11. Обектно ориентирано програмиране (на базата на C++ или Java): Основни принципи. Класове и обекти. Конструктори и деструктори. Оператори. Производни класове и наследяване.

1. Основни принципи на обектно ориентираното програмиране.

Четирите основни принципа на ООП са капсулация на данните, абстракция, наследяване и полиморфизъм. Те са основните характеристики, които определят един език за програмиране като обектно-ориентиран.

Капсулация на данните

Основна концепция в ООП е обектът да се разглежда като "черна кутия" – използващите обекта "виждат" само атрибутите и операциите, които са присъщи на обекта от реалния свят, без да се интересуват от конкретната им реализация – клиентът на обекта трябва да знае само какво може да прави обектът, а не как го прави. В такъв смисъл "капсулация" означава скриване на ненужните детайли за обектите и откриване към външния свят само на важните техни характеристики и свойства. Обектите в ООП съдържат своите данни и средствата за тяхната обработката, капсулирани като едно цяло.

Абстракция

Абстракцията опростява сложната реалност чрез моделиране на класове подходящи за разрешаването на определен проблем. Тя представлява модел, изглед или някакво друго общо представяне на действителен обект. Абстракцията е разработване на програмен обект представящ обект от реалния свят, а капсулацията му скрива детайлите на имплеметацията. Абстракцията на данните налага ясно разделяне между абстрактните свойства на съответния тип и конкретните детайли на представянето. Идеята е, че промени в това представяне не трябва да имат влияние върху код използващ абстракцията. Нека разгледаме един прост пример: имаме клас Person съответно със свойства Height, Weight, Age и т.н. и действия, които може да се извършват с него: Run(), Sleep(), Cry(). Абстрактното представяне на Person се определя от свойствата които притежава и от действията, които могат да се извършват върху него и от информацията, която може да получим или предадем на него. Конкретната имплементация на действията и свойствата на обекта остава скрита от външния свят, който няма достъп до нея.

Наследяване

Ако един обект съдържа всички свойства и действия на друг, първият може да го наследи. По този начин наследеният обект освен собствените си атрибути и операции приема и тези на "родителя" си (базовия клас), като така се избягва повторното им дефиниране и се позволява създаването на йерархии от класове, моделиращи по естествен начин зависимостите от реалността. За да изясним това понятие, ще си послужим с класическият в ООП пример за класа от обекти Animal, който представлява абстракция за множеството от всички животни. Всички обекти от този клас имат общи характеристики (например цвят и възраст) и обща функционалност, например операциите Eat и Sleep, докато за класът Dog, представляващ множеството от всички кучета, които също са животни, би могъл да предоставя операциите Eat, Sleep и Bark. Удачно е класът Dog да наследи Animal – тогава той ще съдържа описание само на собственото си действие Bark, докато тези на Eat и Sleep ще получи от базовия си клас. Чрез наследяването се постига специализация, или конкретизация на класовете, тъй като базовият клас представлява категория от обекти по-обща от тази на наследяващите го. Ако си послужим с горния пример, множествата на кучетата и котките са подмножества на множеството от всички животни.

Полиморфизъм

Полиморфизъм буквално означава приемането на различни форми от един обект. Нека е даден базов клас, представящ категория от обекти, които реализират общо действие, което се наследява от множество класове, описващи по-тесни категории. Въпреки, че те всички споделят това действие, те могат да го реализират по различен начин. Когато разполагаме с обекти от базовия клас, знаем че всички те реализират това действие, независимо на кой наследен клас принадлежат. Поради това можем да го използваме без да се интересуваме от конкретната му реализация. Например, ако класът Animal предоставя действието Talk и разгледаме наследяващите го класове Dog и Cat, всеки от тях го реализира по конкрете начин. И ако имаме животно, което издава звук и то е куче – ще лае, а ако е котка – ще мяучи. Полиморфизмът позволява унифицираното извършване на действие над различни обекти, които го реализират. В този случай издаването на звук от животно е полиморфно действие – такова, което се реализира по различен начин в различните наследници на базовия клас.

2. Класове. Дефиниране на класове. Област на класове. Обекти.

Класове

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).

Дефиниране на класове

Класовете осигуряват механизми за създаване на напълно нови типове данни, които могат да бъдат интегрирани в езика, а също за обогатяване възможностите на вече съществуващи типове. Дефинирането на един клас се състои от две части: декларация на класа и дефиниция на неговите член-функции (методи).

Декларация на клас

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

```
public class 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.

```
public class Dog {
      // … Here the class body comes …
}
```

Дефиниция на метод на клас

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

```
public Dog() {
	// … Some code …
}
```

- **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.

```
// Field/Property-storage definition
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.

```
// Field/Property-storage definition
private String name;
```

- **Методи (methods)** – 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.

Пример

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

Dog.java

```
// Class declaration
class Dog { // Openening brace of the class body

	// Property-field definition
	private String name;

	// Constructor definition
	public Dog() {
		this.name = "Sharo";
	}

	// Constructor definition
	public Dog(String name) {
		this.name = name;
	}

	// Property getter-method definition
	public String getName() {
		return this.name;
```

```
	}

	// Property setter-method definition
	public void setName(String name) {
		this.name = name;
	}

	// Method definition
	public void bark() {
		System.out.printf("Dog %s said: Wow-wow!%n", name);
	}
} // Closing brace of the class body
```

Област на класовете

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. До момента описвахме класа като външен, сега ще покажем другите три вида класове: вътрешни, локални и анонимни.

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 **вътрешен клас** (**inner class**, **nested class**). Нека разгледаме тази възможност с един пример:

OuterClass.java

```
public class OuterClass {
	private String name;

	private OuterClass(String name) {
		this.name = name;
	}

	private class InnerClass {
		private String name;

		private InnerClass(String name) {
			this.name = name;
		}

		private void printNames() {
			System.out.println("Inner name: " + this.name);
			System.out.println("Outer name: " +
				OuterClass.this.name);
		}
	}

	public static void main(String[] args) {
```

```
            OuterClass outerClass = new OuterClass("outer");
            InnerClass innerClass = outerClass.newInnerClass("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 и без 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** (**localclasses**). 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

```
public class LocalClassExample {
      public static void main(String[] args) {
            final int 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

```
public class AnnonymousClassExample {
	public static void 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 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.
```

Обекти

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.

3. Конструктори. Дефиниране на конструктори. Видове конструктори.

Конструктори

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

Дефиниране 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

```
public class MyClass {

	// LEGAL constructor
	public MyClass() {
	}

	// Misleading method – has return type
	String MyClass() {
		return "MyClass() method has finished successfully.";
	}

	public static void main(String[] args) {
		MyClass instance = new MyClass();

		// Calling the tricky method…
		System.out.println(instance.MyClass());
	}
}
```

Списък с параметри

По подобие на методите, ако за създаването на обекта, са необходими допълнителни данни, конструкторът ги получава чрез списък от параметри – **<parameters_list>**. В примерния конструктор на класа **Dog**, няма нужда от допълнителни данни за създаване на обект от такъв тип и затова няма деклариран списък от параметри.

Разбира се след декларацията на конструктора, следва неговото тяло, което е като тялото на всеки един метод в Java.

Модификатори

Забелязваме, че в декларацията на конструктора, може да се добавят модификатори –**<modifiers>**. За модификаторите, които познаваме и които не са модификатори за достъп, т.е.**final** и **static**, трябва да кажем, че не са позволени за употреба при декларирането на конструктори.

Видове конструктори

В Java има 3 основни вида конструктори: по подразбиране, за „общо ползване" и за копиране. Друг начин, по който може да разглеждаме конструкторите е: с параметри и без параметри.

Конструктор по подразбиране и конструктор без параметри

Нека разгледаме следния въпрос – какво става, ако не декларираме конструктор в нашия клас? Как ще създадем обекти от този тип?

Когато не декларираме нито един конструктор, компилаторът ще създаде един за нас и той ще се използва при създаването на обекти от типа на нашия клас. Този конструктор се нарича **конструктор по подразбиране** (**implicit constructor**).

Когато не дефинираме нито един конструктор в даден клас, компилаторът ще създаде един, наречен конструктор по подразбиране.

Например, декларираме класа **Collar**, без да декларираме никакъв конструктор в него:

Collar.java

```
public class Collar {


	private int size;


	public int getSize() {
		return this.size;
	}
}
```

Въпреки това ще можем да създадем обекти от тип, този клас, по следния начин:

```
Collar collar = new Collar();
```

Конструкторът по подразбиране изглежда по следния начин:

```
<class_access_level> <class_name>()
```

Трябва да знаем, че конструкторът по подразбиране винаги носи името на класа **<class_name>**, винаги списъкът му с параметри е празен и винаги нивото му на достъп съвпада с нивото на достъп на класа **<class_access_level>**.

Конструкторът по подразбиране е винаги без параметри.

За да се уверим, че конструктора по подразбиране винаги е без параметри, нека направим опит да извикаме подразбиращия се конструктор, като му подадем параметри:

```
Collar collar = new Collar(5);
```

Компилаторът ще изведе следното съобщение за грешка:

```
The constructor Collar(int) is undefined.
```

Работа на конструктора по подразбиране

Както се досещаме, единственото, което конструктора по подразбиране ще направи при създаването на обекти от нашия клас, е да задели памет за полетата на класа ни (които не са статични) и да ги инициализира с подразбиращите се стойности. Например, ако в класа **Collar**не сме декларирали нито един конструктор и създадем обект от него и се опитаме да отпечатаме стойността в полето **size**:

```
public static void main(String[] args) {

	Collar collar = new Collar();

	System.out.println("Collar's size is: " + collar.getSize());

}
```

Резултатът ще бъде:

```
Collar's size is: 0
```

Виждаме, че стойността, която е запазена в полето **size** на обекта **collar**, е точно стойността по подразбиране.

Разлика между конструктор по подразбиране и конструктор без параметри

Трябва да знаем, че ако декларираме поне един конструктор в един клас, тогава компилаторът няма да създаде конструктор по подразбиране.

За да проверим това, нека разгледаме следния пример:

```
public Collar(int size) {

	this();

	this.size = size;

}
```

Нека това е единственият конструктор на класа **Collar**. В него се опитваме да извикаме конструктор без параметри, надявайки се, че компилаторът ще е създал конструктор по подразбиране за нас (който знаем, че е без параметри). След като се опитаме да компилираме, ще разберем, че това, което се опитваме да направим, е

невъзможно. След като сме декларирали дори един единствен конструктор, компилаторът няма да създаде конструктор по подразбиране за нас:

```
The constructor Collar() is undefined.
```

Преди да приключим със секцията за конструкторите, нека кажем нещо много важно:

Въпреки че конструкторът по подразбиране и този, без параметри, си приличат по сигнатура, те са напълно различни.

Конструкторът по подразбиране се създава от компилатора, ако не декларираме нито един конструктор в нашия клас, а конструкторът без параметри го декларираме ние. Освен това конструкторът по подразбиране винаги ще има нивото на достъп, което има класа. Нивото на достъп на конструктора без параметри зависи отново от нас – ние го определяме.

Конструктор за общо ползване

В Java конструкторите могат да се задават и с параметри. Конструктор с параметри съответстващи на клас данните се нарича конструктор за общо ползване. Обикновено конструкторите с непълен брой параметри викат конструктора за общо ползване и му задават deafult-ни стойности на недефинираните параметри.

Конструктор за копиране

Конструкторът за копиране има за аргумент само референция към обект от същия клас. Той е специален конструктор и се създава за да направи копие на обект.

Пример

Time.java

```
public class Time {

	private int hour;
	private int minute;
	private int second;

	// Конструктор по подразбиране/без параметри
	public Time() {
	   // Извиква се конструктор за общо ползване
	   this(0, 0, 0);
	}

	// Конструктор с един параметър
	public Time(int h) {
	   // Извиква се конструктор за общо ползване
	   this(h, 0, 0);

	}
	// Конструктор с два параметъра
	public Time(int h, int m) {
	   // Извиква се конструктор за общо ползване
	   this(h, m, 0);
```

```
    }
    // Конструктор за общо ползване
    public Time(int h, int m, int s) {
        // Вика setTime за валидиране на данните
        setTime(h, m, s);

    }
    // Конструктор за копиране
    public Time(Time time) {
        // Извиква се конструктор за общо ползване
        this(time.getHour(), time.getMinute(), time.getSecond());

    }

    …

}
```

4. Указатели към обекти. Масиви и обекти. Динамични обекти.

Указатели към обекти на класове

Два указателя към обекти (референции) могат да сочат към един и същи обект. Това е показано в следния програмен код:

```
String str = "Some text";
String anotherStr = str;
```

След изпълнението на този код, двете променливи **str** и **anotherStr** ще сочат към един и същи обект (обект **String** със стойност "Some text").

Променливите от тип референция към обект могат да бъдат проверени, дали сочат към един и същ обект, посредством оператора за сравнение **==**. Този оператор не сравнява съдържанието на обектите, а само дали се намират на едно и също място в паметта, т. е. дали са един и същ обект. За променливи от тип обект, не са приложими сравненията по големина (**<**, **>**, **<=** и **>=**).

За да разберем разликата между сравнение на референции към обекти (адреси на обекти в паметта) и сравнение на стойности на обекти, нека следния програмен код:

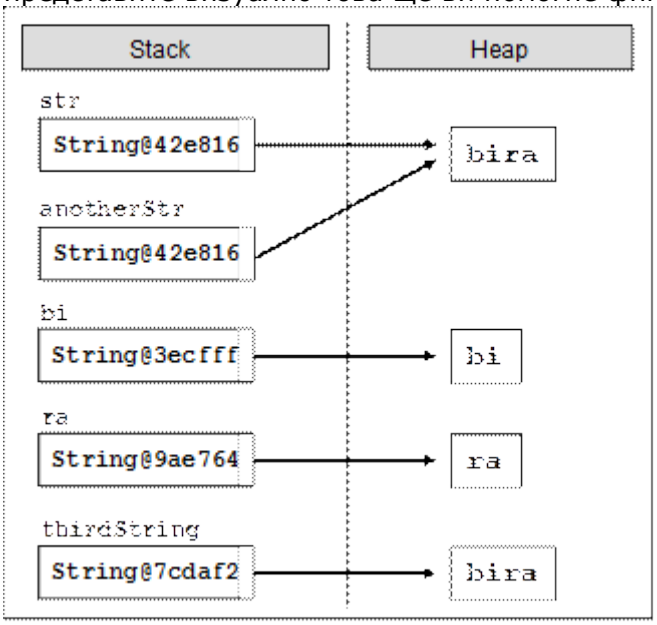
```
String str = "bira";
String anotherStr = str;
String bi = "bi";
String ra = "ra";
String thirdStr = bi  + ra;

System.out.println("str = " + str);
System.out.println("anotherStr = " + anotherStr);
```

```
System.out.println("thirdStr = " + thirdStr);
System.out.println(str == anotherStr);
System.out.println(str == thirdStr);
```

Ако изпълним примера, ще получим следния резултат:

```
str = bira
anotherStr = bira
thirdStr = bira
true
false
```

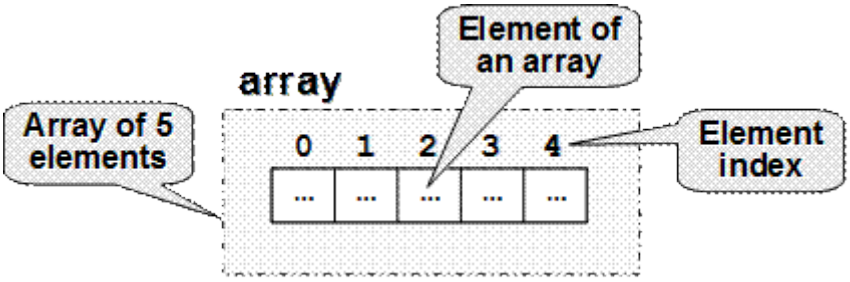
В примера се създават три променливи, които съдържат една и съща стойност "**bira**". Първите две от тях са референции към един и същ обект в паметта, т.е. са еднакви указатели (адреси в паметта). Третият обект, обаче, се намира на друго място в паметта, макар че има същата стойност като другите два. За да си представите визуално това ще ви помогне фигурата:



Масиви и обекти

Елементите на масив могат да са обекти, но разбира се от един и същ клас. Дефинират се по общоприетия начин.

Масивите са неизменна част от езиците за програмиране. Те представляват съвкупности от променливи, които наричаме **елементи**:



Елементите на масивите са номерирани с числата 0, 1, 2, … N-1. Тези номера на елементи се наричат **индекси**. Броят елементи в даден масив се нарича **дължина на масива**.

Всички елементи на даден масив са от един и същи тип, независимо дали е **примитивен** или**референтен**. Това ни помага да представим група от еднородни елементи като подредена свързана последователност и да ги обработваме като едно цяло.

Масивите могат да бъдат от различни размерности, като най-често използвани са **едномерните** и**двумерните** масиви. Едномерните масиви се наричат още вектори, а двумерните матрици.

Деклариране и заделяне на масиви

В Java масивите имат фиксирана дължина, която се указва при инициализирането му и определя броя на елементите му. След като веднъж сме задали дължината на масив не е възможно да я променяме.

Деклариране на масив

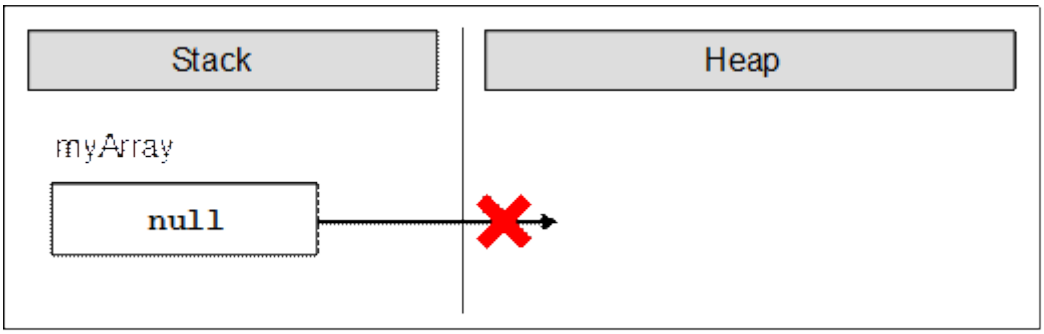
Масиви в Java декларираме по следния начин:

```
int[] myArray;
```

Тук променливата **myArray** е името на масива, който е от тип (**int[]**) т.е. декларирали сме масив от цели числа. С **[]** се обозначава, че променливата, която декларираме ще е масив, а не единичен елемент.

При декларация името на променливата, която е от тип масив, представлява референция (**reference**), която сочи към **null**, тъй като още не е заделена памет за елементите на масива.

Ето как изглежда една променлива от тип масив, която е декларирана, но още не е заделена памет за елементите на масива:



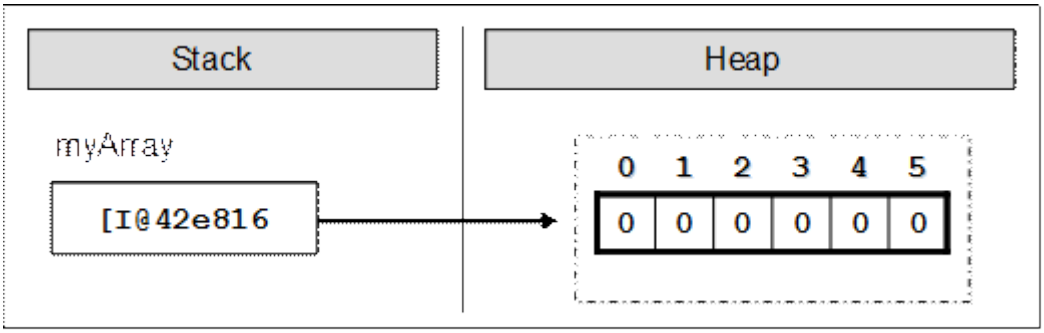
В стека за изпълнение на програмата се заделя променлива с име **myArray** и в нея се поставя стойност **null** (липса на стойност).

Създаване (заделяне) на масив – оператор new

В Java масив се създава с ключовата дума **new**, която служи за заделяне (алокиране) на памет:

```
int[] myArray = new int[6];
```

В примера заделяме масив с размер 6 елемента от целочислен тип. Това означава, че в динамичната памет (heap) се заделя участък от 6 последователни цели числа:



Картинката показва, че след заделянето на масива променливата **`myArray`** сочи някакъв адрес (**`0x42e816`**) в динамичната памет, където се намира нейната стойност. Елементите на масивите винаги се съхраняват в динамичната памет (в т. нар. **heap**).

При заделянето на масив в квадратните скоби задаваме броя на елементите му (цяло неотрицателно число) и така се фиксира неговата дължина. Типът на елементите се пише след**`new`**, за да се укаже за какви точно елементи трябва да се задели памет. Масив с вече зададена дължина не може да се промени т.е. масивите са с фиксирана дължина.

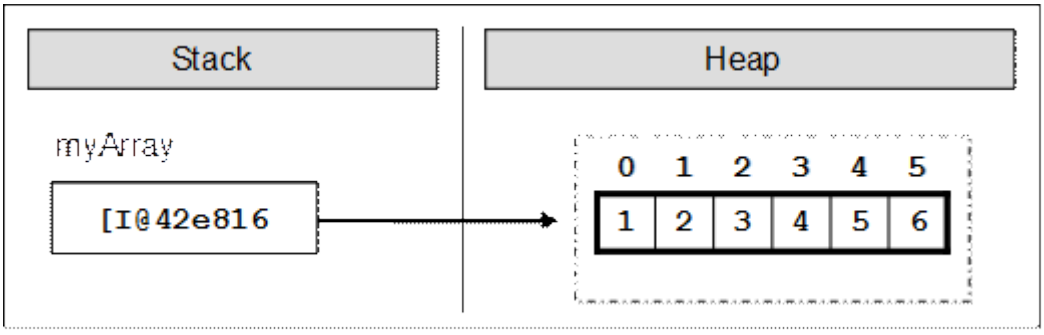
Инициализация на масив. Стойности по подразбиране

Преди да използваме елемент от даден масив той трябва да има начална стойност. В някои езици за програмиране не се задават начални стойности по подразбиране, и тогава при опит за достъпване да даден елемент възниква грешка. В Java всички променливи, включително и елементите на масивите имат начална стойност по подразбиране (default initial value)> Тази стройност е равна на 0 при числените типове или неин еквивалент при нечислени типове (например **`null`** за обекти и **`false`** за булевия тип).

Разбира се, начални стойности можем да задаване и изрично. Това може да стане по различни начини. Един възможен е чрез използване на литерален израз за елементите на масива (**array literal expression**):

```
int[] myArray = {1, 2, 3, 4, 5, 6};
```

В този случай създаваме и инициализираме масива едновременно. Ето как изглежда масива в паметта, след като стойностите му са инициализирани още в момента на деклариране:



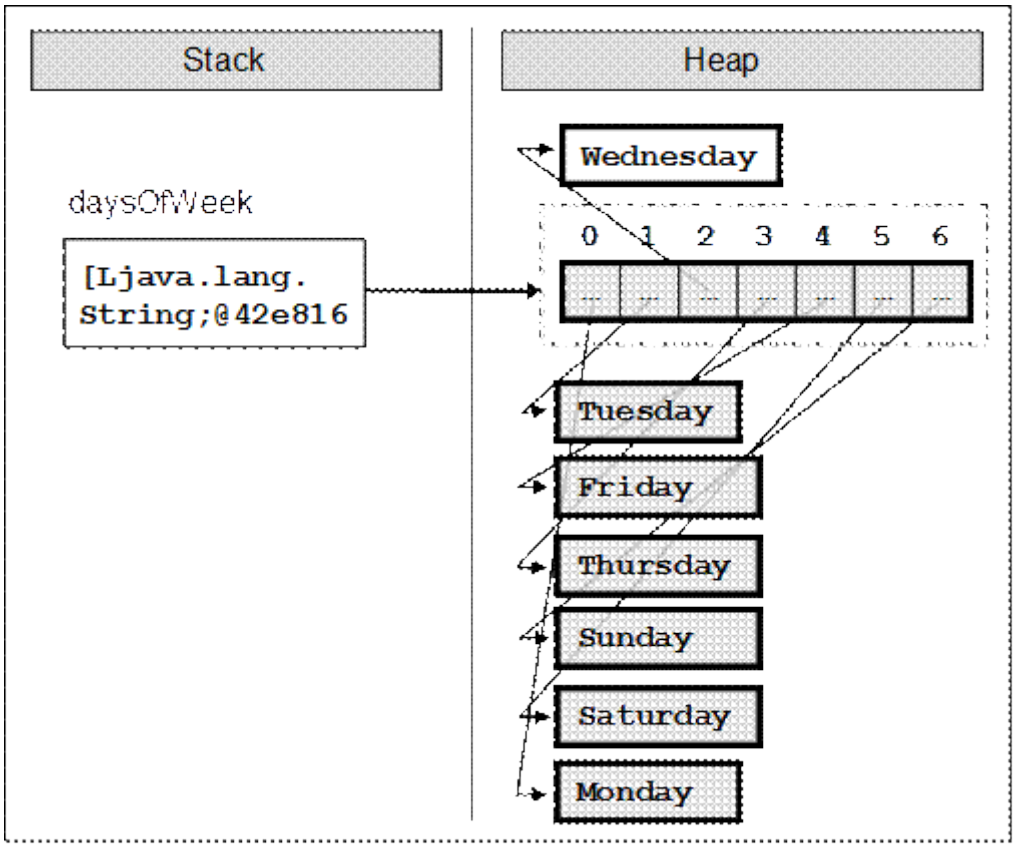
При този синтаксис къдравите скоби заместват оператора **new** и между тях има изброени началните стойности на масива, разделени със запетаи. Техния брой определя дължината му.

Деклариране и инициализиране на масив – пример

Ето още един пример за деклариране и непосредствено инициализиране на масив:

```
String[] daysOfWeek = {

      "Monday", "Tuesday", "Wednesday", "Thursday",

      "Friday", "Saturday", "Sunday" };
```

В случая масивът се заделя със 7 елемента от тип **String**. Типът String е референтен тип (обект) и неговите стойности се пазят в динамичната памет. Ето как изглежда масивът в паметта:



В стека се заделя променливата **daysOfWeek**, която сочи към участък в динамичната памет, който съдържа елементите на масива. Всеки от тези 7 елементи е обект от тип символен низ, който сам по себе си сочи към друга област от динамичната памет, в която се пази стойността му.

Динамични обекти

Създават се в частта heap на паметта чрез **new**, който извиква конструктора.
В C++ се разрушават чрез **delete**, който извиква деструктора. В следващата точка е описано по-подробно как става това.

5. Деструктори. Създаване и разрушаване на обекти на класове.

Деструктори

В Java няма деструктори, създаването и унищожаването на обекти е описано по подробно в следващата точка.

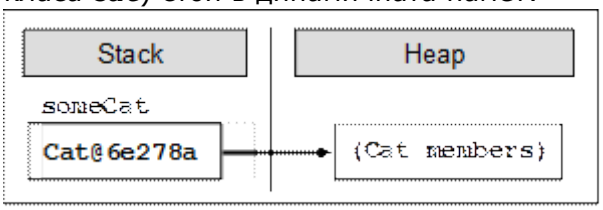
Създаване и разрушаване на обекти на класове

Създаване и освобождаване на обекти

Създаването на обекти от предварително дефинирани класове по време на изпълнението на програмата става чрез оператора **new**. Новосъзданеният обект обикновено се присвоява на променлива от тип, съвпадащ с класа на обекта. Ще отбележим, че при това присвояване същинският обект не се копира. 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:



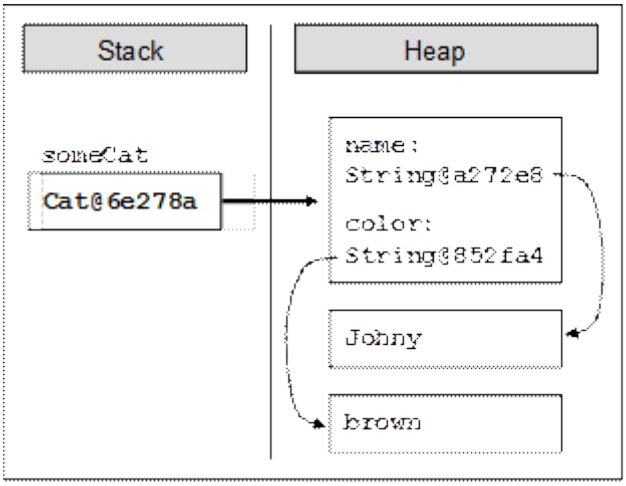
Създаване на обекти със задаване на параметри

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). Следващата картинка показва това нагледно:

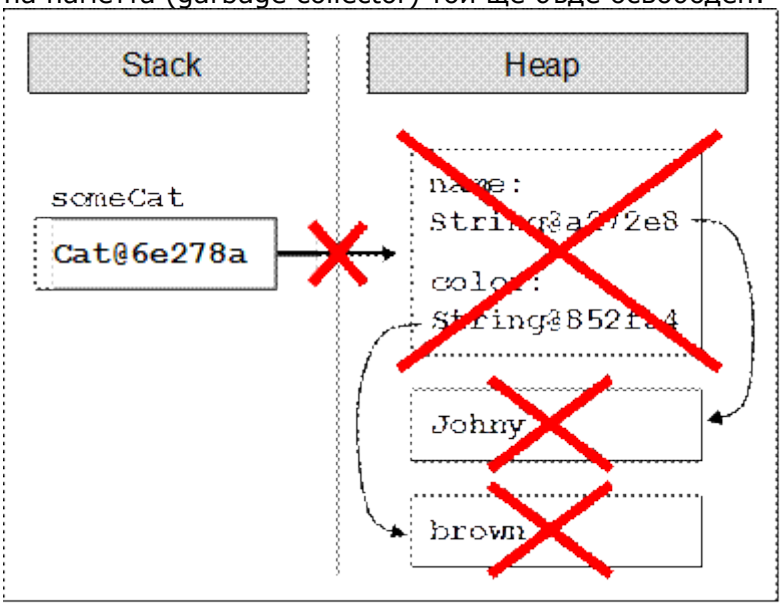


Освобождаване на обектите

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:



6. Операторни. Предефиниране на оператори

Оператори

Езикът C++ има богат набор от оператори. В него са дадени също средства за предефиниране на оператори.

Всеки оператор се характеризира с:

- позиция на оператора спрямо аргументите му;
- приоритет;
- асоциативност.

Позицията на оператора спрямо аргументите му го определя като: префиксен (операторът е пред единствения му аргумент), инфиксен (операторът е между аргументите си) и постфиксен (операторът е след аргумента си).

Предефиниране на оператори

Предефинирането може да стане по два начина:

- чрез функция–приятел;
- чрез член-функция.

Предефинирането на оператори, което се смята за характерна особеност на C++, не се поддържа от Java. Подобна функционалност може да се имплементира като методи в класовете на Java, синтактичното удобство на предефинирането на операторите липсва. В защита на Java трябва да се отбележи, че предефинирането понякога може да направи неясни програмите, които ги ползват. Разработчиците на Java са решили да не се поддържа, за да се запази езика колкото може по-прост.

7. Производни класове. Дефиниране. Наследяване и достъп до наследените компоненти.

Производни класове.

Производните класове и наследяването са една от най-важните характеристики на обектно-ориентираното програмиране (ООП). Чрез механизма на наследяване от съществуващ клас се създава нов клас. Класът от който се създава се нарича **базов (основен) клас**, а този, който е създаден - **производен**. Понятията основен и производен клас са относителни, тъй като производен клас може да е основен за други класове, а основен – да е производен от други основни класове. Производният клас може да наследи компонентите на един или няколко базови класа. В първия случай наследяването се нарича **единично (просто)**, а във втория – **множествено**.

Дефиниране

Дефинирането на производни класове е еквивалентно на конструирането на йерархии от класове. Ако множество от класове имат общи данни и методи, тези общи части могат да се обособят като основни класове, а всяка от останалите части да се дефинира като производен клас на съответния основен клас. Така се прави икономия на памет, тъй като се избягва многократното описание на едни и същи програмни фрагменти.

При конструирането на производни класове е достатъчно да се разполага само с обектните модули на основните класове, а не с техния програмен код. Това позволява да бъдат създавани библиотеки от класове, които да бъдат използвани при създаването на производни класове.

Наследяване и достъп до наследените компоненти

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;


public class Felidae { // Latin word for "cat"


	private boolean male;


	public Felidae() {
		this(true);
	}


	public Felidae(boolean male) {
		this.male = male;
	}
```

```
	public boolean isMale() {
		return male;
	}

	public void setMale(boolean male) {
		this.male = male;
	}
}
```

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

Lion.java

```
package introjavabook;

public class Lion extends Felidae {
	private int weight;

	public Lion(boolean male, int weight) {
		super(male); // Shall be explained in the next paragraph
		this.weight = weight;
	}

	public int getWeight() {
		return weight;
	}

	public void setWeight(int weight) {
		this.weight = weight;
	}
}
```

Ключовата дума super

В горния пример в конструктора на **Lion** използваме ключовата дума **super**. Тя указва да бъде използван базовият клас и позволява достъп до негови методи, конструктори и член-променливи. Със **super()** можем да извикваме конструктор на базовия клас. Със **super.method()** можем да извикваме метод на базовия клас, да му подаваме параметри и да използваме резултата от него. Със **super.field** можем да вземем стойността на член-променлива на базовия клас или да ѝ присвоим друга стойност.

В Java наследените от базовия клас методи могат да се **пренаписват** (**override**). Това означава да им се подмени имплементацията, като оригиналният сорс код от базовия клас се игнорира, а на негово място се написва друг код.

Можем да извикваме непренаписан метод от базовия клас и без **super**. Употребата на ключовата дума е необходима само ако имаме пренаписан метод или променлива със същото име в наследения клас.

super може да се използва изрично, за яснота. super.method() извиква метод, който задължително е от базовия клас. Такъв код се чете по-лесно, защото знаем къде да търсим въпросния метод.
Имайте предвид, че ситуацията с this не е такава. this може да означава както метод от конкретния клас, така и метод от който и да е базов клас.

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) {
                // If we comment the next line, AfricanLion
                // will not compile. Try it.
                super(male, weight);
        }

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

        // …
}
```

Ако закоментираме или 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.

Нива на достъп при наследяване

В главата "Дефиниране на класове" разгледахме нивата на достъп за свойствата и методите:**public**, **private** и **default** (friendly). Освен тях в Java има и още едно ниво на достъп –**protected**. То е свързано с наследяването.

Когато се наследява един базов клас:

- Всички негови **public** и **protected** методи и свойства са видими за класа наследник.
- Всички негови **private** методи и свойства не са видими за класа наследник.
- Всички негови **default** (friendly) методи и свойства са видими за класа наследник само ако базовият клас и наследникът са в един и същ пакет (package).

Ето един пример, с който ще демонстрираме нивата на видимост при наследяване:

Felidae.java

```
package introjavabook;


public class Felidae { // Latin for cat


	private boolean male;


	public Felidae() {
		// Call another constructor with default values
		this(true);
	}


	public Felidae(boolean male) {
		this.male = male;
	}




	//…
}
```

Ето как изглежда и класът **Lion**:

Lion.java

```
package introjavabook;


public class Lion extends Felidae {

	private int weight;
```

```
	public Lion(boolean male, int weight) {
		super(male); // visible - Felidae's public constructor.
		super.male = male; // invsible – male is private.
		this.weight = weight;
	}

	//…
}
```

Ако се опитаме да компилираме този пример, ще получим грешка, тъй като private променливата**male** от класа **Felidae** не е достъпна от класа **Lion**.

Класът Object

Появата на обектно-ориентираното програмиране de facto става популярно с езика 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 вглавите 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 заколекции, текстови файлове и символни низове). Тези класове са част от Java платформата – навсякъде, където има Java, ги има и тях. Тези класове се наричат **стандартни клас-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;


public class ObjectExample {

	public static void main(String… args) {

		AfricanLion africanLion = new AfricanLion();

		// Implicit casting

		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;


public class ToStringExample {

	public static void main(String… args) {

		AfricanLion africanLion = new AfricanLion();

		System.out.println(africanLion.toString());


		// Result: "introjavabook.AfricanLion@de6ced"
```

```
	}

}
```

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

```
Public class Object {

	// …

	public String toString() {

		return getClass().getName() +

			"@" + Integer.toHexString(hashCode());

	}

	// …

}
```

Пренаписване на toString() – пример

Нека сега ви покажем колко полезно може да е пренаписването на метода **toString()**, наследено от **java.lang.Object**:

AfricanLion.java

```
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());
	}


	// …
}
```

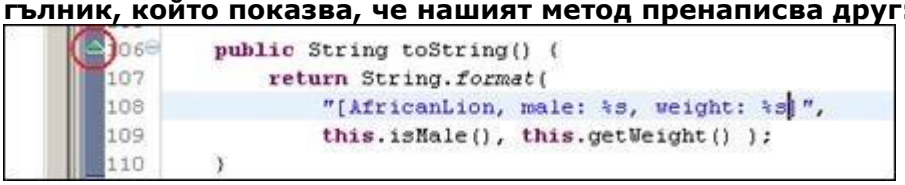
В горния код използваме **String.format(String format, Object… args)** метода, за да форматираме резултата по подходящ начин. Ето как можем след това да извикваме пренаписания метод **toString()**:

ToStringExample.java

```
package introjavabook;


public class ToStringExample {


	public static void main(String… args) {


		AfricanLion africanLion = new AfricanLion(true, 15);
		System.out.println(africanLion);


		// Result: "[AfricanLion, male: true, weight: 15]"
	}
}
```

Забележете, че извикването на **toString()** става скрито. Когато на метода **println()** подадем някакъв обект, този обект първо се преобразува до стринг чрез **toString()** метода му и след това се отпечатва в изходния поток. Така при печатане на конзолата няма нужда изрично да преобразуваме обектите до стринг.

Ако ползваме средата Eclipse и искаме да сме сигурни, че пренаписваме метод, можем да следим за зелен триъгълник, който показва, че нашият метод пренаписва друг:



От Java 5 нататък има начин да укажем изрично на компилатора, че искаме нашият метод да пренаписва друг. За целта се използват се т. нар. **анотации**, а в конкретния случай се използва анотацията **`@Override`**:

AfricanLion.java

```
public class AfricanLion extends Lion {

	// …


	@Override
	public String toString() {
		return String.format(
			"(AfricanLion, male: %s, weight: %s)",
			this.isMale(), this.getWeight());
	}


	// …
}
```

Изричното указване на компилатора, че искаме да пренапишем метод от базов клас, е препоръчителна практика и намалява грешките. Ако си създадем навика при пренаписване на метод винаги да ползваме анотацията **`@Override`**, ако случайно сбъркаме една буква от името на метода или типовете на неговите параметри, компилаторът веднага ще ни съобщи за грешката.

Транзитивност при наследяването

В математиката транзитивност означава прехвърляне на взаимоотношения. Нека вземем операцията "по-голямо". Ако A>B и B>C, то можем да заключим, че A>C. Това означава, че релацията "по-голямо" (>) е транзитивна, защото може еднозначно да бъде определено дали A е по-голямо от C или обратното.

Ако клас **`Lion`** наследява клас **`Felidae`**, а клас **`AfricanLion`** наследява клас **`Lion`**, това индиректно означава, че **`AfricanLion`** наследява **`Felidae`**.

Следователно **AfricanLion** също има свойство **male**, което е дефинирано във **Felidae**. Това полезно свойство позволява определена функционалност да бъде описана в най-подходящия за нея клас.

Транзитивност – пример

Ето един пример, който демонстрира транзитивността при наследяване:

TransitiveInheritance.java

```
package introjavabook;


public class TransitiveInheritance {


      public static void main(String… args) {


            AfricanLion africanLion = new AfricanLion(true, 15);

            // Method defined in Felidae

            africanLion.isMale();

            // Method defined in Felidae

            africanLion.setMale(true);

      }

}
```

Заради транзитивността на наследяването можем да сме сигурни, че всички класове имат**toString()** и другите методи на **Object** без значение кой клас наследяват.

Наследяване и достъп до наследените компоненти (2)

Веднъж създаден един клас като наследник на друг той притежава всички полета и методи на своя базов клас. За разлика обаче от базовия клас производния няма достъп до дефинираните като private компоненти на базовия клас.

Когато изучавахме спецификаторите за достъп казахме, че те са 4 на брой: public, private, protected, (default). Тогава не разгледахме спецификатора за достъп **protected** понеже казахме, че той има отношение към наследяването, а именно: До компонентите на деден клас дефинирани като **protected** имат достъп само методите

на същия клас и методите на неговоте наследници. Методите на всички останали класове нямат достъп до тях.

Следната таблица описва различните права на достъп до компонентите на базов клас.

	public	private	protected	(default)
Base class	Yes	Yes	Yes	Yes
Derived class	Yes	No	Yes	-
Class in the same package	Yes	No	-	Yes
Class in different package	Yes	No	-	No

Access Levels

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N