д-р Коста Гъров

Съдържание:

1. Увод.

2. Създаване и използване на обекти.

3. Обработка на изключения.

4. Символни низове.

5. Дефиниране на класове.

6. Линейни структори от данни.

7. Дървета и графи.

8. Принципи на обектно-ориентираното програмиране.

9. Заключение.

10. Литература.

1. Увод.

Обектните (обектно-базирани) езици за програмиране позволяват дефиниране на класове и създаване и опериране с техните обекти. В допълнение към това обектно-ориентираните езици за програмиране позволяват създаване на йерархии от класове, породени от релация на наследяване, а също така и реализация на полиморфизъм.

Същността на обектно-ориентирания подход се състои в предаването на съобщения между обектите. Обектите са основни единици на програмата, които комбинират информация за състоянието и поведението. Всеки обект може да изпраща и да реагира на стимули. За разлика от процедурното програмиране, където отначало се определят структурите от данни и след това се подават като параметри на процедурите, при ООП обектите се дефинират заедно със съобщенията, на които те ще реагират.Съществуват 5 компоненти на обектно-ориентираната парадигма: обект, съобщение, клас, наследяване и метод. Тези компоненти много силно зависят една от друга, при това всяка се дефинира в термините на останалите.

2. Създаване и използване на обекти.

Обект.

Под ***обект*** се разбира познаваем предмет, елемент или същност (реална или абстрактна), имащ важно функционално предназначение в разглежданата приложна област. Всеки обект има **състояние**, **поведение** и **индивидуалност**. **Състоянието** на обекта се характеризира с изброяване на всички възможни свойства на обекта и текущите стойности на тези свойства. **Поведението** се характеризира с възможните действия, които могат да се извършват от обекта и с обекта. **Индивидуалност** означава, че всеки обект има самостоятелност на свойствата и поведението, независими от тези на останалите подобни обекти.При програмното представяне на реален обект се използва негов модел. Разглеждат се не всички негови свойства и елементи на поведението, а само тези които ни интересуват за решаване на конкретен проблем. Разглежданите свойства се наричат ***атрибути.*** Атрибутите се реализират в програмата под формата на ***променливи***, организирани в структура. Всеки атрибут има област на допустимите стойности. Разглежданите действия, които са елементи на поведението на обекта се наричат ***операции*** или още ***методи***. Те се реализират в програмата под формата на ***функции*** (процедури).

Съобщение: Взаимодействието между обекти се осъществява чрез съобщения (messages). Съобщението предизвиква реакция у обекта-получател. Той изпълнява посочената в съобщението операция и връща управлението на обекта подател. Чрез съобщенията се описва поведението на обектите и на ОО-система като цяло.

Клас.

Класът дефинира абстрактните характеристики на даден обект. Може още да се каже, че класът е план или шаблон, който описва природата на нещо (някакъв обект). Класовете са градивните елементи на ООП и са

неразделно свързани с обектите. Нещо повече, всеки обект е представител на единствен точно определен клас.
Ще дадем пример за клас и обект, който е негов представител. Нека имаме клас **Dog** и обект **Lassie**, който е представител на класа **Dog** (казваме още обект от тип **Dog).** Класът **Dog** описва характеристиките на всички кучета, докато **Lassie** е конкретно куче**.**
Класовете предоставят модулност и структурност на обектно-ориентираните програми. Техните характеристики трябва да са смислени в общ контекст, така че да могат да бъдат разбрани и от хора, които са запознати с проблемната област, без да са програмисти. Например, не може класът **Dog** да има характеристика "RAM памет" поради простата причина, че в контекста на този клас такава характеристика няма смисъл.

Какво представляват класовете в Java?

Класът в Java се дефинира чрез ключовата дума **class**, последвана от идентификатор (име) на класа и съвкупност от член-данни и методи, обособени в собствен блок код.
Класовете в Java могат да съдържат следните елементи:

- Полета (fields) – член-променливи от определен тип;
- Свойства (properties) – това са специален вид елементи, които разширяват функционалността на полетата като дават възможност за допълнителна обработка на данните при извличането и записването им. Ще се спрем по-подробно на тях в темата "Дефиниране на класове";
- Методи – реализират манипулацията на данните.

Примерен клас

Ще дадем пример за прост клас в Java, който съдържа изброените елементи. Класът **Cat**моделира реалния обект котка и притежава свойствата име и цвят. Посоченият клас дефинира няколко полета, свойства и методи, които по-късно ще използваме наготово. Следва дефиницията на класа (засега няма да разглеждаме в детайли дефиницията на класовете –

ще обърнем специално внимание на това в главата "Дефиниране на класове"):

```
public class Cat {
private String name;
private String color;
public String getName() {
return this.name;
}
public void setName(String name) {
this.name = name;
}
public String getColor() {
return this.color;
}
public void setColor(String color) {
this.color = color;
}
public Cat() {
this.name = "Unnamed";
this.color = "gray";
}
public Cat(String name, String color) {
this.name = name;
this.color = color;
}
374 Въведение в програмирането с Java
public void sayMiau() {
System.out.printf("Cat %s said: Miauuuuuu!%n", name);
}
}
```

Извикването на метода **System.out.printf(…)** на класа **java.lang.System** е пример за употребата на **системен клас** в Java. Системни наричаме класовете, дефинирани в стандартните библиотеки за изграждане на приложения с Java (или друг език за програмиране). Те могат да се използват във всички наши приложения на Java. Такива са например класовете **String**, **System** и **Math**, които ще разгледаме малко по-късно. Важно е да се знае, че имплементацията на класовете е **капсулирана** (скрита). При използването на методите на даден клас от приложния програмист, тяхната имплементация е независима от употребата им. При

системни класове имплементация обикновено дори не е достъпна за програмиста, който ги използва. Това е така, защото за програмиста е от значение какво правят методите, а не как го правят. По този начин се създават **нива на абстракция**, което е един от основните принципи в ООП.

Ще обърнем специално внимание на системните класове малко по-късно. Сега е време да се запознаем със създаването и използването на обекти в програмите.

Създаване и освобождаване на обекти

Създаването на обекти от предварително дефинирани класове по време на изпълнението на програмата става чрез оператора **new**. Новосъздаденият обект обикновено се присвоява на променлива от тип, съвпадащ с класа на обекта. Ще отбележим, че при това присвояване същинският обект не се опира. В променливата се записва само **референция** към новосъздадения обект (неговият адрес в паметта). Следва прост пример как става това:

```
Cat someCat = new Cat();
```

На променливата **someCat** от тип **Cat** присвояваме новосъздадена инстанция на класа **Cat**. Променливата someCat стои в стека, а нейната стойност (инстанцията на класа **Cat**) стои в динамичната памет.

Създаване на обекти със задаване на параметри

Сега ще разгледаме леко променен вариант на горния пример, при който задаваме параметри при създаването на обекта:

```
Cat myBrownCat = new Cat("Johnny", "brown");
```

В този случай искаме обектът **myBrownCat** да представлява котка, която се казва Johnny и има кафяв цвят. Указваме това чрез думите **"Johnny"** и **"brown"**, написани в скоби след името на класа.

При създаването на обект с оператора **new** се случват две неща**:** заделя се памет за този обект и се извършва начална инициализация на член-данните му. Инициализацията се осъществява от специален метод на класа, наречен **конструктор**. В горния пример инициализиращите параметри са всъщност параметри на конструктора на класа. Ще се спрем по-подробно на конструкторите след малко. Понеже член-променливите **name** и **color** на класа Cat са от референтен тип (от класа **String**), те се записват също в динамичната памет (heap) и в самия обект стоят техните референции (адреси).

Освобождаване на обектите

Важна особеност на работата с обекти в Java е, че обикновено няма нужда от ръчното им разрушаване и освобождаване на паметта, заета от тях. Това е възможно поради наличието на **garbage collector** във виртуалната машина, който се грижи за това вместо нас. Обектите, към които в даден момент вече няма референция в програмата автоматично се унищожават и паметта, която заемат се освобождава. По този начин се предотвратяват много потенциални бъгове и проблеми. Ако искаме ръчно да освободим даден обект, трябва да унищожим референция към него, например така:

```
myBrownCat = null;
```

Това не унищожава обекта веднага, но го оставя в състояние, в което той е недостъпен от програмата и при следващото включване на системата за почистване на паметта (garbage collector) той ще бъде освободен.

Достъп до полета на обекта

Достъпът до полетата и свойствата (properties) на даден обект става чрез оператора **.** (точка), поставен между името на обекта и името на полето (или свойството). Операторът **.** не е необходим в случай, че достъпваме поле или свойство на даден клас в тялото на метод на същия клас.

Можем да достъпваме полетата и свойствата или с цел да извлечем данните от тях, или с цел да запишем нови данни. В случай на свойство, достъпът се

реализира чрез два специални метода, наречени **getter** и **setter**. Те извършват съответно извличането на стойността на свойството и присвояването на нова стойност. В дефиницията на класа Cat (която дадохме по-горе) такива методи са **getName()** и **setName(…)**.

Достъп до полета на обекта – пример

Ще дадем прост пример за употребата на свойство на обект, като използваме вече дефинирания по-горе клас **Cat**. Създаваме инстанция **myCat** на класа **Cat** и присвояваме стойност **"Alfred"** на свойството **name.** След това извеждаме на стандартния изход форматиран низ с името на нашата котка. Следва реализацията на примера:

```
publicclass CatManipulating {
	publicstaticvoid main(String[] args) {
		Cat myCat = new Cat();
		myCat.name = "Alfred";

		System.out.println("The name of my cat is %s.",myCat.name);
  }
}
```

Извикване на методи на обект

Извикването на методите на даден обект става отново чрез оператора **.** (точка). Операторът точка не е нужен единствено, в случай че съответният метод се извиква в тялото на друг метод на същия клас.

Тук е моментът да споменем факта, че методите на класовете имат **модификатори за достъп public**, **private** или**protected**, чрез които възможността за извикването им може да се ограничава.

Извикване на методи на обект – пример

Ще допълним примера, който вече дадохме като извикаме метода **sayMiau** на класа **Cat**. Ето какво се получава:

```
publicclass CatManipulating {
```

```
        publicstaticvoid main(String[] args) {
                Cat myCat = new Cat();
                myCat.name = "Alfred";

                System.out.println("The name of my cat is %s.%n",
      myCat.name);
                myCat.sayMiau();
  }
}
```

След изпълнението на горната програма на стандартния изход ще бъде изведен следния текст:

```
The name of my cat is Alfred.
Cat Alfred said: Miauuuuuu!
```

Конструктори

Конструкторът е специален метод на класа, който се извиква автоматично при създаването на обект от този клас и извършва инициализация на данните му (това е неговото основно предназначение). Конструкторът няма тип на връщана стойност и неговото име не е произволно, а задължително съвпада с името на класа. Конструкторът може да бъде със или без параметри. Конструктор без параметри наричаме още конструктор по подразбиране (default constructor).

Конструктори с параметри

Конструкторът може да имат параметри, както всеки друг метод. Всеки клас може да има произволен брой конструктори с единственото ограничение, че броят и типът на параметрите им трябва да бъде различен. При създаването на обект от този клас се извиква точно един от дефинираните конструктори.

При наличието на няколко конструктора в един клас естествено възниква въпросът кой от тях се извиква при създаването на обект. Този проблем се решава по много интуитивен начин. Подходящият конструктор се избира автоматично в зависимост от подадените параметри при създаването на обекта. Използва се принципът на най-добро съвпадение.

Извикване на конструктори – пример

Да разгледаме отново дефиницията на класа **Cat** и по-конкретно двата конструктора на класа:

```
publicclass Cat {
	private String name;
	private String color;
	...
	public Cat() {
		this.name = "Unnamed";
		this.color = "gray";
	}
	public Cat(String name, String color) {
		this.name = name;
		this.color = color;
	}
	...
}
```

Ще използваме тези конструктори, за да илюстрираме употребата на конструктор без и с параметри. При така дефинирания клас **Cat** ще дадем пример за създаването на негови инстанции чрез всеки от двата конструктора. Единият обект ще бъде обикновена неопределена котка, а другият – нашата кафява котка Johnny. След това ще изпълним метода **sayMiau** на всяка от двете и ще разгледаме резултата**.** Следва изходният код**:**

```
publicclass CatManipulating {
	publicstaticvoid main(String[] args) {
		Cat someCat = new Cat();

		someCat.sayMiau();
		System.out.println("The color of cat %s is %s.%n",
			someCat.name, someCat.color);

		Cat myBrownCat = new Cat("Johnny", "brown");

		myBrownCat.sayMiau();
		System.out.println("The color of cat %s is %s.%n",
			myBrownCat.name, myBrownCat.color);
}
```

В резултат от изпълнението на програмата се извежда следният текст на стандартния изход:

```
Cat Unnamed said: Miauuuuuu!
The color of cat Unnamed is gray.
Cat Johnny said: Miauuuuuu!
The color of cat Johnny is brown.
```

Статични полета и методи

Член-данните, които разглеждахме досега реализират състояния на обектите и са пряко свързани с конкретни инстанции на класовете. В ООП има специална категория полета и методи, които се асоциират с тип данни (клас), а не с конкретна инстанция (обект). Наричаме ги **статични членове** (static members), защото са независими от конкретните обекти. Нещо повече – те се използват, без да има създадена инстанция на класа, в който са дефинирани. Ще разгледаме накратко статичните членове в Java – това могат да бъдат полета, методи и конструктори.

Статично поле или метод се дефинира чрез ключовата дума **static**, поставена преди типа на полето или типа на връщаната стойност на метода. При дефинирането на статичен конструктор думата **static** се поставя преди името на конструктора. Статичните конструктори не са предмет на настоящата тема – засега ще се спрем на статичните полета и методи.

Статични полета и методи – пример

Примерът, който ще дадем решава следната проста задача: нужен ни е метод, който всеки път връща стойност с едно по-голяма от стойността, върната при предишното извикване на метода. Избираме първата върната от метода стойност да бъде 0. Очевидно такъв метод генерира редицата на естествените числа. Подобна функционалност има широко приложение в практиката – за еднозначно номериране на обекти. Сега ще видим как може да се реализира с инструментите на ООП.

Да приемем, че методът е наречен **nextValue()** и е дефиниран в клас с име **Sequence**. Класът има поле **currentValue** от тип **int**, което съдържа последно върнатата стойност от метода. Искаме в тялото на метода да се извършват последователно следните две действия: да се увеличава стойността на полето и да се връща като резултат новата му стойност. Връщаната от метода стойност очевидно не зависи от конкретна инстанция на класа **Sequence**. Поради тази причина методът и полето са статични. Следва описаната реализация на класа:

```
publicclass Sequence {

	privatestaticintcurrentValue = -1;
	private Sequence() {
	}
	publicstaticint nextValue() {
		currentValue++;
		returncurrentValue;
	}
}
```

Наблюдателният читател е забелязал, че така дефинираният клас има конструктор по подразбиране, който е деклариран като **private**. Тази употреба на конструктор може да изглежда особена, но е съвсем умишлена. Добре е да знаем следното:

	Клас, който има само private конструктори не може да бъде инстанциран. Такъв клас обикновено има само статични членове и се нарича utility клас.

Нека сега видим една проста програма, която използва класа **Sequence**:

```
publicclass SequenceManipulating {
	publicstaticvoid main(String[] args) {
		System.out.printf("Sequence[1..3]: %d, %d, %d%n",
			Sequence.nextValue(), Sequence.nextValue(),
			Sequence.nextValue());
}
```

Извеждаме на стандартния изход първите три естествени числа чрез последователно извикване на метода **nextValue() на класа Sequence.** Резултатът от този код е следният:

```
Sequence[1..3]: 0, 1, 2
```

Примери за системни Java класове

След като вече се запознахмес основната функционалност на обектите, ще разгледаме накратко няколко често използвани системни класа от стандартните библиотеки на Java. По този начин ще видим на практика казаното дотук, а също ще покажем как системните класове улесняват работата ни.

Класът System

Започваме с един от основните системни класове в Java. Той съдържа набор от полезни полета и методи, улесняващи взаимодействието на програмите с операционната система. Ето част от функционалността, която предоставя този клас:

- Стандартните входно-изходни потоци **System.out**, **System.in** и **System.err** (които вече сме разглеждали).
- Достъп до външно дефинирани свойства (properties) и променливи на обкръжението (environment variables), които няма да разглеждаме в настоящата книга.
- Средства за зареждане на файлове и библиотеки.

Класът String

Вече сме споменавали класа в Java, който представя символни низове (последо-вателности от символи). Да припомним, че можем да считаме низовете за примитивен тип данни в Java, въпреки че работата с тях се различава до известна степен от работата с другите примитивни типове (цели и реални числа, булеви променливи и др.).

Класът Math

Съдържа методи за извършването на основни числови операции като повдигане в степен, логаритмуване, коренуване и тригонометрични функции. Ще дадем един прост пример, който илюстрира употребата му.

Съставяме програма, която пресмята лицето на триъгълник по дадени дължини на две от страните и ъгъла между тях в градуси. За тази цел имаме нужда от методите **sin(…)** и **toRadians(…)** на класа **Math**. Следва примерна реализация:

```
publicclass MathTest {
	publicstaticvoid main(String[] args) {
		java.util.Scanner input = new java.util.Scanner(System.in);

		System.out.println("Length of the first side:");
		double a = input.nextDouble();
		System.out.println("Length of the second side:");
		double b = input.nextDouble();
		System.out.println("Size of the angle in degrees:");
		int angle = input.nextInt();

		System.out.printf("Face of the triangle: %f%n",
			0.5 * a * b * Math.sin(Math.toRadians(angle)));
	}
}
```

Можем лесно да тестваме програмата като проверим дали пресмята правилно лицето на равностранен триъгълник. За допълнително улеснение избираме дължина на страната да бъде 2 – тогава лицето му намираме с добре известната формула:

$$S = \frac{\sqrt{3}}{4}2^2 = \sqrt{3} = 1{,}7320508...$$

Въвеждаме последователно числата 2, 2, 60 и на стандартния изход се извежда:

```
Face of the triangle: 1,732051
```

Класът Math – още примери

Освен математически методи класът **Math** дефинира и две добре известни в математиката константи: числото π и Неперовото число $\boldsymbol{e}$.Ето как се достъпват те:

```
System.out.println(Math.PI);
System.out.println(Math.E);
```

При изпълнение на горния код се получава следния резултат:

```
3.141592653589793
2.718281828459045
```

Класът Random

Понякога в програмирането се налага да използваме случайни числа. Например искаме да генерираме 6 случайни числа в интервала между 1 и 49 (ТОТО 6/49). Това можем да направим използвайки класа **java.util. Random** и неговия метод **nextInt**(). Преди използваме класа **Random** трябва да създадем негова инстанция, при което тя се инициализира със случайна стойност (извлечена от текущото системно време в операционната система). След това можем да генерира случайно число в интервала **[0…n)** чрез извикване на метода **nextInt(n)**. Забележете, че този метод може да върне нула, но връща винаги случайно число по-малко от зададената стойност **n**. Затова ако искаме да получим число в интервала **[0…49]**, трябва използваме израза **nextInt(49)+1**. Ето сорс кода на една програма, която използвайки класа **Random** генерира 6 случайни числа от ТОТО 6/49:

```
import java.util.Random;

publicclass TOTO649 {
        publicstaticvoid main(String[] args) {
                Random rand = new Random();
                for (int number=1; number<=6; number++) {
                        int randomNumber = rand.nextInt(49) + 1;
                        System.out.printf("%d ", randomNumber);

                }

        }

}
```

Ето как изглежда един възможен изход от работата на програмата:

```
14 49 7 16 29 2
```

Константите в Java представляват неизменими променливи, чиито стойности се задават по време на инициализацията им в сорс кода на програмата и след това не могат да бъдат променяни. Те се декларират с модификаторите **static** и **final**. Използват се за дефиниране на дадено число или стринг, което се използва след това многократно в програмата. По този начин се спестяват повторенията на определени стойности в сорс кода и се позволява лесно тези стойности да се променят като се бута само на едно място в сорс кода. Например ако в даден момент решим, че символът **","** (запетая) не трябва да се ползва при генерирането на пароли, можем да променим само 1 ред в програмата (съответната константа) и промяната ще се отрази навсякъде, където е използвана съответната константа. Константите в Java се изписват само с главни букви, като за разделител между думите се ползва символът "_" (долна черта).

Упражнения

1. Напишете програма, която извежда на стандартния изход броя на дните, часовете и минутите, които са изтекли от 1 януари 1970 година до момента на изпълнението на програмата. За реализацията използвайте класа **System**.

2. Напишете програма, която по дадени два катета намира хипотенузата на правоъгълен триъгълник. Реализирайте въвеждане на дължините на катетите от стандартния вход, а за пресмятането на хипотенузата използвайте методи на класа **Math**.

Упътвания

1. Използвайте метода **System.currentTimeMillis()**,за да получите броя на изтеклите милисекунди. Използвайте факта, че в една секунда има 1000 милисекунди и пресметнете минутите, часовете и дните.
2. Хипотенузата на правоъгълен триъгълник се намира с помощта на известната теорема на Питагор: $\mathbf{a}^2 + \mathbf{b}^2 = \mathbf{c}^2$, където **a** и **b** са двата катета, а **c** е хипотенузата. Коренувайте двете страни, за да получите формула за дължината на хипотенузата. За реализацията на коренуването използвайте метода **sqrt(…)** на класа **Math**.

3. Обработка на изключения

Какво е изключение?

Докато програмираме ние описваме постъпково какво трябва да направи компютъра и в повечето случаи разчитаме на нормалното изпълнение на програмата. В повече от 99% от времето програмите следват този нормален ход на изпълнение, но съществуват и изключения от това правило. Да речем, че искаме да прочетем файл и да покажем съдържанието му на екрана. Името на файла се подава от потребителя. По невнимание потребителя въвежда име на файл, който не съществува. Програмата няма да може да се изпълни нормално и да покаже съдържанието на файла на екрана. В този случай имаме изключение от правилното изпълнение на програмата и за него трябва да се сигнализира на потребителя и/или администратора.

Изключение

Изключение (exception) в общия случай е уведомление за дадено събитие, нарушаващо нормалната работа на една програма. Изключенията дават възможност това необичайно събитие да бъде обработено и програмата да реагира по някакъв начин. Когато възникне изключение конкретното състояние на програмата се запазва и се търси **обработчик на изключението (exception handler)**.

Изключенията се предизвикват или **"хвърлят" (throw an exception)**.

Прихващане и обработка на изключения

Exception handling (**инфраструктура за обработка на изключенията**) е част от средата – механизъм, който позволява хвърлянето и прихващането на изключения. Част от тази инфраструктура са дефинираните езиковите конструкции за хвърляне и прихващане на изключения. Тя се грижи и затова изключението да стигне до кода, който може да го обработи.

Изключенията в Java

Изключение (exception) в Java представлява събитие, което уведомява програмиста, че е възникнало обстоятелство (грешка) непредвидено в нормалния ход на програмата. Това става като методът, в който е възникнала грешката изхвърля специален обект съдържащ информация за вида на грешката, мястото в програмата, където е възникнала, и състоянието на програмата в момента на възникване на грешката.

Всяко изключение в Java носи т.нар **stack trace** (няма да се мъчим да го превеждаме) – информация за това къде точно в кода е възникнала грешката.

Пример за код, който хвърля изключения

Типичен пример за код, който хвърля изключения е следният метод:

```
publicstaticvoid readFile(String fileName) {
	FileInputStream fis = new FileInputStream(fileName);
	BufferedReader in = new BufferedReader(
		new InputStreamReader(fis));
	String tmp = null;
	while ((tmp = in.readLine()) != null) {
		System.out.println(tmp);
	}
	in.close();
	fis.close();
}
```

Това е код, който се опитва да отвори текстов файл и да чете от негоред по ред докато файлът свърши.

Подчертаните методи и конструктори са тези, в които се хвърлят изключенията. В примера конструкторът **FileInputStream(fileName)** хвърля **FileNotFoundException**, ако не съществува файл с име, каквото му се подава. Методите на потоците **readLine()** и **close()**, хвърлят **IOException**ако възникне неочакван проблем при входно-изходните операции.Този пример няма да се компилира (местата, където са грешките от компилация, са подчертани), защото хвърляните изключения трябва да бъдат прихванати и да бъдат подходящо обработени.

Как работят изключенията?

Ако по време на нормалния ход на програмата някой от извикваните методи неочаквано хвърли изключение, то нормалният ход на програмата се преустановява. Това ще се случи, ако например възникне изключение от типа **FileNotFoundException** при инициализиране на файловия поток от горния пример. Нека разгледаме сления ред:

```
FileInputStream fis = new FileInputStream(fileName);
```

Ако се случи изключение, променливата **fis** няма да бъде инициализирана и ще остане със стойност **null**. Нито един от следващите редове от метода няма да бъде изпълнен. Програмата ще преустанови своя ход докато виртуалната машина не намери обработчик на възникналото изключение **FileNotFoundException**.

Прихващане на изключения в Java

След като един метод хвърли изключение, виртуалната машина търси код, който да го прихване и евентуално обработи. За да разберем как действа този механизъм ще разгледаме понятието **стек** на извикване на методите. Това е същият този стек, в който се записват всички променливи в програмата, параметрите на методите и стойностните типове.

Всяка програма на Java започва с **main()** метод. В него може да се извика друг метод да го наречем "Метод 1", който от своя страна извиква "Метод 2" и т.н., докато се извика "Метод N".

Когато "Метод N" свърши работата си управлението на програмата се връща на предходния и т. н., докато се стигне до **main()** метода. След като се излезе от него свършва и програмата. Като се извиква нов метод той се добавя най-отгоре в стека, а като свърши изпълнението му метода се изважда от стека.

Програмна конструкция try-catch

За да прихванем изключение обгръщаме парчето код, където може да възникне изключение с програмната конструкция **try-catch**:

```
try {
        Some sode that may throw and exception
} catch (ExceptionType objectName) {
} catch (ExceptionType objectName) {
}
```

Конструкцията се състои от един **try** блок, обгръщащ валидни Java конструкции, които могат да хвърлят изключения, следван от един или много **catch** блока, които обработват различни по тип изключения. В **catch** блокът **ExceptionType** трябва да е тип на клас, който е наследник на класа **java.lang.Throwable**. В противен случай ще получим проблем при компилация. Изразът в скобите след **catch** играе роля на декларация на променлива и затова вътре в блока **catch** можем да използваме **objectName**, за да извикваме методите или да използваме свойствата на изключението.

Прихващане на изключения – пример

Да направим така, че горният пример да се компилира. Заграждаме целият проблемен код, където могат да се хвърлят изключения с **try-catch** блок и добавяме прихващане на двата вида изключения:

```
publicstaticvoidreadFile(String fileName) {
        try {
                FileInputStream fis = new FileInputStream(fileName);
                BufferedReader in = new BufferedReader(
                        new InputStreamReader(fis));
                String tmp = null;
                while ((tmp = in.readLine()) != null) {
                        System.out.println(tmp);
                }
                in.close();
                fis.close();
        } catch (FileNotFoundException e) {
                System.out.println("The file \"" + fileName +
                        "\" does not exist! Unable to read it.");
        } catch (IOException e) {
                e.printStackTrace();
        }
}
```

Добре, сега методът се компилира, но работи по малко по-различен начин. При възникване на **FileNotFoundException** по време на изпълнението на **new**

FileInputStream(fileName) виртуалната машина няма да изпълни следващите редове, а ще се прескочи чак на реда, където изключението е прихванато с **catch (FileNotFoundException e)** и ще се изпълни блока след него:

```
catch (FileNotFoundException e) {
        System.out.println("The file \"" + fileName +
                "\" does not exist! Unable to read it.");
}
```

Като обработка на изключението просто потребителите ще бъдат информирани, че такъв файл не съществува. Това се извършва чрез съобщение, изведено на стандартния изход.

Stack Trace

Информацията, която носи т. нар. **Stack trace**, съдържа подробно описание на естеството на изключението и за мястото в програмата, където то е възникнало. Stack trace се използва, за да се намерят причините за възникването на изключението и последващото им отстраняване (довеждане до нормалното изпълнение на програмата). Stack trace съдържа голямо количество информация и е предназначен за анализиране само от програмистите и администраторите, но не и от крайните потребители на програмата, които не са длъжни да са технически лица. Stack trace е стандартно средство за търсене и отстраняване (дебъгване) на проблеми.

Stack Trace – пример

Ето как изглежда stack trace на изключение за липсващ файл от примера по-горе. Подали сме несъществуващ файл **C:\missingFile.txt** и вместо да изведем съобщението сме използвали метода **e.printStackTrace()**.

```
java.io.FileNotFoundException: C:\missingFile.txt (The system cannot
find the file specified)
        at java.io.FileInputStream.open(Native Method)
        at java.io.FileInputStream.<init>(Unknown Source)
        at java.io.FileInputStream.<init>(Unknown Source)
        at ReadFile.readFile(ReadFile.java:12)
        at ReadFile.main(ReadFile.java:35)
```

Системата не може да намери този файл и затова хвърля изключението **FileNotFoundException**.

Хвърляне на изключения (конструкцията throw)

Изключения се хвърлят с ключовата дума **throw**, като първо се създава инстанция на изключението и се попълва нужната информация за него. Могат да се хвърлят само класове наследници на **java.lang.Throwable**.

Ето един пример:

```
publicstaticvoid main(String... args) {
        RuntimeException exception =new RuntimeException("Problem");
        throw exception;
}
```

Резултатът от изпълнението на програмата е следният:

```
Exception in thread "main" java.lang.RuntimeException: Problem
        at introjavabook.Program.main(Program.java:10)
```

Видове изключения в Java

В Javaима 3 вида изключения: проверени (checked), непроверени (unchecked) и грешки (errors). Когато ги разглеждаме, ще използваме най-вече оригиналните английски термини, защото те са трудно преводими.

Принципът "хвани или изхвърли"

Принципът "хвани или изхвърли" важи за изключенията, които задължително трябва да се обработят. Те или трябва да бъдат прихванати, или изхвърлени от метода, в който са възникнали, или от някой от следващите методи от стека на извикванията.

За изключенията, които **трябва** да бъдат обработени, има два варианта:

- Изключението да бъде обработено **(хвани)**:

```
publicstaticvoid openFile(String fileName) {
	try {
		FileInputStream fis = new FileInputStream(fileName);
		// ...
	} catch (FileNotFoundException e) {
		// ...
	}
}
```

- Отговорността за изключението да бъде оставена на друг **(изхвърли)**, като той бъде задължен да обработи това изключение или да задължи някой друг:

```
publicstaticvoid openFile(String fileName)
		throws FileNotFoundException {
	FileInputStream fis = new FileInputStream(fileName);
}
```

Изключението или се обработва на място или се обявява като изхвърляно от метода, в който възниква (или хвани или изхвърли). По този начин методът прехвърля отговорността за обработката на даден тип изключения на извикващия метод.

Checked exceptions

Checked (проверени) са изключения, които **задължителнотрябва** да спазват принципа "хвани или изхвърли" и това се гарантира от компилатора. Тези изключения наследяват класа **java.lang.Exception**, но не наследяват **java.lang.RuntimeException**.

Checked са изключения, които една добре написана програма трябва да очаква и би трябвало да може да се възстанови от тях.

Например програма, която чете данни от сървър с бази от данни. Ако кабелът до сървъра в дадения момент бъде прекъснат, програмата ще получи **ConnectException** и може да съобщи на потребителя да опита отново или да му обясни, че в момента може да използва само други части на програмата.

Checked изключенията или трябва да бъдат прихванати и обработени или трябва да бъдат изхвърляни изрично чрез **throws** декларация в съответния метод. Ако нито едно от двете не е направено, компилаторът ще даде съобщение за грешка.

Грешки (Errors)

Грешките (errors) са критични ситуации (fatal errors), при които изпълнението на програмата обикновено не може да се възстанови и трябва принудително да завърши. Пример за такава грешка е **java.lang. OutOfMemoryError**. Когато паметта свърши, програмата обикновено няма какво да направи и трябва принудително да запише състоянието си (за да няма загуба на данни) и да завърши.

Грешките не спазват принципа "хвани или изхвърли". Не се очаква да ги обработваме, въпреки че е възможно.

Unchecked exceptions

Unchecked (непроверени) изключения, са изключения, които не са задължени да спазват принципа "хвани или изхвърли". Тези изключения наследяват класа **RuntimeException**. Възникването на такова изключение най-често означава бъг в програмата или неправилна употреба на някоя библиотека.

Вероятно сте се сблъсквали с грешката **NullPointerException**. Тя е типичен представител на unchecked изключенията. Може да възникне по невнимание, когато се обърнем към обект, който няма стойност. Прихващането и обработването на такива проблеми не е задължително, но е възможно.

Конструкцията try-finally

Всеки блок **try** може да съдържа блок **finally**. Блока **finally** се изпълнява **винаги** при излизане от **try** блока, независимо как се излиза от **try** блока. Това гарантира

изпълнението на **finally** блока дори ако възникне неочаквано изключение или се излезе с израз **return**.

	Блокът finally няма да се изпълни, ако по време на изпълнението на блока try виртуалната машина прекрати изпълнението си!

Блокът **finally** има следната основна форма:

```
try {
        Some code that could or could not cause an exception
} finally {
}
```

Всеки **try** блок може да има един единствен блок **finally** освен блоковете **catch**. Възможна е и комбинация с множество **catch** блокове и един **finally** блок.

```
try {
        some code
} catch (...) {
} catch (...) {
} finally {
}
```

В случай на нужда от освобождаване на вече заети ресурси блока **finally** е незаменим. Ако го нямаше, никога не бихме били сигурни дали разчистването няма случайно да се прескочи при неочаквано изключение или заради използването на **return**, **continue**, или **break** изрази.

Предимства при използване на изключения

След като се запознахме подробно с изключенията, техните свойства и с това как да работим с тях, нека разгледаме причините те да бъдат въведени и да придобият широко разпространение.

Отделяне на кода за обработка на грешките

Използването на изключения позволява да се отдели кода описващ нормалното протичане на една програма от кода необходим в изключителни ситуации и кода необходим при обработване на грешки. Това ще демонстрираме със следния пример, които е псевдокод на примера разгледан от началото на главата.

```
readFile() {
	openFileInputStream();
	while (fileHasMoreLines) {
		readNextLine();
		printTheLine();
	}
	closeTheFile();
}
```

Нека сега преведем последователността от действия на български:

- Отваряме файл;
- Четем следващ ред от файла до края;
- Изписваме прочетения ред;
- Затваряме файла;

Методът е добре написан, но ако се вгледаме по-внимателно започват да възникват въпроси:

- Какво ще стане, ако няма такъв файл?
- Какво ще стане, ако файлът не може да се отвори (например, ако друг процес вече го е отворил за писане)?
- Какво ще стане, ако пропадне четенето на някой ред?
- Какво ще стане, ако файла не може да се затвори?

Да допишем метода, така че да взима под внимание тези въпроси, без да използваме изключения, а да използваме кодове за грешка връщани от всеки използван метод (кодовете за грешка са стандартен похват за обработка на грешките в процедурно ориентираното програмиране. Всеки метод връща **int**, който определя дали методът е изпълнен правилно. Код за грешка 0 означава, че всичко е правилно, код различен от 0 означава различен тип грешка).

```
errorCode readFile() {
	errorCode = 0;
	openFileErrorCode = openFileInputStream();
```

```
	if (openFileErrorCode = 0) {
		while (fileHasMoreLines) {
			readLineErrorCode = readNextLine();
			if (readLineErrorCode == 0) {
				printTheLine();
			} else {
				errorCode = -1;
				break;
			}
		}
		closeFileErrorCode = closeTheFile();
		if (closeFileErrorCode != 0 && errorCode == 0) {
			errorCode = -2;
		} else {
			errorCode = -3;
		}
	} elseif (openFileErrorCode = -1) {
		errorCode = -4;
	} elseif (openFileErrorCode = -2) {
		errorCode = -5;
	}
	return errorCode;
}
```

Получава се един доста замотан, трудно разбираем и лесно объркващ – "спагети" код. Логиката на програмата е силно смесена с логиката за обработка на грешките и непредвидените ситуации. По-голяма част от кода е тази за правилна обработка на грешките.

Всички тези нежелателни последици се избягват при използването на изключения. Ето колко по-прост и чист е псевдокода на същия метод, само че с изключения:

```
readFile() {
	try {
		openFileInputStream();
		while (fileHasMoreLines) {
			readNextLine();
			printTheLine();
		}
		closeTheFile();
	} catch (FileNotFoundException) {
		doSomething;
```

```
        } catch (IOException) {
                doSomething;
        }
}
```

Всъщност изключенията не ни спестяват усилията при намиране и обработка на грешките, но ни позволяват да правим това по далеч по-елегантен, и ефективен начин.

4. Символни низове

В практиката често се налага обработката на текст: четене на текстови файлове, търсене на ключови думи и заместването им в даден параграф, валидиране на входни потребителски данни и др. В такива случаи можем да запишем текстовото съдържание, с което ще боравим, в символни низове, и да го обработим с помощта на езика Java.

Какво е символен низ (стринг)?

Символният низ е последователност от символи, записана на даден адрес в паметта. Помните ли типа **char**? В променливите от тип **char** можем да запишем само 1 символ. Когато е необходимо да обработваме повече от един символ, на помощ идват стринговете.

В Java всеки символ има пореден номер в **Unicode** таблицата. Unicode е стандарт, създаден в края на 80-те и началото на 90-те години с цел съхраняването на различни типове текстови данни. Предшественикът му ASCII позволява записването на едва 128 или 256 символа (ASCII стандарт със 7-битова или 8-битова таблица). За съжаление, това често не удовлетворява нуждите на потребителя – тъй като в 128 символа могат да се поберат само цифри, малки и главни латински букви и някои специални знаци. Когато опре до работа с текст на кирилица или друг специфичен език (например азиатски или африкански), 128 символа са напълно недостатъчни. Ето защо Java използва 16-битова кодова таблица за символи. С помощта на знанията ни за бройните системи и представянето на информацията в компютрите, можем да сметнем, че кодовата таблица съхранява 2^16 = 65536 символа. Някои от символите се кодират по специфичен начин, така че е възможно използването на 2 символа от Unicode таблицата за създаване на нов символ – така получените знаци надхвърлят 100 000.

Класът java.lang.String

Класът **java.lang.String** позволява обработка на символни низове в Java. Работата със **String** ни улеснява при манипулацията на текстови данни: построяване на текстове, търсене в текст и много други операции. Пример за декларация на символен низ:

```
String greeting = "Hello, Java";
```

Декларирахме променливата **greeting** от тип **String**, която има съдържание "**Hello, Java**". Представянето на съдържанието в символния низ изглежда по подобен начин:

H	e	l	l	o	,		J	a	v	a

Вътрешното представяне на класа е съвсем просто – масив от символи. По принцип ние можем да избегнем използването на класа, като декларираме променлива от тип **char[]** и запълним елементите на масива символ по символ. Недостатъците на това обаче са няколко:

1. Запълването на масива става символ по символ, а не наведнъж.
2. Трябва да знаем колко дълъг ще е текстът, за да сме наясно дали ще се побере в заделеното място за масива.
3. Обработката на текстовото съдържание става ръчно.

Класът String – универсално решение?

Използването на **String** не е идеално и универсално решение – понякога е уместно използването на други символни структури.

В Java съществуват и други класове за обработка на текст – с някои от тях ще се запознаем по-нататък в главата.

Класът **String** има важна особеност – последователностите от символи, записани в променлива от класа, са **неизменими (immutable)**. Веднъж записано, съдържанието на променливата не се променя директно - ако опитаме да

променим стойността, тя ще бъде записана на ново място в динамичната памет, а променливата ще започне да сочи към него.

Типът **String** е по-особен от останалите типове данни. Той е клас и спазва принципите на обектно-ориентираното програмиране: стойностите се записват в динамичната памет, а променливите пазят препратка към паметта (референция към обект в динамичната памет). От друга страна, **String** променливите са неизменими. Ако няколко променливи сочат към една и съща област в паметта с дадена стойност, тази стойност не може да бъде директно променена. Промяната ще се отрази само на променливата, чрез която е редактирана стойността, тъй като това ще създаде нова стойност в динамичната памет и ще насочи въпросната променлива към нея, докато останалите променливи ще сочат на старото място.

Символни низове – прост пример

Използването на променливи от тип **String** изглежда по следния начин:

```
String msg = "Stand up, stand up, Balkan superman.";


System.out.printf("msg = \"%s\"%n", msg);
System.out.printf("msg.length() = %d%n", msg.length());


for (int i = 0; i < msg.length(); i++) {
	System.out.printf("msg[%d] = %c%n", i, msg.charAt(i));
}
```

В посочения фрагмент от код виждаме декларация на променливата **s** и задаването на стойност:

```
Stand up, stand up, Balkan superman.
```

Обърнете внимание на стойността на стринга – кавичките не са част от текста, а ограждат стойността му.

Ето как изглежда резултатът от изпълнението на горния пример (със съкращения):

```
msg = "Stand up, stand up, Balkan superman."
msg.length() = 36
msg[0] = S
msg[1] = t
msg[2] = a
msg[3] = n
msg[4] = d
...
```

Деклариране на символен низ

Можем да декларираме променливи от тип символен низ чрез класа **java.lang.String**:

```
String str;
```

Декларацията на символен низ представлява декларация на променлива от класа **String**. Това не е еквивалентно на създаването на променлива и заделянето на памет за нея! С декларацията уведомяваме компилатора, че ще използваме променлива **str** и очакваният тип за нея е **String**. Ние не създаваме променливата в паметта и тя все още не е достъпна за обработки (има стойност **null**, което означава липса на стойност).

Сравняване на низове по азбучен ред

Има множество начини за сравнение на символни низове. В зависимост от това какво точно ни е необходимо в конкретния случай, може да се възползваме от различни възможности на класа **String**.

Сравнение за еднаквост

Ако условието изисква да сравним два символни низа и да установим дали стойностите им са еднакви или не, удобни методи са **equals(…)** и **equalsIgnoreCase(…)**. Двата метода връщат булев резултат със стойност **true**, ако низовете имат еднакви стойности, и **false**, ако те са различни. Първата функция проверява за равенство на стойностите на променливите, като прави разлика между малки и главни букви. Т.е. сравняването на "Java" и "JAVA" с метода **equals(…)** ще върне стойност **false**. В практиката често ще ни интересува самото съдържание, без значение от регистъра (**casing**) на буквите. Използването на метода **equalsIgnoreCase(…)** в горния пример би игнорирал разликата между малки и главни букви и ще върне стойност **true**.

```
String word1 = "Java";
String word2 = "JAVA";
System.out.println(word1.equals(word2)); // false
System.out.println(word1.equalsIgnoreCase(word2)); // true
```

Сравнение на низове по азбучен ред

Дотук добре, но как ще установим лексикографската подредба на няколко низа? Ако искаме да сравним две думи и да получим данни коя от тях е преди другата, според азбучния ред на буквите в нея, на помощ идват **compareTo(…)** и **compareToIgnoreCase(…)**. Двата метода ни дават възможност да сравним стойностите на два символни низа, като установим лексикографския им ред.

Връщайки се на темата за кодировката на символите, си припомняме, че всеки символ има свой уникален номер в Unicode таблицата. Например главната латинска буква "B" има стойност 66, докато главната "E" – 69. Методът **compareTo(…)** сравнява 2 символни низа за равенство или различие. За да бъдат два низа с еднакви стойности, то те трябва да имат една и съща дължина (брой символи) и символите да бъдат еднакви и подредени в един и същ ред. Низовете "give" и "given" са различни, защото имат различна дължина. "near" и "fear" се

различават по първия си символ, а "stop" и "post" имат едни и същи символи и дължина, но в различен ред - което отново ги прави различни.

Обобщавайки поведението на метода, можем да приемем, че **compareTo(…)** връща положително число, отрицателно число или 0 в зависимост от лексикографската подредба.

За да не бъдем голословни, нека разгледаме няколко примера:

```
String score = "sCore";
String scary = "scary";
System.out.println(score.compareToIgnoreCase(scary)); // 14
System.out.println(scary.compareToIgnoreCase(score)); // -14
System.out.println(scary.compareTo(score)); // 32
```

За примера ще използваме променливи със стойности "sCore" и "scary". Първият експеримент е извикването на метода **compareToIgnoreCase(…)** на низа **score**, като подаден параметър е променливата **scary**. Тъй като методът игнорира регистъра за малки и главни букви, първите 2 символа и от двата низа връщат знак за равенство. Различието се открива едва в третия символ, който в първия низ е "o", а във втория: "a". Тогава изваждаме кода на параметъра от кода на променливата, за която е извикан методът. Крайният резултат е 14 (кодът на 'o' е 111, кодът на 'a' е 97; 111-97 = 14). Извикването на същия метод с разменени места на променливите връща - 14, защото тогава отправната точка е низът **scary** и кодовете се изваждат в обратен ред.

Последният тест е с метода **compareTo(…)** – тъй като той прави разлика между главни и малки букви, разлика откриваме още във втория символ на двата низа. В променливата **scary** символът "c" има код 99, в **score** главно ю "C" е 67 и връщаният резултат е 32.

Защо операторите == и != не работят за низове?

По-любознателните от вас може вече да са се запитали защо операторите за равенство и различие не работят при работа със символни низове? Причината е

тяхната логика в света на обектно-ориентираното програмиране. Когато работим със стойностни типове (цели числа, символи, числа с плаваща запетая), тогава операторите сравняват стойностите на променливите. Тъй като символните низове в Java са реализирани с класове, тук влизат правилата за сравняване на препратки към паметта, известни още като референции или указатели. Тогава сравняването на две променливи **str1** и **str2** няма да сравнява техните стойности, а дали те сочат към една и съща област в динамичната памет.

Търсене на низ в друг низ

Когато имаме символен низ със зададено съдържание, често се налага да обработим само част от стойността му. За да автоматизираме процеса, можем да претърсваме даден стринг за определени ключови думи.

Java платформата ни предоставя 2 метода за търсене на низове: **indexOf(…)** и **lastIndexOf(…)**. Те претърсват даден символен низ и проверяват дали подаденият като параметър подниз се среща в съдържанието му. Връщаният резултат на методите е цяло число. Ако резултатът е неотрицателна стойност, тогава това е позицията, на която е открит първият символ от подниза. Ако методът върне стойност **-1**, това означава, че поднизът не е открит. Напомняме, че в Java индексите на символите в низовете започват от 0.

Търсене в символен низ – пример

Ето и един пример за използване на метода **indexOf(…)**:

```
String book = "Introduction to Java book";
int index = book.indexOf("Java");
System.out.println(index); // index = 16
```

В примера променливата **book** има стойност "Introduction to Java book". Търсенето на подниза "Java" в горната променлива ще върне стойност 16, защото поднизът е открит в стойността на отправната променлива и първият символ "**J**" от търсената дума се намира на 16-та позиция.

Методите **indexOf(…)** и **lastIndexOf(…)** претърсват съдържанието на текстова последователност, но в различна посока. Търсенето при първата функция започва от началото на низа към неговия край, а при втората функция – отзад-напред. Когато се интересуваме от първия срещнат резултат, тогава използваме **indexOf(…)**. Ако искаме да претърсваме низа от неговия край (например за откриване на последната точка в името на даден файл или последната наклонена черта в URL адрес), уместно решение е **lastIndexOf(…)**.

Понякога искаме да открием всички срещания на даден подниз в текущия низ. Използването на двата метода само с 1 подаден аргумент за търсен низ не би ни свършило работа, защото винаги ще връща само първото срещане на подниза. Ето защо е възможно подаването на втори параметър за индекс, който посочва началната позиция, от която започва търсенето.

Всички срещания на дадена дума – пример

Ето един пример за използването на **indexOf (…) по** дадена дума и начален индекс: откриване на всички срещания на думата "Java" в даден текст:

```
String quote = "The main subject in the \"Intro Java\"" +
        " book is Java for Java newbies.";
int index = quote.indexOf("Java");
while(index != -1) {
        System.out.println("Java found on index: " + index);
        index = quote.indexOf("Java", index + 1);
}
```

Първата стъпка е да направим търсене за ключовата дума "Java". Ако думата е открита в текста (т.е. връщаната стойност е различна от -1), извеждаме я на конзолата и продължаваме търсенето надясно от позицията, на която сме открили думата, увеличена с 1. Повтаряме действието, докато **indexOf(…)** върне стойност -1.

Забележка: Ако на последния ред пропуснем задаването на начален индекс, то търсенето винаги ще започва отначало и ще връща една и съща стойност. Това ще доведе до безкраен цикъл на приложението ни. Ако пък търсим директно от индекса, без да увеличаваме с единица, ще попадаме отново на последния резултат, чийто индекс сме записали. Ето защо правилното търсене на следващ резултат е с аргумент **index + 1**. За **lastIndexOf(…)**, аналогично, тъй като търсенето е в обратен ред, индексът се намалява с единица.

Преминаване към главни и малки букви

Понякога имаме нужда да променим съдържанието на символен низ, така че всички символи в него да бъдат само с главни или малки букви. Двата метода, които биха ни свършили работа в случая, са **toLowerCase()** и **toUpperCase()**. Първата функция конвертира всички главни букви към малки:

```
String text = "All Kind OF LeTTeRs";
System.out.println(text.toLowerCase());
```

В примера се вижда, че всички главни букви от текста сменят регистъра си и целият текст остава изцяло с малки букви.

Ако искаме да сравним въведен вход от потребителя и не сме сигурни по какъв точно начин е написан той, можем да уеднаквим регистъра на буквите и да го сравним с дефинираната от нас константа. По този начин не правим разлика за малки и главни букви. Например, ако имаме входен панел на потребителя, в който въвеждаме име и парола, и няма значение дали паролата е написана с малки, или главни букви, може да направим подобна проверка:

```
String pass1 = "Parola";
String pass2 = "PaRoLa";
String pass3 = "parola";
boolean isEqual;
isEqual = pass1.toUpperCase().equals("PAROLA") ; // true
```

```
isEqual = pass2.toUpperCase().equals("PAROLA") ; // true
isEqual = pass3.toUpperCase().equals("PAROLA") ; // true
```

В примера сравняваме 3 пароли с еднакво съдържание, но различен регистър, като при крайната проверка съдържанието им е еквивалентно на "PAROLA". В този случай малко обезсмисляме действието на метода **equalsIgnoreCase(…)**, като дефинираме проверката ръчно.

Премахване на празно пространство в края на низ

Въвеждайки текст във файл или през конзолата, понякога се появяват 'паразитни' интервали в началото или в края на текста. В началото или след края на дадена променлива може да се запише неволно някой друг интервал или табулация, които да не могат да се доловят на пръв поглед. Това може да не е съществено, но ако валидираме потребителски данни, би било проблем от гледна точка на проверка съдържанието на входната информация. На помощ идва методът **trim()** – той се грижи именно за премахването на паразитните празни места. Извиквайки метода на променлива от тип **String**, която има празни места в началото или края, той ще се погрижи за премахването им. Празните места могат да бъдат интервали, табулация, нови редове и др.

Ако в променливата **fileData** сме прочели съдържанието на файл, в който е записано име, а пишейки текста или преобръщайки го от един формат в друг са се появили паразитни интервали, променливата може да изглежда по подобен начин:

```
String fileData = "   \n\n     Mario Peshev       ";
```

Ако изведем съдържанието на конзолата, ще получим 2 празни реда, последвани от няколко интервала, търсеното от нас име и още няколко допълнителни интервала в края. Тъй като на нас ни е необходимо само името, може да редуцираме информацията от променливата и да премахнем ненужните интервали:

```
String reduced = fileData.trim();
```

Когато изведем повторно информацията на конзолата, съдържанието ще бъде "**Mario Peshev**", без нежеланите интервали.

Форматиране на низове

Java предлага на програмиста механизми за форматиране на символните низове. Практически всеки създаден обект на клас, както и примитивните променливи, могат да бъдат представени като текстово съдържание. Налице са форматиращи класове и методи, които служат за правилното форматиране на текст, числа, дати. Спомнете си метода **printf(…)** от **System.out.printf(…)** – с негова помощ извеждаме символни низове с предварително форматирано съдържание, можем да задаваме шаблони, в които да попълваме празните места с променливи или литерали; да форматираме дати, числа и т.н.

Класът java.util.Formatter

java.util.Formatter дава възможност за извеждане на форматиращи символни низове. Сред възможностите на класа са подравняването на текста и различни методи за форматиране на текст, символи, дати и специфичен изход в зависимост от местоположението. Създаването на класа е вдъхновено от функцията **printf(…)** в езика C, като имплементацията е реализирана със сходен синтаксис, но с по-стриктни изисквания, съобразени с езика Java.

Всеки метод, който връща форматиран изход, изисква форматиращ стринг и списък от аргументи. Форматиращият низ е **String** обект, който съдържа фиксиран текст и един или повече вложени форматиращи спецификатори (**format specifiers**). Основните спецификатори за символни и числови типове имат следния синтаксис:

```
%[индекс_на_аргумента$][флагове][ширина][.точност]формат
```

- **индекс_на_аргумента** – незадължителен спецификатор; десетично число, указващо позицията на аргумента. Първият аргумент има индекс "1$", вторият – "2$", и т.н.
- **флагове** – незадължителен спецификатор; списък от символи, модифициращи начина на извеждане на низа. Зависи пряко от формата.
- **ширина** – незадължителен спецификатор; неотрицателно десетично число, посочващо минималния брой от символи, които да бъдат изведени на изхода. Удобен за таблично форматиране.
- **точност** – незадължителен спецификатор; неотрицателно десетично число, ограничаващ броя символи. Зависи от типа формат, широко използван при десетични числа.
- **формат** (**conversion**) – символ, указващ как да бъде форматиран аргументът. Зависи от типа на подадения аргумент.

Упражнения

1. Напишете програма, която прочита символен низ, обръща го отзад напред и го принтира обратно на конзолата. Например: "introduction" → "noitcudortni".
2. Напишете програма, която открива колко пъти даден подниз се съдържа в текст. Например, ако търсим подниза "in" в текста:

```
We are living in a yellow submarine. We don't have anything else. Inside the submarine is very tight. So we are drinking all the day. We will move out of it in 5 days.
```

   Резултатът е 9.

3. Даден е текст. Напишете програма, която променя регистъра на буквите на всички места в текста, заградени с таговете **<upcase>** и **</upcase>**. Таговете не могат да бъдат вложени.

   Пример:

```
We are living in a <upcase>yellow submarine</upcase>. We don't
```

```
have <upcase>anything</upcase> else.
```

Резултат:

```
We are living in a YELLOW SUBMARINE. We don't have
ANYTHING else.
```

5. Дефиниране на класове

Да си припомним: какво са класовете и обектите?

Клас (**class**) наричаме описание на даден обект от реалността. Класът представлява шаблон, който описва видовете състояния и поведението на обектите (екземплярите), които биват създавани от този клас (шаблон).

Обект (**object**) наричаме екземпляр създаден по дефиницията (описанието) на даден клас. Когато един обект е създаден по описанието, което един клас дефинира, казваме, че **обектът е от тип "името на този клас"**.

Например, ако имаме клас **Dog**, описващ някакви характеристики на куче от реалния свят, казваме, че обектите, които са създадени по описанието на този клас са от тип – класът **Dog**. Това означение е същото, като например, низът **"some string"** казваме, че е от тип **String**. Разликата е, че обектът от тип **Dog** е екземпляр от клас, който не е част от библиотеката с класове на Java, а е дефиниран от самите нас.

Какво съдържа един клас?

Класът съдържа дефиниция на това какви данни трябва да се съдържат в един обект, за да се опише състоянието му. Обектът (конкретния екземпляр от този клас) съдържа самите данни. Тези данни дефинират състоянието му.

Освен състоянието, в класа също се описва и поведението на обектите. Поведението се изразява в действията, които могат да бъдат извършвани от обектите. Средството на ООП, чрез което можем да описваме поведението на обектите от даден клас, е декларирането на методи в класа.

Елементи на класа

Сега ще изброим основните елементи на един клас, а по-късно ще разгледаме подробно всеки един от тях.

Основните елементи на класа са следните:

- **Декларация на класа (class declaration)** – това е редът, на който декларираме името на класа. Например:

```
publicclass Dog {
```

- **Тяло на клас** – по подобие на методите, класовете също имат част, която следва декларацията им, оградена с фигурни скоби – "**{**" и "**}**", между които се намира съдържанието на класа. Тя се нарича тяло на класа. Елементите на класа, които се описват в тялото му са изброени в следващите точки.

```
publicclass Dog {
        }
```

- **Конструктор (constructor)** – това е псевдометод, който се използва за създаване на нови обекти. Така изглежда един конструктор:

```
public Dog() {
}
```

- **Полета (fields)** – полетата са променливи (някъде в литературата се срещат като **член-променливи**), декларирани в класа. В тях се пазят данни, които отразяват състоянието на обекта и са нужни за работата на методите на класа. Стойността, която се пази в полетата, отразява конкретното състояние на дадения обект, но съществуват и такива полета, наречени статични, които са общи за всички обекти.

```
private String name;
```

- **Свойства (properties)** – наричаме характерните особености на даден клас. Обикновено стойността на тези характеристики се пази в полета. Подобно на полетата, свойствата могат да бъдат притежавани само от конкретен обект, или да са споделени между всички обекти от тип даден клас.

```
private String name;
```

- **Методи (methods)** – от главата "Методи", знаем, че методите са мястото в класа, където се описва поведението на обектите от този тип. В методите се изпълняват алгоритмите и се обработват данните на обекта.

Ето как изглежда един клас, който сме дефинирали сами и който притежава елементите, които описахме току-що:

Dog.java

```
class Dog {
	private String name;
	public Dog() {
		this.name = "Sharo";
	}
	public Dog(String name) {
		this.name = name;
	}
	public String getName() {
		returnthis.name;
	}
	publicvoid setName(String name) {
		this.name = name;
	}
	publicvoid bark() {
		System.out.printf("Dog %s said: Wow-wow!%n", name);
	}
}
```

Сега няма да обясняваме изложения код, тъй като подробна информация ще бъде дадена по време на обяснението как се декларира всеки един от елементите на класа.

Как да използваме дефиниран от нас клас?

За да можем да използваме някой клас, първо трябва да създадем обект от тип този клас. За целта използваме ключовата дума **new** в комбинация с някой от конструкторите на класа. Това ще създаде обект от дадения тип.

За да можем да манипулираме новосъздадения обект, ще трябва да го присвоим на променлива от типа на обекта. По този начин в тази променлива ще бъде запазена връзка (референция) към него.

Чрез променливата, използвайки точкова нотация, можем да извикваме методите, getter и setter методите на обекта, както и да достъпваме полетата (член-променливите) му.

Пример – кучешка среща

Нека вземем примера от предходната секция на тази глава, където е дефинирахме класа, който описва куче – **Dog** и добавим метод **main()** към него. В него ще онагледим казаното току-що:

```
publicstaticvoid main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Write first dog's name: ");
        String firstDogName = input.nextLine();
        Dog firstDog = new Dog(firstDogName);
        System.out.print("Write second dog's name: ");
        Dog secondDog = new Dog();
        secondDog.setName(input.nextLine());
        Dog thirdDog = new Dog();

        Dog[] dogs = new Dog[] { firstDog, secondDog, thirdDog };
        for (Dog dog : dogs) {
                dog.bark();
```

```
        }
}
```

Съответно изходът от изпълнението ще бъде следният:

```
Write first dog's name: Bobcho
Write second dog's name: Walcho
Dog Bobcho said: Wow-wow!
Dog Walcho said: Wow-wow!
Dog Sharo said: Wow-wow!
```

В този метод, с помощта на класа **Scanner**, получаваме имената на обектите от тип куче, които потребителят трябва да въведе в конзолата.

Присвояваме първия въведен низ на променливата **firstDogName**. След това използваме тази променлива при създаването на първия обект от тип **Dog** – **firstDog**, като я подаваме като параметър на конструктора.

Създаваме втория обект от тип **Dog**, без да подаваме низ за името на кучето на конструктора му. След това въвеждаме името на второто куче, чрез класа **Scanner**, и получената стойност директно подаваме на setter метода – **setName()**. Извикването на метода **setName()** става чрез точкова нотация, приложена към променливата, която пази референция към втория създаден обект от тип **Dog – secondDog.setName()**.

Когато създаваме третия обект от тип **Dog**, не подаваме име на кучето на конструктора, нито след това модифицираме подразбиращата се стойност "**Sharo**".

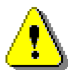
След това създаваме масив от тип **Dog**, като го инициализираме с трите обекта, които току що създадохме.

Накрая, използваме цикъл, за да обходим масива от обекти от тип **Dog**. На всеки елемент от масива, отново използвайки точкова нотация, извикваме метода **bark()** чрез **dog.bark()**.

Модификатори и нива на достъп

Както казахме, в Java, има три модификатора за достъп – **public**, **protected** и **private**. С тях ние ограничаваме или позволяваме достъпа (видимостта) до елементите на класа, пред които те са поставени. На всеки един от тях, съответства ниво на достъп, което носи името на съответния модификатор, съответно – **public**, **protected** и **private**.

В Java обаче, има четвърто ниво на видимост до елемент на клас, за което няма модификатор за достъп. В литературата, това ниво на достъп се нарича **default** или **package** (след малко ще видим защо), а понякога в по-стари книги за Java се среща и като **friendly**. Съответно, класове, полета, свойства или методи на даден клас, които нямат модификатор за видимост в декларацията си, считаме, че имат ниво на достъп **default**.

	**В Java има три модификатора за видимост, но четири нива на достъп. Четвъртото, се нарича ниво на достъп default, и е в сила, когато пред съответния елемент на класа няма никакъв модификатор за достъп.**

Сега ще дадем общо обяснение за различните нива, а в последствие, когато разглеждаме всеки един от елементите на класа, ще дадем по-подробна информация за всяко едно от тях.

Ниво на достъп public

Използвайки модификатора **public**, ние указваме на компилатора, че елементът, пред който е поставен, може да бъде достъпен от всеки друг клас, независимо дали е в текущия пакет или извън него. Той определя най-малко ограничителното ниво на видимост от всички нива в Java.

Ниво на достъп default

Това ниво на достъп се прилага, когато не се използва никакъв модификатор за достъп пред съответния елемент.

То е по-ограничително от **public**-видимостта, тъй като позволява, да достъпваме съответният елемент, само от класове, които се намират в същия пакет, в който се намира класът, на който принадлежи елементът.

Ниво на достъп private

Това е нивото на достъп, което налага най-голяма рестрикция на видимостта на класа и елементите му. Модификаторът **private** служи за индикация, че елементът, за който се отнася, не може да бъде достъпван от никой друг клас, дори този клас да се намира в същия пакет.

Деклариране на класове

Декларирането на клас има строго определени правила (синтаксис), които трябва да спазваме:

```
[<access_modifier>] class <class_name>
```

Когато декларираме клас, задължително трябва да използваме ключовата дума **class**. След нея трябва да стои името на класа **<class_name>**.

Задължителните елементи от декларацията на класа са ключовата дума class и името на класа.

Освен ключовата дума **class** и името на класа, в декларацията на класа могат да бъдат използвани някои модификатори. Тук, ще обърнем внимание само на позволените модификатори за достъп.

Ключовата дума this

Ключовата дума **this**, в Java, е референция към текущия обект – обектът, чийто метод или конструктор бива извикван. Можем я приемем като указател (референция), дадена ни априори от създателите на Java, с която да достъпваме елементите (полета, методи, конструктори) на собствения ни клас:

```
this.myField

this.doMyMethod()
```

```
this(3,4)
```

В момента няма да обясняваме изложения код. Разяснения ще дадем по-късно, в местата от секциите на тази глава, посветени на елементите на класа (полета, методи, конструктори) и засягащи ключовата дума **this**.

Деклариране на полета в даден клас

До момента, сме се сблъсквали само с два типа променливи (вж. главата "Методи"), в зависимост от това къде са декларирани:

- **Локални променливи** – това са променливите, които са дефинирани в тялото на някой метод (или блок).
- **Параметри** – това са променливите в списъка с параметри, който един метод може да има в реда, на който е деклариран.

В Java съществува и трети вид променливи, наречени **полета** (**fields**) или **член-променливи на класа** (**instance variables**).

Те се декларират в тялото на класа, но извън тялото на блок, метод или конструктор (какво е конструктор, ще разгледаме подробно след малко).

	Полетата се декларират в тялото на класа, но извън тялото на метод, конструктор или блок.

Ето един примерен код, в който се декларират различни полета:

MyClass.java

```
class MyClass {
        intage;
        longdistance;
        String[] names;
        Dog myDog;
}
```

Формално, декларацията на полетата става по следния начин:

```
[<modifiers>] <field_type><field_name>;
```

<field_type> определя типа на даденото поле. Той може да бъде, както примитивен тип (**byte**, **short**, **char** и т.н.) или масив, така и от тип, някакъв клас (например **String**).

<field_name> е името на даденото поле. Както при имената на обикновените променливи, когато именуваме една член-променлива трябва да спазваме правилата за идентификатори в Java (вж. главата "Примитивни типове и променливи").

<modifiers> е понятие, с което сме означили, както модификаторите за достъп, така и други модификатори. Те не са задължителна част от декларацията на едно поле.

В тази глава, от другите модификатори, които не са за достъп, и могат да се използват при декларирането на полета на класа, ще обърнем внимание само на **static** и **final**. Оставащите модификатори (**transient** и **volatile**) са извън обсега на тази книга и няма да бъдат разглеждани.

Методи

В главата "Методи" подробно се запознахме с това как да декларираме и използваме метод. В тази секция, накратко ще припомним казаното там и ще се фокусираме върху някои нови особености при декларирането и създаването на методи.

Деклариране на методи в даден клас

Декларирането на методи, както знаем става по следния начин:

```
[<modifiers>] <return_type><method_name>([<parameters_list>]) {
	[<return_statement>];
}
```

Задължителните елементи при декларирането на метода са типът на връщаната стойност **<return_type>**, името на метода **<method_name>** и отварящата и затварящата кръгли скоби – "(" и ")".

Списъкът от параметри **<params_list>** не е задължителен. Използваме го да подаваме информация на метода, който декларираме, ако той се нуждае от такава.

Знаем, че ако типът на връщаната стойност **<return_type>** е **void**, тогава **<return_statement>** може да участва само с оператора **return**, с цел прекратяване действието на метода. Ако **<return_type>** е различен от **void**, методът задължително трябва да връща резултат чрез ключовата дума **return**, като резултатът е от тип **<return_type>** или съвместим с него.

Реалната работа, която методът трябва да свърши, се намира в тялото му, заградена от фигурни скоби – "**{**" и "**}**".

Макар, че разгледахме някои от модификаторите за достъп, позволени да се използват при декларирането на един метод, в секцията "Видимост на полета и методи" ще разгледаме по-подробно тази тема.

Ще разгледаме модификатора **static** в последната секция на тази глава.

Пример – деклариране на метод

Нека погледнем декларирането на един метод за намиране сбор на две цели числа:

```
int add(int number1, int number2) {
        int result = number1 + number2;
        return result;
}
```

Името, с което сме го декларирали, е **add()**, а типът на връщаната му стойност е **int**. Списъкът му от параметри се състои от два елемента – променливите

number1 и **number2**. Съответно, връщаме стойността на сбора от двете числа като резултат.

Конструктори

В обектно-ориентираното програмиране, когато създаваме обект от даден клас, е необходимо да извикаме елемент от класа, наречен конструктор.

Какво е конструктор?

Конструктор на даден клас, наричаме псевдометод, който няма тип на връщана стойност, носи името на класа и който се извиква чрез ключовата дума **new**.

Задачата на конструктора е да задели памет в хийпа, където ще съхраняват данните, които се пазят в полетата на конкретния обект (тези, които не са **static**), инициализира всяко поле с подразбиращата се за типа му стойност и връща референция към новосъздадения обект.

Извикване на конструктор

За разлика от методите, в Java, единствения начин да извикаме един конструктор е чрез използването на ключовата дума **new**.

Нека разгледаме един пример, от който ще стане ясно как работи конструктора. От главата "Създаване и използване на обекти", знаем как се създава обект:

```
Dog myDog = new Dog();
```

Деклариране на конструктор

Ако имаме класа **Dog**, ето как би изглеждал неговия най-опростен конструктор:

```
public Dog() {        }
```

Формално, декларацията на конструктора изглежда по следния начин:

```
[<modifiers>] <class_name>([<parameters_list>])
```

Както вече казахме, конструкторите приличат на методи, но нямат тип на връщана стойност (затова ги нарекохме псевдометоди).

Име на конструктора

В Java, задължително, името на всеки конструктор съвпада с името на класа, в който го декларираме - **<class_name>**. В примера по-горе, името на конструктора е същото, каквото е името на класа – **Dog**. Трябва да знаем, че както при методите, името на конструктора винаги е следвано от кръгли скоби – "(" и ")".

Трябва да отбележим, че в Java е напълно легално, да се декларира метод, който притежава име, което съвпада с името на класа. Разбира се това не го прави конструктор, тъй като конструкторите нямат тип на връщаната стойност. Ето един такъв пример:

MyClass.java

```
publicclass MyClass {
	public MyClass() {
	}
	String MyClass() {
		return"MyClass() method has finished successfully.";
	}
	publicstaticvoid main(String[] args) {
		MyClass instance = new MyClass();
		System.out.println(instance.MyClass());
	}
}
```

Свойствата в Java

Въпреки, че Java е обектно-ориентиран език, в нейната спецификация няма елемент от класа, който да съответства на идеята за свойство. От друга страна, тъй като използването свойства е доказано добра практика и важна част от концепциите на обектно-ориентираното програмиране, в тази секция ще

разгледаме, как свойствата могат да бъдат реализирани в един Java клас. Това става чрез деклариране на два метода – един за достъп (четене) и един за модификация (записване) на стойността на съответното свойство.

Нека разгледаме един пример. Нека имаме отново клас **Dog**, който описва куче. Характерно свойство за едно куче е, например, цвета му (colour). Достъпът до свойството цвят на едно куче може да осъществим по следния начин:

```
String colourName = dogInstance.getColour();
dogInstance.setColour("black");
```

Статични полета

Когато създаваме обекти от даден клас, всеки един от тях има различни стойности в полетата си. Например, нека разгледаме отново класа **Dog**:

Dog.java

```
publicclass Dog {
	private String name;
	privateintage;
}
```

Той има две полета съответно за име – **name** и възраст – **age**. Във всеки обект, всяко едно от тези полета има собствена стойност, която се съхранява на различно място в паметта за всеки обект.

Понякога обаче, искаме да имаме полета, които са общи за всички обекти от даден клас. За да постигнем това, трябва в декларацията на тези полета да използваме модификатора **static**. Както казахме, такива полета се наричат **статични полета**. В литературата се срещат, също и като **променливи на класа**.

Декларация на статични полета

Статичните полета ги декларираме по същия начин, както се декларира поле на клас, като след модификатора за достъп (ако има такъв), добавяме ключовата дума **static**:

```
[<access_modifier>] static <field_type><field_name>
```

Ето как би изглеждало едно поле **dogCount**, което пази информация за броя на създадените обекти от клас **Dog**:

Dog.java

```
publicclass Dog {
        staticintdogCount;
        private String name;
        privateintage;
}
```

Статичните полета се създават, когато за първи път се опитаме да създадем обект от класа, на който принадлежат или когато заредим класа в паметта (как става това обаче, е извън обхвата на тази книга и няма да го разглеждаме). След създаването си, по подобие на обикновените полета в класа, те се инициализират с подразбиращата се стойност за типа си.

Вътрешни, локални и анонимни класове

В Java можем да дефинираме класове вътре в даден друг клас или дори в даден метод. Понякога това може да е много удобно, когато ни трябва клас, който искаме да използваме временно или искаме да скрием от външния свят.

Вътрешни класове

В Java е възможно в един клас да се дефинира друг клас, т.е. класът да е член на клас. Такъв клас наричаме **вътрешенклас** (**inner class**, **nested class**). Нека разгледаме тази възможност с един пример:

OuterClass.java

```
publicclass OuterClass {
	private String name;
	private OuterClass(String name) {
		this.name = name;
	}
	privateclass InnerClass {
		private String name;
		private InnerClass(String name) {
			this.name = name;
		}
		privatevoid printNames() {
			System.out.println("Inner name: " + this.name);
			System.out.println("Outer name: " +
				OuterClass.this.name);
		}
	}

	publicstaticvoid main(String[] args) {
		OuterClass outerClass = new OuterClass("outer");
		InnerClass innerClass = outerClass.new InnerClass("inner");
		innerClass.printNames();
	}
```

```
}
```

В примера външният клас **OuterClass** дефинира в себе си като **private** член класа **InnerClass**. Нестатичните методи на вътрешния клас имат достъп както до собствената си инстанция **this**, така и до инстанцията на външния клас (чрез синтаксиса **OuterClass.this**). При създаването на вътрешния клас на конструктора му се подава **this** референцията на външния клас, защото вътрешният клас не може да съществува без конкретна инстанция на външния. Забележете, че външния клас може да вика свободно **private** методи и конструктори от вътрешния клас.

Ако изпълним горния пример, ще получим следния резултат:

```
Inner name: inner
Outer name: outer
```

Вътрешните класове могат да бъдат декларирани като статични (чрез модификатора **static**). В този случай те могат да съществуват ибез външния клас, в който са разположени, но нямат достъп до неговата **this**инстанция.

Локални класове

В Java можем да дефинираме класове и в даден метод. Наричаме ги **локални класове** (**local classes**). Локалните класове са подобни на вътрешните класове, но не могат да бъдат статични. Те имат достъп до член-променливите и методите на външния им клас. Локалните класове могат да осъществяват достъп и до променливите, декларирани в метода, в който се съдържат, стига тези променливи да са обявени като **final**. Ето един пример:

LocalClassExample.java

```
publicclass LocalClassExample {
        publicstaticvoid main(String[] args) {
                finalint value = 5;
```

```
        class LocalClass {
            void printSomething() {
                System.out.println(value);
            }
        }
        LocalClass localClass = new LocalClass();
        localClass.printSomething();
    }
}
```

Ако изпълним горния пример, ще получим следния резултат:

```
5
```

Локалните класове са достъпни само и единствено в метода, в който са декларирани и нямат модификатори за видимост и не могат да бъдат статични, както всяка една локална променлива.

Анонимни класове

В Java можем да декларираме локален клас без име. Такъв клас се нарича **анонимен клас**(**annonymous class**). Да разгледаме един пример:

AnnonymousClassExample.java

```
publicclass AnnonymousClassExample {
    publicstaticvoid main(String[] args) {
        new Object() {
            void printSomething() {
                System.out.println("I am annonymous class.");
            }
        }.printSomething();
```

```
        }
}
```

В примера декларираме клас без име (анонимен клас), който наследява класа **java.lang.Object** и добавя към него нов метод **printSomething()**. След това създаваме инстанция на този анонимен клас и му извикваме добавения метод **printSomething()**. За наследяването ще ви разкажем подробно в главата "Принципи на ООП". За момента приемете, че анонимните класове са локални класове без име, които ползват за основа даден съществуващ клас и му добавят допълнителни методи.

Ако изпълним горния пример, ще получим следния резултат:

```
I am annonymous class.
```

Упражнения

1. Дефинирайте клас **Student**, който съдържа следната информация за студентите: трите имена, курс, специалност, университет, електронна поща и телефонен номер.

2. Декларирайте няколко конструктора за класа **Student**, които имат различни списъци с параметри (за цялостната информация за даден студент или част от нея). Данните, за които няма входна информация да се инициализират съответно с **null** или **0**.

3. Добавете статично поле в класа **Student**, в което се съхранява броя на създадените обекти от този клас.

4. Добавете метод в класа **Student**, който извежда пълна информация за студента.

6. Линейни структури от данни

Какво е структура данни?

Много често, когато пишем програми ни се налага да работим с множество от обекти (данни). Понякога добавяме и премахваме елементи, друг път искаме да ги подредим или да обработваме данните по друг специфичен начин. Поради това са изработени различни начини за съхранение на данните в зависимост от задачата, като най-често между елементите съществува някаква наредба (например обект А е преди обект Б).

В този момент на помощ ни идват **структурите данни** – множество от данни организирани на основата на логически и математически закони. Много често избора на правилната структура прави програмата много по-ефективна – можем да спестим памет и време за изпълнение.

Какво е абстрактен тип данни?

Най-общо **абстрактният тип данни (АТД)** дава определена дефиниция (абстракция) на конкретната структура т.е. определя допустимите операции и свойства, без да се интересува от конкретната реализация. Това позволява един тип абстрактни данни да има различни реализации, респективно различна ефективност.

Основни структури от данни в програмирането

Могат ясно да се различат няколко групи структури:

- Линейни – към тях спадат списъците, стековете и опашките
- Дървовидни – различни типове дървета
- Речници – хеш-таблици
- Множества

В настоящата тема ще разгледаме линейните (списъчни) структури от данни, а в следващите няколко теми ще обърнем внимание и на по-сложните структури като дървета, графи, хеш-таблици и множества и ще обясним кога се използва и прилага всяка от тези структури.

Овладяването на основните структури данни в програмирането е от изключителна важност, тъй като без тях не можете да програмирате ефективно. В основата на програмирането стоят структурите от данни и алгоритмите, с които малко по малко ви запознаваме в настоящата книга.

Списъчни структури

Най–често срещаните и използвани са линейните (списъчни) структури. Те представляват абстракция на всякакви видове редици, последователности, поредици и други подобни от реалния свят.

Списък

Най–просто можем да си представим списъка като редица от елементи. Например покупките от магазина или задачите за деня представляват списъци. В списъка можем да четем всеки един елементите напр. покупките, както и да добавяме нови покупки в него. Можем спокойно да задраскваме (изтрием) покупки или да ги разместваме.

Абстрактна структура данни "списък"

Нека сега дадем една по-строга дефиниция на структурата списък:

Списък е линейна структура от данни, която съдържа поредица от елементи. Списъкът има свойството дължина (брой елементи) и елементите му са наредени последователно.

Списъкът позволява добавяне елементи на всяко едно място както и премахването им, както и последователното им обхождането. Както споменахме по-горе един АТД може да има няколко реализации. Пример за такъв АТД е интерфейсът **java.util.List**.

Интерфейсите в Java изграждат една "рамка" за техните имплементации – класовете. Тази рамка представлява съвкупност от методи и свойства, които всеки клас, имплементиращ интерфейса, трябва да реализира.

Всеки АТД реално определя някакъв интерфейс. Нека разгледаме интерфейса **java.util.List**. Основните методи, които той декларира, са:

- **void add(int, Object) -** добавя елемент на предварително избрана позиция
- **boolean contains(Object) –** проверява дали елемента се съдържа в списъка
- **Object get(int) –** взима елемента на съответната позиция
- **boolean isEmpty() –** проверява дали списъка е празен
- **boolean remove(Object) –** премахва съответния елемент
- **Object remove(int) –** премахва елемента на дадена позиция
- **int indexOf(Object) –** връща позицията на елемента

Нека видим няколко от основните реализации на АТД списък и обясним в какви ситуации се използва всяка от тях.

Класът ArrayList

След като се запознахме с някои от основните реализации на списъците, ще се спрем на класовете в Java, които ни предоставят списъчни структури "на готово". Първият от тях е класът **ArrayList**, който представлява динамично-разширяем масив. Той е реализиран по сходен начин със статичната реализация на списък, която разгледахме по-горе. Имаме възможност да добавяме, премахваме и търсим елементи. Някои по-важни методи, които можем да използваме са:

- **add(Object)** – добавяне на нов елемент
- **add(index, Object)** – добавяне елемент на определено място (индекс)
- **size()** – връща броя на елементите в списъка

- **remove(Object)** – премахване определен елемент
- **remove(index)** – премахване на елемента на определено място (индекс)
- **clear()** – изчистване на списъка

Както видяхме, един от основните проблеми при тази реализация е преоразмеряването на вътрешния масив при добавянето и премахването на елементи. В класа **ArrayList** проблемът е решен чрез предварително създаване на по-голям масив, който ни предоставя възможност да добавяме елементи, без да преоразмеряваме масива при всяко добавяне или премахване на елементи. След малко ще обясним това в детайли.

Класът ArrayList – пример

В класа **ArrayList** можем да записваме всякакви елементи – числа, символни низове и други обекти. Ето един малък пример:

```
import java.util.ArrayList;
import java.util.Date;

publicclass ArrayListExample {
	publicstaticvoid main(String[] args) {
		ArrayList list = new ArrayList();
		list.add("Hello");
		list.add(5);
		list.add(3.14159);
		list.add(new Date());

		for (int i=0; i<list.size(); i++) {
			Object value = list.get(i);
			System.out.printf("Index=%d;	Value=%s\n",	i,
value);
```

```
            }
        }
}
```

В примера създаваме **ArrayList** и записваме в него няколко елемента от различни типове: **String**, **int**, **double** и **Date**.След това итерираме по елементите и ги отпечатваме. Ако изпълним примера, ще получим следния резултат:

```
Index=0; Value=Hello
Index=1; Value=5
Index=2; Value=3.14159
Index=3; Value=Sat Nov 29 23:17:01 EET 2008
```

Шаблонни класове (generics)

Когато използваме класа **ArrayList**, а и всички класове, имплементиращи интерфейса**java.util.List,** се сблъскваме с проблема, който видяхме по-горе: когато добавяме нов елемент от даден клас ние го предаваме като обект от тип **Object**. Когато по-късно търсим даден елемент, ние го получаваме като **Object** и се налага да го превърнем в изходния тип. Не ни се гарантира, обаче, че всички елементи в списъка ще бъдат от един и същ тип. Освен това превръщането от един тип в друг отнема време, което забавя излишно изпълнението на програмата.

За справяне в описаните проблеми на помощ идват шаблонните класове. Образно казано те са шаблони създадени да работят с един или няколко типа, като при създаването си ние указваме какъв точно тип обекти ще съхраняваме в тях. Създаването на инстанция от даден шаблонен тип, примерно **GenericType**, става като в счупени скоби се зададе типа, от който трябва да бъдат елементите му:

```
GenericType<T> instance = new GenericType<T>();
```

Този тип **T**може да бъде всеки наследник на класа **java.lang.Object**, примерно **String** или **Date**. Понеже числата не са обекти и не наследяват класа **Object**, ако трябва да ги използваме като тип в шаблонен клас, трябва да използваме съответния им обвиващ (wrapper) клас. Така вместо примитивния тип **int** трябва да ползваме класа **Integer**, а вместо типа **boolean** трябва да ползваме класа **Boolean**.Ето няколко примера:

```
ArrayList<Integer> intList = new ArrayList<Integer>();

ArrayList<Boolean> boolList = new ArrayList<Boolean>();

ArrayList<Double> realNumbersList = new ArrayList<Double>();
```

Нека сега разгледаме някои от шаблонните колекции в Java.

Класът ArrayList<T>

ArrayList<T> е шаблонният вариант на **ArrayList**. При инициализацията на обект от тип **ArrayList<T>** указваме типа на елементите, който ще съдържа списъка, т. е. заместваме означения с **T** тип с някой истински тип данни (например число или стринг).

Например искаме да създадем списък от целочислени елементи. Можем да го направим по следния начин:

```
ArrayList<Integer> genericList = new ArrayList<Integer>();
```

Създаденият по този начин списък може да приема като стойности цели числа, но не може и други обекти, например символни низове. Ако се опитаме да добавим към **ArrayList<Integer>** обект от тип **String**, ще получим грешка по време на компилация. Чрез шаблонните типове компилаторът на Java ни пази от грешки при работа с колекции.

Забележете, че не посочваме типа **int**, а неговия обвиващ тип **Integer**. Причината за това е фактът, че шаблоните приемат като параметър само референтни типове (обекти) и не могат да работят с обикновени стойностни типове, които не се пазят в динамичната памет. По тази причина ползването на **ArrayList<String>**става директно, а ползването на списък от **int** или **double**

изисква да ползваме съответните обвиващи типове: **ArrayList<Integer>** и **ArrayList<Double>**.

Класът LinkedList<T>

Този клас представлява динамична реализация на двусвързан списък. Елементите му пазят информация за обекта, който съхраняват, и указател към следващия и предишния елемент.

Кога да използваме LinkedList<T>?

Видяхме, че динамичната и статичните реализации имат специфика по отношение бързодействие на различните операции. С оглед на структурата на свързания списък трябва да имаме предвид следното:

- Добавянето на елементи в **LinkedList** става много бързо – независимо от броя на елементите.
- Можем да добавяме бързо в началото и в края на списъка (за разлика от **ArrayList<T>**).
- Търсенето на елемент по индекс или по съдържание в **LinkedList** е бавна операция, тъй като се налага да обхождаме всички елементи последователно като започнем от началото на списъка.
- Изтриването на елемент е бавна операция, защото включва търсене.

Основни операции в класа LinkedList<T>

LinkedList<T> притежава същите операции като **ArrayList<T>**, което прави двата класа взаимозаменяеми в зависимост от конкретната задача. По-късно ще видим, че **LinkedList<T>** се използва и при работа с опашки.

Кога да ползваме LinkedList<T>?

Като цяло класът **LinkedList<T>** се използва много рядко, защото **ArrayList<T>** върши същата работа не по-бавно, а предлага в допълнение и други бързи операции.

Стек

Да си представим няколко кубчета, които сме наредили едно върху друго. Можем да слагаме ново кубче на върха, както и да махаме най-горното кубче. Или да си представим една ракла. За да извадим прибраните дрехи или завивки от дъното на раклата, трябва първо да махнем всичко, което е върху тях.

Точно тази конструкция представлява стекът – можем да добавяме елементи и да извличаме последният добавен елемент, но не и предходните (които са затрупани под него). Стекът е често срещана и използвана структура от данни. Стек се използва и вътрешно от Java виртуалната машина за съхранение на променливите в програмата и параметрите при извикване на метод.

Абстрактна структура данни "стек"

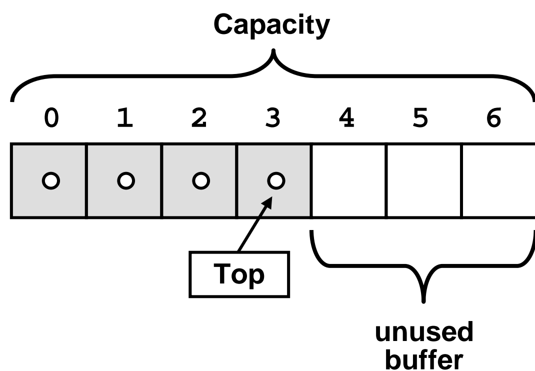
Стекът представлява структура от данни с поведение "последният влязъл първи излиза". Както видяхме в примера с кубчетата, елементите могат да се добавят и премахват само от върха на стека.

Структурата от данни стек също може да има различни реализации, но ние ще се спрем на двете основни – динамичната и статичната реализация.

Статичен стек (реализация с масив)

Както и при статичния списък и можем да използваме масив за пазене на елементите на стека. Можем да имаме индекс или указател, който сочи към елемента, който се намира на върха. Обикновено при запълване на масива следва заделяне на двойно повече памет, както това се случва при статичния списък (**ArrayList**).

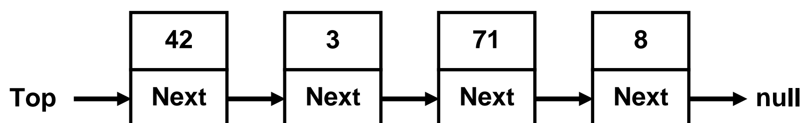
Ето как можем да си представим един статичен стек:



Както и при статичния масив се поддържа свободна буферна памет с цел по-бързо добавяне.

Свързан стек (динамична реализация)

За динамичната реализация ще използваме елементи, които пазят, освен обекта, и указател към елемента, който се намира "по-долу". Тази реализация решава ограниченията, които има статичната реализация както и необходимостта от разширяване на масива при нужда:



Когато стекът е празен, върхът има стойност **null**. При добавяне на нов елемент, той се добавя на мястото, където сочи върхът, след което върхът се насочва към новия елемент. Премахването става по аналогичен начин.

Класът Stack<T>

В Java можем да използваме класа **java.util.Stack<T>**, предоставя структурата от данни стек. Използвана е статичната имплементация като вътрешният масив се преоразмерява при необходимост.

Класът Stack<T> – основни операции

Реализирани са всички необходими операции за работа със стек:

- **push(T) –** позволява ни добавянето на нов елемент на върха на стека
- **pop() –** връща ни най-горния елемент като го премахва от стека

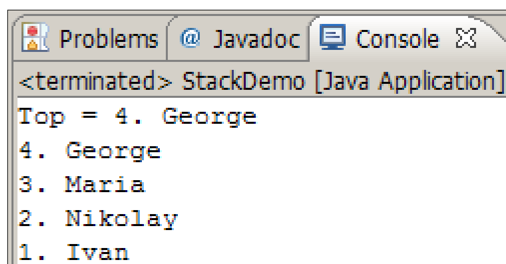
- **peek() –** връща най горния елемент без да го премахва
- **size() –** връща броя на елементите в стека
- **clear() –** премахва всички елементи
- **contains(T) –** проверява дали елемента се съдържа в стека
- **toArray() –** връща масив, съдържащ елементите от стека

Използване на стек – пример

Нека сега видим един прост пример как да използваме стек. Ще добавим няколко елемента, след което ще ги вземе всички и ще ги изведем на конзолата.

```
publicstaticvoid main(String[] args) {
	Stack<String> stack = new Stack<String>();
	stack.push("1. Ivan");
	stack.push("2. Nikolay");
	stack.push("3. Maria");
	stack.push("4. George");
	System.out.println("Top = " + stack.peek());
	while (stack.size() > 0) {
		String personName = stack.pop();
		System.out.println(personName);
	}
}
```

Тъй като стекът е структура "последен влязъл – пръв излязъл", програмата ще изведе записите в ред обратен на реда на добавянето. Ето как нейният изход:



Опашка

Структурата "**опашка**" е създадена да моделира опашки, като например опашка от чакащи документи за принтиране, чакащи процеси за достъп до общ ресурс и други. Такива опашки много удобно и естествено се моделират чрез структурата "опашка". В опашките можем да добавяме елементи само най-отзад и да извличаме елементи само от най-отпред.

Нека, например, искаме да си купим билет за концерт. Ако отидем по-рано ще си купим едни от билети. Ако обаче се забавим ще трябва да се наредим на опашката и да изчакаме всички желаещи преди нас да си купят билети. Нямаме право да се пререди́м, защото охраната ще ни се скара. Това поведение е аналогично за обектите в АТД опашка.

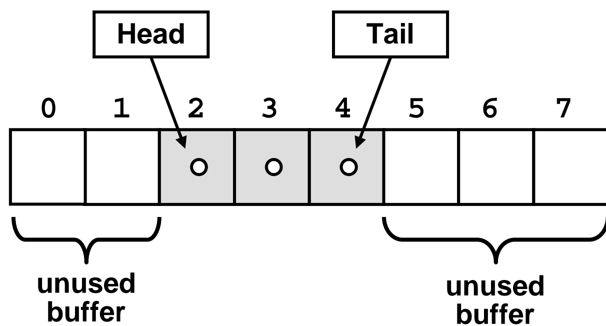
Абстрактна структура данни "опашка"

Абстрактната структура опашка изпълнява условието "първият влязъл първи излиза". Добавените елементи се нареждат в края на опашката, а при извличане поредният елемент се взима от началото (главата) й.

Както и при списъка за структурата от данни опашка отново е възможна статична и динамична реализация.

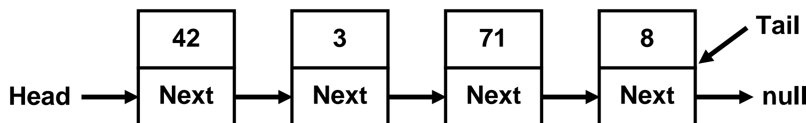
Статична опашка (реализация с масив)

В статичната опашка отново ще използваме масив за пазене на данните. При добавяне на елемент той се добавя на индекса, който следва края, след което края започва да сочи към ново добавения елемент. При премахване на елемент, се взима елемента, към който сочи главата, след което главата започва да сочи към следващия елемент. По този начин опашката се придвижва към края на масива. Когато стигне до края, при добавяне на нов елемент той се добавя на първо място. Ето защо тази имплементация се нарича още **зациклена опашка**, тъй като мислено залепяме началото и края на масива и опашката обикаля в него:



Свързана опашка (динамична реализация)

Динамичната реализация на опашката много прилича на тази на свързания списък. Елементите отново съдържат две части – обекта и указател към предишния елемент:



Тук обаче елементите се добавят на опашката, а се вземат от главата, като нямаме право да взимаме или добавяме елементи на друго място.

Интерфейсът Queue<T>

В Java се използва динамичната реализация на опашка чрез интерфейса **Queue<T>.** Както видяхме, интерфейсите декларират определени методи и свойства (т. е. АТД).При инициализация ние използвамекласа **LinkedList<T>,** като му указваме да има поведение на опашка,т.е. получаваме имплементация със свързани елементи, която ще притежава методите на характерни за опашка.Тук отново можем да укажем типа на елементите, с които ще работим, тъй като опашката и свързаният списък са шаблонни типове.

Интерфейсът Queue<T> – основни операции

Queue<T> ни предоставя основните операции характерни за структурата опашка. Ето някои от често използваните:

- **offer(T)** – добавя елемент накрая на опашката
- **poll()** – взима елемента от началото на опашката и го премахва

- **peek()** – връща елементът от началото на опашката без да го премахва
- **clear()** – премахва всички елементи от опашката
- **contains(T)** – проверява дали елемента се съдържа в опашката

Използване на опашка – пример

Нека сега разгледаме прост пример. Да си създадем една опашка и добавим в нея няколко елемента. След това ще извлечем всички чакащи елементи и ще ги изведем на конзолата:

```
publicstaticvoid main(String[] args) {
        Queue<String> queue = new LinkedList<String>();
        queue.offer("Message One");
        queue.offer("Message Two");
        queue.offer("Message Three");
        queue.offer("Message Four");

        while (queue.size() > 0) {
                String msg = queue.poll();
                System.out.println(msg);
        }
}
```

Ето как изглежда изходът е примерната програма:

```
Message One
Message Two
Message Three
Message Four
```

Вижда се, че елементите излизат от опашката в реда, в който са постъпили в нея.

Упражнения

1. Реализирайте структурата дек. Това е специфична структура, позволяваща елементи да бъдат добавяни и премахвани от двата й края. Нека освен това, елемент поставен от едната страна да може да бъде премахнат само от същата. Реализирайте операции за премахване добавяне и изчистване на дека. При невалидна операция подавайте подходящо изключение.

2. Използвайки опашка реализирайте пълно обхождане на всички директории на твърдия ви диск и ги отпечатвайте на конзолата. Реализирайте алгоритъма "обхождане в ширина" – Breadth-First-Search (BFS) – може да намерите стотици статии за него в Интернет.

3. Използвайки опашка реализирайте пълно обхождане на всички директории на твърдия ви диск и ги отпечатвайте на конзолата. Реализирайте алгоритъма "обхождане в дълбочина" – Depth-First-Search (DFS) – може да намерите стотици статии за него в Интернет.

Решения и упътвания

1. Използвайте два стека с общо дъно. По този начин, ако добавяме елементи отляво на дека ще влизат в левия стек, след което ще могат да бъдат премахнати отново оттам. Аналогично за десния стек.

2. Алгоритъмът е много лесен: започваме от празна опашка, в която слагаме коренната директория (от която стартира обхождането). След това докато опашката не остане празна, изваждаме от нея поредната директория, отпечатваме я и прибавяме към опашката всички нейни поддиректории. По този начин ще обходим файловата система в ширина. Ако в нея няма цикли (както е под Windows), процесът ще е краен.

3. Ако в решението на предната задача заместим опашката със стек, ще получим обхождане в дълбочина. Хитро, нали?

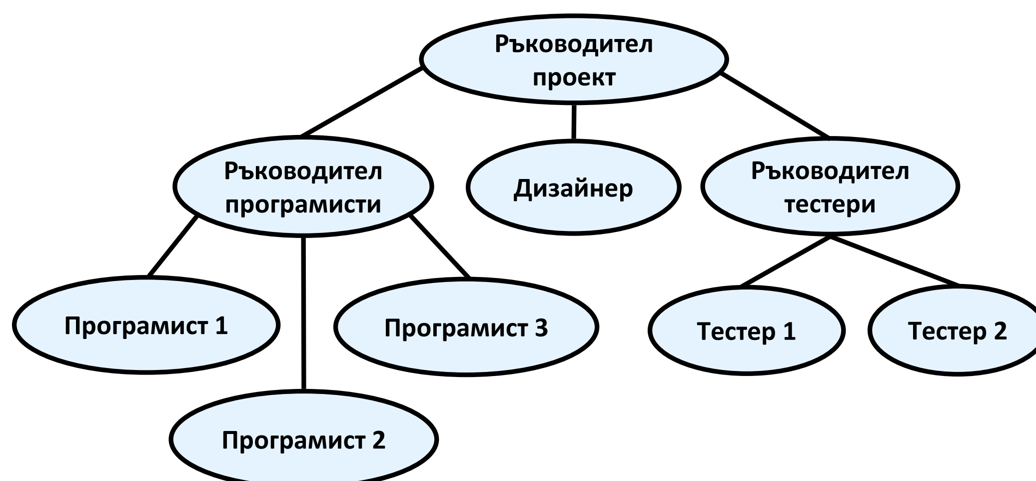
7. Дървета и графи

Дървета

В програмирането дърветата са изключително често използвана структура от данни, защото те моделират по естествен начин всякакви йерархии от обекти, които постоянно ни заобикалят в реалния свят. Нека дадем един пример, преди да изложим терминологията, свързана с дърветата.

Пример – йерархия на участниците в един софтуерен проект

Да вземем за пример един екип, отговорен за изработването на даден софтуерен проект. Участниците в него са взаимно свързани с връзката ръководител-подчинен. Ще разгледаме една конкретна ситуация, в която имаме екип от 9 души:

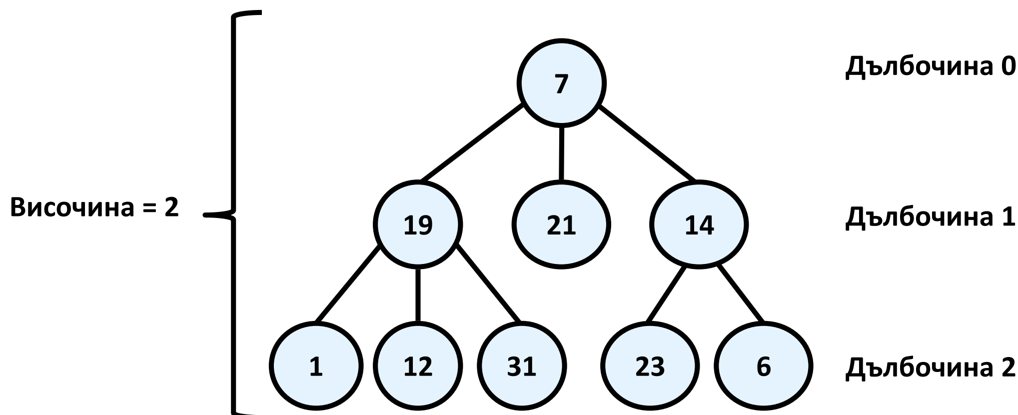


Каква информация можем да извлечем от така изобразената йерархия? Прекият шеф на програмистите е съответно "Ръководител програмисти". "Ръководител проект" е също е техен началник, но непряк, т.е. те отново са му подчинени. "Ръководител програмисти" е подчинен само на "Ръководител проект". От друга страна, ако погледнем "Програмист 1", той няма нито един подчинен. "Ръководител проект" стои най-високо в йерархията и няма шеф.

По аналогичен начин можем да опишем и ситуацията с останалите участници в проекта. Виждаме как една на пръв поглед малка фигура ни носи много информация.

Терминология, свързана с дърветата

За по-доброто разбиране на тази точка силно препоръчваме на читателя да се опита на всяка стъпка да прави аналогия между тяхното абстрактно значение и това, което използваме в ежедневието.



Нека да опростим начина, по който изобразихме нашата йерархия. Можем да приемем, че тя се състои от точки, свързани с отсечки. За удобство, точките ще номерираме с произволни числа, така че после лесно да можем да говорим за някоя конкретна.

Всяка една точка, ще наричаме **връх**, а всяка една отсечка – **ребро**. Върховете "19", "21" и "14" стоят под върха "7" и са директно свързани с него. Тях ще наричаме **преки наследници (деца)** на "7", а "7" – техен **родител (баща)**. Аналогично "1", "12" и "31" са деца на "19" и "19" е техен родител. Съвсем естествено ще казваме, че "21" е **брат** на "19", тъй като са деца на "7" (обратното също е вярно – "19" е брат на "21"). От гледна точка на "1", "12", "31", "23" и "6", "7" е предшестващ ги в йерархията (в случая е родител на техните родители). Затова "7" ще наречем техен **непряк предшественик (дядо, прародител)**, а тях – негови **непреки наследници**.

Корен е върхът, който няма предшественици. В нашия случай той е "7".

Листа са всички върхове, които нямат наследници. В примера – "1", "12", "31", "21", "23" и "6" са листа.

Вътрешни върхове са всички върхове, различни от корена и листата (т.е. всички върхове, които имат както родител, така и поне един наследник). Такива са "19" и "14".

Път ще наричаме последователност от свързани чрез ребра върхове, в която няма повтарящи се върхове. Например последователността "1", "19", "7" и "21" е път. "1", "19" и "23" не е път, защото "19" и "23" не са свързани помежду си с ребро.

Дължина на път е броят на ребрата, свързващи последователността от върхове в пътя. Практически този брой е равен на броят на върховете в пътя минус единица. Дължината на примера ни за път ("1", "19", "7" и "21") е три.

Дълбочина на връх ще наричаме дължината на пътя от корена до дадения връх. На примера ни "7" като корен е с дълбочина нула, "19" е с дълбочина едно, а "23" – с дълбочина две.

И така, ето и дефиницията за това какво е дърво:

Дърво (tree) – рекурсивна структура от данни, която се състои от върхове, които са свързани помежду си с ребра. За дърветата са в сила твърденията:

- Всеки връх може да има 0 или повече преки наследници (деца).
- Всеки връх има най-много един баща. Съществува точно един специален връх, който няма предшественици – коренът (ако дървото не е празно).
- Всички върхове са достижими от корена, т.е съществува път от корена до всички тях.

Можем да дефинираме дърво и по по-прост начин: всеки единичен връх наричаме дърво и той може да има нула или повече наследници, които също са дървета.

Височина на дърво е максималната от дълбочините на всички върхове. В горния пример височината е 2.

Степен на връх ще наричаме броят на преките наследници (деца) на дадения връх. Степента на "19" и "7" е три, докато тази на "14" е две. Листата са от нулева степен.

Разклоненост на дърво се нарича максималната от степените на всички върхове в дървото. В нашият пример степента на върховете е най-много 3, следователно разклонеността на дървото ни е 3.

Графи

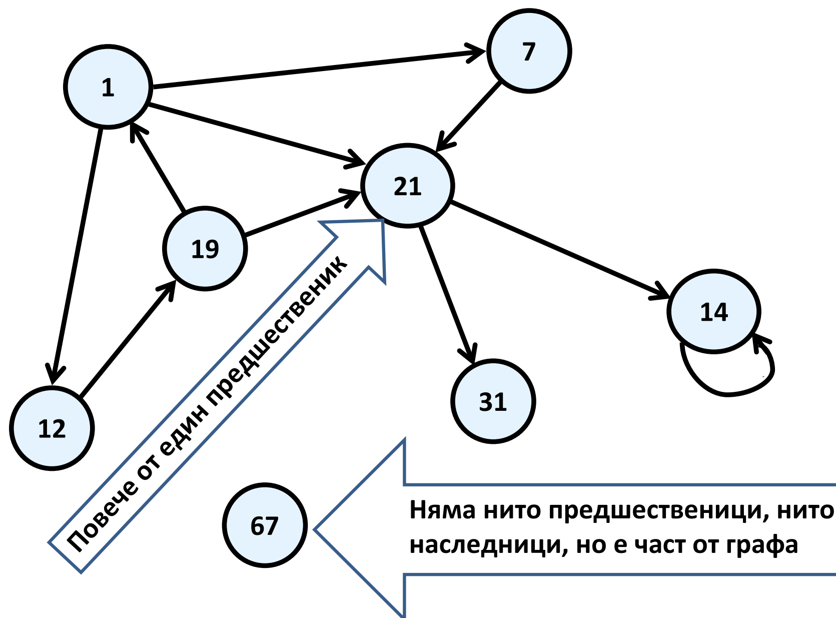
Графите се една изключително полезна и доста разпространена структура от данни. Използват се за описването на най-разнообразни взаимовръзки между обекти от практиката, свързани с почти всичко. Както ще видим по-късно, дървета са подмножество на графите, т.е. графите представляват една обобщена структура, позволяваща моделирането на доста голяма съвкупност от реални ситуации.

Честата употреба на графите в практиката е довела до задълбочени изследвания в "теория на графите", в която са известни огромен брой задачи за графи и за повечето от тях има и добре известно решение.

Графи – основни понятия

В тази точка ще въведем някои от по-важните понятия и дефиниции. Част от тях са аналогични на тези, въведени при структурата от данни дърво, но двете структури, както ще видим, имат много сериозни различия, тъй като дървото е само един частен случай на граф.

Да разгледаме следният примерен граф, чийто тип по-късно ще наречем краен ориентиран. В него отново имаме номерация на върховете, която е абсолютно произволна и е добавена, за да може по-лесно да говорим за някой конкретен:



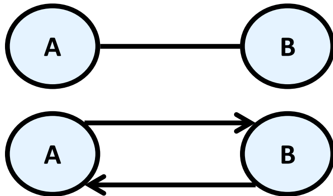
Кръгчетата на схемата, ще наричаме **върхове**, а стрелките, които ги свързват, ще наричаме **ориентирани ребра (дъги)**. Върхът, от който излиза стрелката ще наричаме **предшественик** на този, който стрелката сочи. Например "19" е предшественик на "1". "1" от своя страна се явява **наследник** на "19". За разлика от структурата дърво, сега всеки един връх може да има повече от един предшественик. Например "21" има трима - "19", "1" и "7". Ако два върха са свързани с ребро, то казваме, че тези два върха са **инцидентни** с това ребро.

Следва дефиниция за **краен ориентиран граф (finite directed graph)**:

Краен ориентиран граф се нарича наредената двойката двойка **(V, E)**, където **V** е крайно множество от върхове, а **E** е крайно множество от ориентирани ребра. Всяко ребро ***e*** принадлежащо на **E** представлява наредена двойка от върхове ***u*** и ***v*** т.е. ***e***=(***u***, ***v***), които еднозначно го определят.

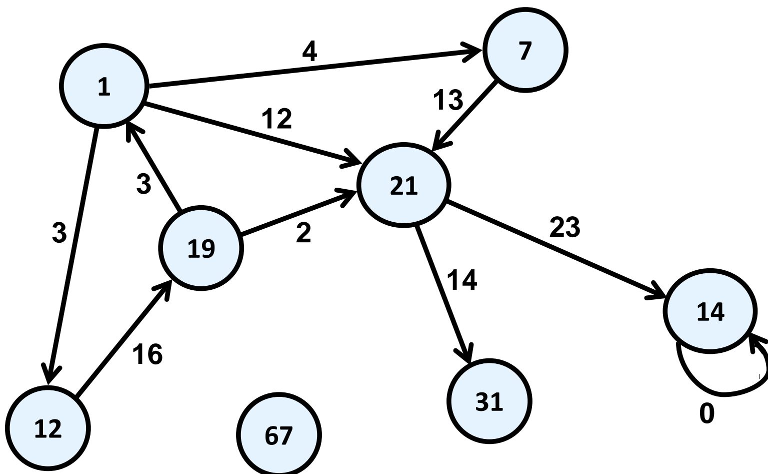
За по-доброто разбиране на тази дефиниция силно препоръчваме на читателя да си мисли за върховете например като за градове, а ориентираните ребра като еднопосочни пътища. Така, ако единият връх е София, а другият е Велико Търново то еднопосочният път (дъгата) ще се нарича София-Велико Търново. Всъщност това е един от класическите примери за приложение на графите – в задачи свързани с пътища.

Ако вместо стрелки върховете са свързани с отсечки (както при структурата дърво), то тогава отсечките ще наричаме **неориентирани ребра**, а графът – **неориентиран**. На практика можем да си представяме, че едно неориентирано ребро от връх А до връх В представлява двупосочно ребро еквивалентно на две противоположни ориентирани ребра между същите два върха:



Два върха свързани с ребро, ще наричаме **съседни**.

За ребрата може се зададе функция, която на всяко едно ребро съпоставя реално число. Тези така получени реални числа ще наричаме **тегла**. Като примери за тегла можем да дадем дължината на директните връзки между два съседни града, пропускателната способност на една тръба и др. Граф, който има тегла по ребрата се нарича **претеглен (weighted)**. Ето как се изобразява претеглен граф:



Път в граф ще наричаме последователност от върхове $v_1, v_2, \dots, v_n$, такава, че съществува ребро от v_i до v_{i+1} за всяко i от 1 до n-1. В нашия граф път е например последователността "1", "12", "19", "21". "7", "21" и "1" обаче не е път, тъй като не съществува ребро започващо от "21" и завършващо в "1".

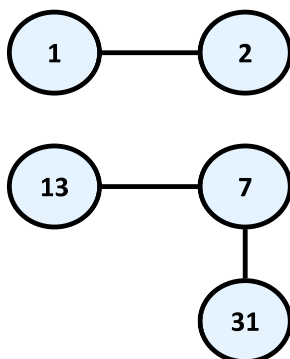
Дължина на път е броят на ребрата, свързващи последователността от върхове в пътя. Този брой е равен на броят на върховете в пътя минус единица. Дължината на примера ни за път "1", "12", "19", "21" е три.

Цена на път в претеглен граф, ще наричаме сумата от теглата на ребрата участващи в пътя. В реалния живот пътят от София до Варна например е равен на дължината на пътя от София до Велико Търново плюс дължината на пътя от Велико Търново до Варна. В нашия пример дължината на пътя "1", "12", "19" и "21" е равна на 3 + 16 + 2 = 21.

Цикъл е път, в който началният и крайният връх на пътя съвпадат. Пример за цикъл е "1", "12" и "19". "1", "7" и "21" обаче не е цикъл.

Примка ще наричаме ребро, което започва от и свършва в един и същ връх. В нашия пример върха "14" има примка.

Свързан неориентиран граф наричаме неориентиран граф, в който съществува път от всеки един връх до всеки друг. Например следният граф не е свързан, защото не съществува път от "1" до "7".



И така, вече имаме достатъчно познания, за да дефинираме понятието дърво по още един начин – като специален вид граф:

Дърво – неориентиран свързан граф без цикли.

Като леко упражнение оставяме на читателя да покаже защо двете дефиниции за дърво са еквивалентни.

Основни приложения и задачи за графи

Графите се използват за моделиране на много ситуации от реалността, а задачите върху графи моделират множество реални проблеми, които често се налага да бъдат решавани. Ще дадем само няколко примера:

- Карта на град може да се моделира с ориентиран претеглен граф. На всяка улица се съпоставя ребро с дължина съответстваща на дължината на улицата и посока – посоката на движение. Ако улицата е двупосочна може да й се съпоставят две ребра за двете посоки на движение. На всяко кръстовище се съпоставя връх. При такъв модел са естествени задачи като търсене на най-кратък път между две кръстовища, проверка дали има път между две кръстовища, проверка за цикъл (дали можем да се завъртим и да се върнем на изходна позиция), търсене на път с минимален брой завои и т.н.

- Компютърна мрежа може да се моделира с неориентиран граф, чиито върхове съответстват на компютрите в мрежата, а ребрата съответстват на комуникационните канали между компютрите. На ребрата могат да се съпоставят различни числа, примерно капацитет на канала или скорост на обмена и др. Типични задачи при такива модели на компютърна мрежа са проверка за свързаност между два компютъра, проверка за двусвързаност между две точки (съществуване на двойно-подсигурен канал, който остава при отказ на който и да е компютър) и др. В частност Интернет може да се моделира като граф, в който се решават задачи за маршрутизация на пакети, които се моделират като задачи за графи.

- Речната система в даден регион може да се моделира с насочен претеглен граф, в който всяка река се състои от едно или няколко ребра, а всеки връх съответства на място, където две или повече реки се вливат една в друга. По ребрата могат да се съпоставят стойности, свързани с количеството вода, което преминава по тях. Естествени при този модел са задачи като изчисление на обемите вода, преминаващи през всеки връх и предвиждане на евентуални наводнения при увеличаване на количествата.

Виждате, че графите могат да имат многобройни приложения. За тях има изписани стотици книги и научни трудове. Съществуват десетки класически задачи за графи, за които има известни решения или е известно, че нямат ефективно решение. Ние няма да се спираме на тях. Надяваме се чрез краткото представяне да събудим интересът ви към темата и да ви подтикнем да отделите достатъчно внимание на задачите за графи от упражненията.

8. Принципи на обектно-ориентираното програмиране

Да си припомним: класове и обекти

Класовете са описание (модел) на реални предмети или явления, наречени същности (entities). Например класът "Ученик".

Класовете имат характеристики – в програмирането са наречени **свойства (properties)**. Например съвкупност от оценки.

Класовете имат и поведение **–** в програмирането са наречени **методи(methods)**. Например явяване на изпит.

Методите и свойствата могат да бъдат видими и невидими – от това зависи дали всеки може да ги използва или са само за вътрешна употреба в рамките на класа.

Обектите (objects) са екземпляри (инстанции) на класовете. Например Иван е Ученик, Петър също е ученик.

Обектно-ориентирано програмиране (ООП)

Обектно-ориентираното програмиране е наследник на процецурното (структурно) програмиране. Процедурното програмиране най-общо казано описва програмите чрез група от преизползваеми парчета код (процедури), които дефинират входни и изходни параметри. Процедурните програми представляват съвкупност от процедури, които се извикват една друга.

Проблемът при процедурното програмиране е, че преизползваемостта на кода е трудно постижима и ограничена – само процедурите могат да се преизползват, а те трудно могат да бъдат направени общи и гъвкави. Няма лесен начин да се реализират абстрактни структури от данни, които имат различни имплементации.

Обектно-ориентираният подход залага на парадигмата, че всяка програма работи с данни, описващи същности (предмети и явления) от реалния живот. Например

една счетоводна програма работи с фактури, стоки, складове, наличности, продажби и т.н.

Така се появяват обектите – те описват характеристиките (свойства) и поведението (методи) на тези същности от реалния живот.

Основни предимства и цели на ООП – да позволи по-бърза разработка на сложен софтуер и по-лесната му поддръжка. ООП позволява по лесен начин да се преизползва кода, като залага на прости и общоприети правила (принципи). Нека ги разгледаме.

Основни принципи на ООП

За да бъде един програмен език обектно-ориентиран, той трябва не само да позволява работа с класове и обекти, но и трябва да дава възможност за имплементирането и използването на принципите и концепциите на ООП: наследяване, абстракция, капсулация и полиморфизъм. Сега ще разгледаме в детайли всеки от тези основни принципи на ООП.

- **Наследяване (Inheritance)**

  Ще обясним за как йерархиите от класове подобряват четимостта на кода и позволяват преизползване на функционалност.

- **Абстракция (Abstraction)**

  Ще се научим да виждаме един обект само от гледната точка, която ни интересува, и да игнорираме всички останали детайли.

- **Капсулация (Encapsulation)**

  Ще се научим да скриваме ненужните детайли в нашите класове и да предоставяме прост и ясен интерфейс за работа с тях.

- **Полиморфизъм (Polymorphism)**

  Ще обясним как да работим по еднакъв начин с различни обекти, които дефинират специфична имплементация на някакво абстрактно поведение.

Наследяване (Inheritance)

Наследяването е основен принцип от обектно-ориентираното програмиране. То позволява на един клас да "наследява" (поведение и характеристики) от друг, по-общ клас. Например лъвът е от семейство котки. Всички котки имат четири лапи, хищници са, преследват жертвите си. Тази функционалност може да се напише веднъж в клас Котка и всички хищници да я преизползват – тигър, пума, рис и т.н.

Как се дефинира наследяване в Java?

Наследяването в Java става с ключовата дума **extends**. В Java и други модерни езици за програмиране един клас може да наследи само един друг клас (**single inheritance**), за разлика от C++, където се поддържа множествено наследяване (**multiple inheritance**). Ограничението е породено от това, че при наследяване на два класа с еднакъв метод е трудно да се реши кой от тях да се използва (при C++ този проблем е решен много сложно). В Java могат да се наследяват множество интерфейси, за които ще говорим по-късно.

Класът, който наследяваме, се нарича **клас-родител** или още **базов клас (base class, super class)**.

Наследяване на класове – пример

Да разгледаме един пример за наследяване на класове в Java. Ето как изглежда базовият (родителски)клас:

Felidae.java

```
package introjavabook;
publicclass Felidae {
privatebooleanmale;
	public Felidae() {
		this(true);
	}
```

```
	public Felidae(boolean male) {
		this.male = male;
	}
	publicboolean isMale() {
		returnmale;
	}
	publicvoid setMale(boolean male) {
		this.male = male;
	}
}
```

Ето как изглежда и класът-наследник **Lion**:

Lion.java

```
package introjavabook;
publicclass Lion extends Felidae {
	privateintweight;
	public Lion(boolean male, int weight) {
		super(male);
		this.weight = weight;
	}
	publicint getWeight() {
		returnweight;
	}
	publicvoid setWeight(int weight) {
		this.weight = weight;
	}
}
```

Конструкторите при наследяване

При наследяване на един клас, нашите конструктори задължително трябва да извикат конструктор на базовия клас, за да може и той да инициализира член-променливите си. Ако не го направим изрично, в началото на всеки наш конструктор компилаторът поставя извикване на базовия конструктор без параметри: **super()**. Ако базовият клас няма конструктор по подразбиране (без параметри), нашите конструктори трябва да извикат изрично някои от другите конструктори на базовия клас. Липсата на изрично извикване предизвиква грешка при компилация.

Конструкторите и super – пример

Разгледайте класа **Lion** от последния пример, той няма конструктор по подразбиране. Да разгледаме следния клас-наследник на **Lion**:

AfricanLion.java

```
package introjavabook;

public class AfricanLion extends Lion {

	public AfricanLion(boolean male, int weight) {
		super(male, weight);
	}
	public String toString() {
		return String.format(
			"(AfricanLion, male: %s, weight: %s)",
			this.isMale(), this.getWeight() );
	}
}
```

Ако закоментирамеили изтрием реда "**super(male, weight);**", класът **AfricanLion** няма да се компилира. Опитайте.

	Извикването на конструктор на базов клас трябва винаги да е на първия ред от нашия конструктор. Иначе компилаторът дава грешка. Идеята е полетата на базовия клас да бъдат инициализирани преди да започнем да инициализираме полета в класа-наследник, защото може те да разчитат на някое поле от базовия клас.

Класът Object

Появата на обектно-ориентираното програмиране de facto става популярно сезика C++. В него често се налага да се пишат класове, които трябва да работят с обекти от всякакъв тип. В C++ този проблем се решава по начин, който не се смята за много обектно-ориентиран стил (чрез използване на указатели).

Архитектите на Java поемат в друга посока. Те създават клас, който всички други класове пряко или косвено да наследяват и до който всеки обект може да бъде преобразуван. В този клас е удобно да бъдат сложени важни методи и тяхната имплементация по подразбиране. Този клас се нарича **Object**.

В Java всеки клас, който не наследява друг клас изрично, наследява системния клас **java.lang.Object** по подразбиране. За това се грижи компилаторът. Всеки клас, който наследява друг клас, наследява индиректно **Object** от него. Така всеки клас явно или неявно наследява **Object** и има в себе си всички негови методи и полета.

Благодарение на това свойство всеки обект може да бъде преобразуван до **Object**. Типичен пример за ползата от неявното наследяване на **Object**е при колекциите, които разгледахме в главите за структури от данни. Списъчните структури (например **ArrayList**) могат да работят с всякакви обекти, защото ги разглеждат като инстанции на класа **Object**.

Java, стандартните библиотеки и Object

В Java има много предварително написани класове (вече разгледахме доста от тях в главите за колекции, текстови файлове и символни низове). Тези класове са

част от Java платформата – навсякъде, където има Java, ги има и тях. Тези класове се наричат **стандартниклас-библиотеки – standard class libraries**.

Java е първата платформа, която идва с такъв богат набор от предварително написани класове. Голяма част от тях работят с **Object**, за да могат да бъдат използвани на възможно най-много места.

В Java има и доста библиотеки, които могат да се добавят допълнително и съвсем логично се наричат просто клас-библиотеки или още външни библиотеки.

Object – пример

Нека разгледаме класа **Object** с един пример:

ObjectExample.java

```
package introjavabook;
publicclass ObjectExample {
        publicstaticvoid main(String... args) {
                AfricanLion africanLion = new AfricanLion();
                Object object = africanLion;
        }
}
```

В този пример преобразувахме един **AfricanLion** в **Object**. Тази операция се нарича **upcasting**и е позволена, защото **AfricanLion** е непряк наследник на класа **Object**.

Методът Object.toString()

Един от най-използваните методи, идващи от класа **Object,** е **toString()**. Той връща текстово представяне на обекта. Всеки обект има такъв метод и следователно всеки метод има текстово представяне. Този метод се използва, когато отпечатваме обект чрез **System.out.println()**.

Object.toString() – пример

Ето един пример, в който извикваме **toString()** метода:

ToStringExample.java

```
package introjavabook;
publicclass ToStringExample {
    publicstaticvoid main(String... args) {
        AfricanLion africanLion = new AfricanLion();
        System.out.println(africanLion.toString());
    }
}
```

Тъй като **AfricanLion** не пренаписва (override) метода **toString()**, в конкретния случай се извиква имплементацията от базовия клас. **Lion** и **Felidae** също не пренаписват този метод, следователно се извиква имплементацията, наследена от класа **java.lang.Object**. В резултата, който виждаме по-горе, се съдържа пакетът на обекта, името на класа, както и странна стойност след @ знака. Това всъщност е хеш кодът на обект в шестнайсетична бройна система. Това не е адресът в паметта, а някаква друга стойност. Обикновено тази стойност е различна за различните оебкти.

Ето я и оригиналната имплементация на метода **Object.toString()**, извадена от сорс кода на стандартните библиотеки в Java:

Object.java

```
Publicclass Object {
    public String toString() {
        return getClass().getName() +
            "@" + Integer.toHexString(hashCode());
    }
```

```
}
```

Абстракция (Abstraction)

Следващият основен принцип от обектно-ориентираното програмиране, който ще разгледаме, е "абстракция". **Абстракцията** означава да работим с нещо, което знаем как да използваме, но не знаем как работи вътрешно. Например имаме телевизор. Не е нужно да знаем как работи телевизорът отвътре, за да го ползваме. Нужно ни е само дистанционното, и с малък брой бутони (интерфейс на дистанционното) можем да гледаме телевизия.

Същото се получава и с обектите в ООП. Ако имаме обект **Лаптоп** и той се нуждае от процесор, просто използваме обекта **Процесор**. Не знаем (или по-точно не се интересуваме) как той смята вътрешно. За да го използваме, е достатъчно да извикваме метода **сметни()** с подходящи параметри.

Абстракцията е нещо, което правим всеки ден. Това е действие, при което игнорираме всички детайли, които не ни интересуват от даден обект и разглеждаме само детайлите, които имат значение за проблема, който решаваме. Например в хардуера съществува абстракция "устройство за съхранение на данни", което може да бъде твърд диск, USB memory stick, флопи диск или CD-ROM устройство. Всяко от тях работи вътрешно по различен начин, но от гледна точка на операционната система и на програмите в нея те се използват по еднакъв начин – на тях се записват файлове и директории. В Windows имаме Windows Explorer и той умее да работи по еднакъв начин с всички устройства, независимо дали са твърд диск или USB stick. Той работи с абстракцията "устройство за съхранение на данни" (storage device) и не се интересува как точно данните се четат и пишат. За това се грижат драйверите за съответните устройства. Те се явяват конкретни имплементации на интерфейса "устройство за съхранение на данни".

Абстракцията е една от най-важните концепции в програмирането и в ООП. Тя ни позволява да пишем **код, който работи с абстрактни структури от**

данни(например списък, речник, множество и други). Имайки абстрактния тип данни ние можем да работим с него през неговия интерфейс, без да се интересуваме от имплементацията му. Например можем да запазим във файл всички елементи на списък, без да се интересуваме дали той е реализиран с масив, чрез свързана имплементация или по друг начин. Този код остава непроменен, когато работим с различни конкретни типове данни. Дори можем да пишем нови типове данни (които се появяват на по-късен етап) и те да работят с нашата програма, без да я променяме.

Абстракцията ни позволява и нещо много важно – **да дефинираме интерфейс на нашите програми**, т.е. да дефинираме всички задачи, които тази програма може да извърши, както и съответните входни и изходни данни. Така можем да направим няколко по-малки програми, всяка от които да извършва някаква по-малка задача. Това, допълнено от факта, че можем да работим с абстрактни данни, ни дава голяма гъвкавост при свързването на тези по-малки програми в една по-голяма и ни дава повече възможности за преизползване на код. Тези малки подпрограми се наричат компоненти. Този начин на писане на програми намира широко приложение в практиката, защото ни позволява не само да преизползваме обекти, а дори цели подпрограми.

Абстракция – пример за абстрактни данни

Ето един пример, в който дефинираме конкретен тип данни "африкански лъв", но след това го използваме по абстрактен начин – чрез абстракцията "лъв". Тази абстракция не се интересува от детайлите на всички видове лъвове.

AbstractDataExample.java

```
package introjavabook;
publicclass AbstractDataExample {
	publicstaticvoid main(String... args) {
		Lion lion = new Lion(true, 150);
		Felidae bigCat1 = lion;
```

```
            AfricanLion africanLion = new AfricanLion();
            Felidae bigCat2 = africanLion;
        }
}
```

Интерфейси

В езика Java **интерфейсът** е дефиниция на роля (на група абстрактни действия). Той дефинира какво поведение трябва да има един обект, без да указва как точно се реализира това поведение.

Един обект може да има много роли (да имплементира много интерфейси) и ползвателите му могат да го използват от различни гледни точки.

Например един обект **Човек** може да има ролите **Военен** (с поведение "стреляй по противника"), **Съпруг** (с поведение "обичай жена си"), **Данъкоплатец** (с поведение "плати си данъка"). Всеки човек обаче имплементира това поведение по различен начин: **Иван** си плаща данъците навреме, **Георги** – не навреме, **Петър** – въобще не ги плаща.

Някой може да попита защо най-базовият за всички обекти клас **Object** не е всъщност интерфейс. Причината е, че тогава всеки клас щеше да трябва да имплементира група методи, а това би отнемало излишно време. Оказва се, че и не всеки клас има нужда от специфична реализация, тоест имплементацията по подразбиране върши работа в повечето случаи. От класа **Object** не е нужно да се пренапише (повторно имплементира) никой метод, но ако се наложи, това може да се направи. Пренаписването на методи е обяснено в детайли след малко.

Интерфейси – ключови понятия

В интерфейса може да има само декларации на методи и константи.

Декларация на метод (method declaration) е съвкупността от връщания тип на метода + сигнатурата на метода. Връщаният тип е просто за яснота какво ще върне метода.

Сигнатура на метод (method signature) е съвкупността от името на метода + описание на параметрите (тип и последователност). В един клас/интерфейс всички методи трябва да са с различни сигнатури и да не съвпадат със сигнатури на наследени методи.

	Това, което идентифицира един метод, е неговата сигнатура. Връщаният тип не е част нея. Причината е, че ако два метода се различават само по връщания тип (например два класа, които се наследяват един друг), то не може еднозначно да се идентифицира кой метод трябва да се извика.

Имплементация на клас/метод (class/method implementation) е тялото със сорс код на класа/метода. Най често е заключено между скобите **{** и **}**. При методите се нарича още **тяло на метод**.

Интерфейси – пример

Интерфейсът в Java се дефинира с ключовата думичка **interface**. В него може да има само декларации на методи, както и статични променливи (за константи например).Ето един пример за интерфейс:

Reproducible.java

```
package introjavabook;
publicinterface Reproducible {
	Mammal[] reproduce(Mammal mate);
}
```

Ето как изглежда и класа **Lion**, който имплементира интерфейса **Reproducible**:

Lion.java

```
package introjavabook;
publicclass Lion extends Felidae implements Reproducible {
	public Mammal[] reproduce(Mammal anotherLion) {
```

```
                return new Mammal[]{new Lion(), new Lion()};
        }
}
```

В интерфейса методите само се декларират, имплементацията е в класа, който имплементира интерфейса - **Lion**.

Класът, който имплементира даден интерфейс, трябва да имплементира всеки метод от него. Изключение – ако класът е абстрактен, тогава да имплементира нула, няколко или всички методи. Всички останали методи имплементират в някой от класовете наследници.

Капсулация (Encapsulation)

Капсулацията е един от основните принципи на обектно-ориентираното програмиране. Тя се нарича още "скриване на информацията" (**information hiding**). Един обект трябва да предоставя на ползвателя си само необходимите средства за управление. Една **Секретарка** ползваща един **Лаптоп** знае само за екран, клавиатура и мишка, а всичко останало е скрито. Тя няма нужда да знае за вътрешността на **Лаптопа**, защото не й е нужно и може да оплеска нещо. Тогава част от свойствата и методите остават скрити за нея.

Изборът какво е скрито и какво е публично видимо е на този, който пише класа. Когато програмираме трябва да дефинираме като private (скрит) всеки метод или поле, които не ползваме от друг клас.

Капсулация – пример

Ето един пример за скриване на методи, които не е нужда да са известни на потребителя, а се ползват вътрешно само от автора на класа. Първо дефинираме абстрактен клас **Felidae**, който дефинира публичните операции на котките (независимо какви точно котки имаме):

Felidae.java

```java
package introjavabook;

publicabstractclass Felidae {

	publicabstractvoid walk();

}
```

Ето как изглежда класът **Lion**:

Lion.java

```java
package introjavabook;

publicclass Lion extends Felidae implements Reproducable {

	private movePaw(Paw paw) {

	}

	publicvoid walk() {

		this.movePaw(frontLeft);

		this.movePaw(frontRight);

		this.movePaw(bottomLeft);

		this.movePaw(bottomRight);

	}

}
```

Публичният метод **walk()** извиква 4 пъти някакъв друг скрит (**private**) метод. Така интерфейсът (в този случай абстрактният клас) е кратък – само един метод. Имплементацията обаче извиква друг метод, също част от имплементацията, но скрит за ползвателя на класа. Така класът **Lion** не разкрива публично информация за това как работи вътрешно и това му дава възможност на по-късен етап да промени имплементацията си без останалите класове да разберат (и да имат нужда от промяна).

Полиморфизъм (Polymorphism)

Следващият основен принцип от обектно-ориентираното програмиране е "полиморфизъм". **Полиморфизмът** позволява третирането на обекти от

наследен клас като обекти от негов базов клас. Например големите котки (базов клас) хващат жертвите си (метод) по различен начин. Лъвът (клас наследник) ги дебне, докато Гепардът (друг клас-наследник) просто ги надбягва.

Полиморфизмът дава възможността да третираме произволна голяма котка просто като голяма котка и да кажем "хвани жертвата си", без значение каква точно е голямата котка.

Полиморфизмът може много да напомня на абстракцията, но в програмирането се свързва най-вече с пренаписването (override) на методив наследените класове с цел промяна на оригиналното им поведение, наследено от базовия клас. Абстракцията се свързва със създаването на интерфейс на компонент или функционалност (дефиниране на роля). Пренаписването на методи ще разгледаме в детайли след малко.

Обектно-ориентирано моделиране (ООМ)

Нека приемем, че имаме да решаваме определен проблем или задача. Този проблем идва обикновено от реалния свят. Той съществува в дадена реалност, която ще наричаме заобикаляща го среда.

Обектно-ориентираното моделиране (ООМ) е процес, свързан с ООП, при който се изваждат всички обекти, свързани с проблема, който решаваме (създава се модел). Изваждат се само тези техни характеристики, които са свързани с решаването на конкретния проблем. Останалите се игнорират. Така вече си създаваме нова реалност, която е опростена версия на оригиналната (неин модел), и то такава, че ни позволява да си решим проблема или задачата.

Например, ако моделираме система за продажба на билети, за един пътник важни характеристики биха могли да бъдат неговото име, неговата възраст, дали ползва намаление и дали е мъж или жена (ако продаваме спални места). Пътникът има много други характеристики, които не ни интересуват, примерно какъв цвят са му очите, кой номер обувки носи, какви книги харесва или каква бира харесва.

При моделирането се създава опростен модел на реалността с цел решаване на конкретната задача. При обектно-ориентираното моделиране моделът се прави със средствата на ООП: чрез класове, атрибути на класовете, методи в класовете, обекти, взаимоотношения между класовете и т.н. Нека разгледаме този процес в детайли.

Стъпки при обектно-ориентираното моделиране

Обектно-ориентираното моделиране обикновено се извършва в следните стъпки:

- Идентификация на класовете.
- Идентификация на атрибутите на класовете.
- Идентификация на операциите върху класовете.
- Идентификация на връзките между класовете.

Ще разгледаме кратък пример, с който ще ви покажем как могат да се приложат тези стъпки.

Идентификация на класовете

Нека имаме следната извадка от заданието за дадена система:

На потребителя трябва да му е позволено да описва всеки продукт по основните му характеристики, включващи име и номер на продукта. Ако бар-кодът не съвпада с продукта, тогава трябва да бъде генерирана грешка на екрана за съобщения. Трябва да има дневен отчет за всички транзакции, специфицирани в секция 9.

Ето как идентифицираме ключовите понятия:

На **потребителя** трябва да му е позволено да описва всеки **продукт** по основните му **характеристики**, включващи **име** и **номернапродукта**. Ако **бар-кодът** не съвпада с продукта, тогава трябва да бъде генерирана **грешка** на **екраназасъобщения**. Трябва да има **дневенотчет** за всички **транзакции**, специфицирани в секция

9.

Току-що идентифицирахме класовете, които ще ни трябват. Имената на класовете са съществителните имена в текста, най-често нарицателни в единствено число, например **Студент**, **Съобщение**, **Лъв**. Избягвайте имена, които не идват от текста, примерно: **СтраненКлас**, **АдресКойтоИмаСтудент**.

Понякога е трудно да се прецени дали някой предмет или явление от реалния свят трябва да бъде клас. Например адресът може да е клас **Address** или символен низ. Колкото по-добре проучим проблема, толкова по-лесно ще решим кое трябва да е клас. Когато даден клас стане прекалено голям и сложен, той трябва да се декомпозира на няколко по-малки класове.

Идентификация на атрибутите на класовете

Класовете имат атрибути (характеристики), например: класът **Student** има име, учебно заведение и списък от курсове. Не всички характеристики са важни за софтуерната система. Например: за класа **Student** цвета на очите е несъществена характеристика. Само съществените характеристики трябва да бъдат моделирани.

Идентификация на операциите върху класовете

Всеки клас трябва да има ясно дефинирани отговорности – какви обекти или процеси от реалния свят представя, какви задачи изпълнява. Всяко действие в програмата се извършва от един или няколко метода в някой клас. Действията се моделират с операции (методи).

За имената на методите се използват глагол + съществително. Примери: **PrintReport()**, **ConnectToDatabase()**. Не може веднага да се дефинират всички методи на даден клас. Дефинираме първо най-важните методи – тези, които реализират основните отговорности на класа. С времето се появяват още допълнителни методи.

Идентификация на връзките между класовете

Ако един ученик е от определено училище и за задачата, която решаваме, това е важно, тогава ученик и училище са свързани. Тоест класът Училище **има списък от** Ученици**.**

Упражнения

1. Дефинирайте клас **Human** със свойства "собствено име" и "фамилно име". Дефинирайте клас **Student**, наследяващ **Human**, който има свойство "оценка". Дефинирайте клас **Worker**, наследяващ **Human**, със свойства "надница" и "изработени часове". Имплементирайте и метод "изчисли надница за 1 час", който смята колко получава работникът за 1 час работа, на базата на надницата и изработените часове. Напишете съответните конструктори и методи за достъп до полетата (свойства).
2. Инициализирайте масив от 10 студента и ги сортирайте по оценка в нарастващ ред. Използвайте Java интерфейса **java.lang.Comparable**.
3. Инициализирайте масив от 10 работника и ги сортирайте по заплата в намаляващ ред.
4. Дефинирайте клас **Shape** със само един метод **calculateSurface()** и полета **width** и **height**. Дефинирайте два нови класа за триъгълник и правоъгълник, които имплементират споменатия виртуален метод. Този метод трябва да връща площта на правоъгълника (**height*width**) и триъгълника (**height*width/2**). Дефинирайте клас за кръг с подходящ конструктор, при когото при инициализация и двете полета (**height** и **width**) са с еднаква стойност (радиуса), и имплементирайте виртуалния метод за изчисляване на площта. Направете масив от различни фигури и сметнете площта на всичките в друг масив.

Решения и упътвания

1. Задачата е тривиална. Просто следвайте условието и напишете кода.

2. Имплементирайте **Comparable** в **Student** и оттам просто сортирайте списък от **Comparable**. Можете да използвате и **java.util.Arrays. sort(Object[])**.
3. Задачата е като предната.
4. Имплементирайте класовете както са описани в условието на задачата. Тествайте решението си.

9. Заключение.

Считаме, че поставените цели са изпълнени. В настоящия си вид дипломната работа представлява съвременно помагало за обучение по обектно-ориентирано програмиране с **Java.** Личният принос на автора се състои в адаптацията на учебното съдържание към средата на **Java** и в съставянето на конкретни задачи и упражнения след срочните единици.