Лекция 9

Обектни характеристики на Java

Езикът Java е обектно-ориентиран дори на най-ниско ниво. Всички Java програми се изпълняват в рамките на даден клас, поради което дори най-простата програма (“Hello world”) изисква дефинирането на клас.

Като обектно-ориентиран език Java взема най-добрите концепции и възможности на съществуващите обектно-ориентирани езици. Java надминава C++, като разширява обектния модел и премахва сложността му. С изключение на основните типове данни , всичко в Java е обекти и дори те могат да бъдат капсулирани в обекти, когато възникне такава нужда.

В обектно-ориентираното програмиране са въведени три основни концепции: капсулиране, полиморфизъм и наследяване.

Капсулирането е програмен механизъм, който свързва заедно кода и данните , с които работи, и същевременно ги предпазва от външна намеса и неправомерен достъп. В обектно-ориентираните езици кодът и данните могат да се разположат като в черна кутия, в която се намира всичко необходимо за работа на програмата. Когато кодът и данните са свързани по този начин се създава обект.

В рамките на един обект кодът, данните или и двете могат да бъдат частни (private) за този обект или публични (public). Частният код или данни са познати или достъпни само за този обект (други членове на обекта). Когато кодът или данните са публични, другите части от програмата (извън този обект) имат достъп до тях и могат да ги използват.

Основна единица за капсулиране в Java е класът. Класът дефинира формата на обектите, които могат да се реализират с него. Обектите представляват копия (инстанции) на класа. Класът може да се разглежда като план (проект), по който се изгражда даден обект. Затова в програмите се работи с обекти, които са реални конструкции, а класовете се разглеждат като описание на тези конструкции.

Кодът и данните, които съставят един клас се наричат членове на класа. Данните се наричат членове-променливи, а кодът който оперира с тези данни се нарича членове-методи.

Полиморфизмът (от гръцки – ‘с много форми’) е свойство, което позволява на един и същ интерфейс да извършва достъп до общ клас от действия. Във всеки конкретен случай се реализира някакво специфично действия, което се определя от конкретната ситуация. Ако тези действия имат общ принцип те могат да се активират от една инструкция, а не да се създават множество инструкции, които да активират отделните действия.

По-общо казано, принципът на полиморфизъм може да се изкаже с фразата ‘един интерфейс много методи’. Това означава, че е възможно да се проектира един общ интерфейс за група от свързани процеси. Полиморфизмът помага да се редуцира сложността на алгоритмите. Като пример може да се разгледа обектно-ориентиран графически пакет, състоящ се от различни графични обекти. Те описват геометрични фигури – окръжност, правоъгълник, триъгълник и т.н. Тези обекти се използват за създаване на изображения (например в CorelDraw, ACAD). За всеки обект са зададени параметри – размери, цвят, разположение, а също така и действия - начинът по който реагира на различни инструкции (за преместване в ново положение или за изобразяване на екрана). За да се покаже едно изображение на екрана трябва на всеки обект да се изпрати инструкция да се нарисува (изобрази). Методите които рисуват отделните изображения са различни, тъй като формата на обектите е различна, но те се активират от една и съща инструкция. Този механизъм на обща интерпретация на инструкции се нарича полиморфизъм.

Наследяването е процес, при който един обект може да придобие свойства на друг обект. В процеса на наследяване се поддържа принципът на йерархическата класификация. Това означава, че може да се декларира клас с определени общи свойства, и на негова основа да се декларират дъщерни класове запазващи общите свойства и добавени нови свойства, специфициращи новия обект. Например, Java е част от класификацията на програмни езици,

които от своя страна са част от квалификацията на програмно осигуряване. Java притежава общи характеристики на програмно осигуряване и на програмни езици, но същевременно има специфични характеристики, които се отнасят само за него. Принципът на наследяване позволява, когато има йерархическа класификация на обекти да не се задават за всеки обект всички свойства, а само тези, които са специфични за него, а другите се наследяват родителските класове.
Основен елемент при обектно-ориентираното програмиране е понятието клас. Клас представлява шаблон, който дефинира формата и свойствата на даден обект. Той определя едновременно както данните, така и кодът, който оперира с тези данни. Java използва класова спецификация за изграждане на обект. Класът е логическа абстракция. Докато не бъде създаден обект от даден клас не съществува физическо представяне на този клас в паметта на компютърната система.

1 Класове и обекти

Класът е софтуерна конструкция, която дефинира полетата и методите на обектите. Той определя как ще изглежда обекта и какво поведение ще има, когато бъде създаден въз основа на декларираната спецификация (клас). Следователно, конкретни обекти могат да се създават само въз основа на предварително дефинирани класове. На основата на един клас могат да се създадат множество обекти, по същият начин както може да се построят много къщи по един архитектурен проект. Ето как изглежда декларацията на един много прост клас, наименован Point:

```
class Point {
     public double x;    /*  поле */
     public double y;   /* поле */
}
```

Класът се създава посредством ключовата дума **class**. В разглеждания пример са описани само две полета ***x*** и ***y***, които са реални числови данни, които могат да бъдат интерпретирани като координатите на една точка в равнината. Това е един шаблон, въз основа на който могат да се дефинират реални обекти, които да са точки в равнината. Към тази декларация могат да се добавят различни функции (методи), които биха могли да манипулират данните в обекта.
Въпреки че не съществува задължително синтактично правило, един добре дефиниран клас трябва да обединява логически свързана информация. Например, един клас, който съхранява имена, телефонни номера и адреси, обикновено не бива да съдържа информация за курсов на валутите или резултати от спортни срещи. Смисълът е в това, че един добре структуриран клас съдържа логически свързана информация. Поставянето на несвързана информация в един и същ клас деструктурира кода който обработва информацията и дефиницията на класа губи смисълът си.
Всяка дефиниция на клас задава нов тип данни (наред с вградените типове данни - целочислени, реални, символни и т.н.). В горния пример, новият тип данни се нарича **`Point`**. Това име може да се използва за деклариране на обекти от тип **`Point`**. За създаване на обект като инстанция на даден клас се използва следната синтактична конструкция:

```
Point center = new Point (); //създава един Point обект с име center
```

След изпълнение на този оператор center ще представлява инстанция на клас **`Point`** и ще съществува ‘физически’. В паметта ще бъде отделено място за всяка инстанционна променлива дефинирана в спецификацията на класа.. За достъп до тези променливи се използва символа точка ‘.’, като в обръщението се задава името на обекта и името на променливата, разделени с точка. Например, ако трябва да се зададе стойност на променливите може да се използва оператор за присвояване:

```
center.x := 2.85 ; center.y := 5.24 ; // достъп до променливи от обект
```

Горната дефиниция на обект може да се запише и в две стъпки. В първата стъпка се декларира променлива, която задава само наименованието на обекта. В следваща стъпка се създава физическо копие на обекта в паметта:

```
Point center ;
center = new Point () ;
```

Една завършена програма която използва клас **Point** изглежда по следния начин:

```
// File PointUse.java
class Point {
        public double x;   /* поле */
        public double y;   /* поле  */
        }
class PointUse {
public static void main (String args[]) {
Point first = new Point ();
Point second = new Point () ;
double distance ;
// стойности на полетата на обекти first и second
        first.x = 3.5 ; first.y = 4.7 ;
        second.x =8.3 ; second.y = 9.9 ;
        distance = Math.sqrt((second.x-first.x)*(second.y-first.y) +
           (second.y-first.y)*(second.y-first.y)) ;
        System.out.println ("distance : "+distance) ;
        }
}
```

В примера са дефинирани два обекта от тип Point – first и second. Всеки от тези обекти притежава собствени копия на инстанционните променливи, дефинирани в класа от който произлизат обектите. Съдържанието на променливите от двата обекта са независими едни от други. Между двата обекта няма никаква връзка, освен че произлизат от един и същ клас.

В примера се използва вграден метод за определяне на квадратен корен sqrt (). Този метод е член на класа Math, който съдържа дефиниции на основните математически функции. Класът Math е част от един голям пакет класове java.lang, който автоматично се импортира в процеса на компилация.

Обектите са софтуерни модели, които наподобяват много обектите в реалният живот. Обектите в реалния живот имат структура (състояние) и поведение. Един автомобил има състояние (какъв тип елементи съдържа, каква скорост развива, какъв е разхода на гориво и други), а поведението се определя от такива действия като тръгване, спиране, смяна на скоростта, обръщане, понасяне и т.н. В програмната реализация на обектите, състоянието се описва чрез неговите данни (полета). Данните биват собственост на обекта (private) или общодостъпни (public). Поведението на обекта се описва чрез неговите методи. Методите манипулират полетата, като създават ново състояние на обекта. Методите на обекта могат да създават нови обекти.

Един модел на софтуерен обект може да бъде представен като клетка в ядрото на която се намират данните (полетата), а за обвивка (външна мембрана) се използват методите, които осигуряват връзката на обекта с околният свят. Полетата (данните) на един обект са пакетирани в обекта. Те са оградени от методите и те са единствените средства за използване на данните.

Една важна характеристика, която отличава обектите от обикновените процедури е, че един обект може да има живот по-дълъг от живота на създалият го обект. Този аспект на обектите в по-голяма част от случаите не се забелязва. В разпределените клиент-сървър среди, това дава възможност обектите да бъдат създадени на едно място, предавани по мрежата и запазвани някъде, например в база от данни, за да бъдат извикани обратно по късно.

2 Методи на класове

Методите са съставна част от класовете. Методите са подпрограми, които работят с данните дефинирани в класа, а в много случаи служат за достъп до данните. Обикновено, другите части на програмата взаимодействат с даден клас посредством неговите методи. Затова, съвкупността от методите на един клас се нарича интерфейс.
В добре написан Java код един метод обикновено изпълнява една определена задача. Всеки метод има наименование, което се използва за извикване на метода. По принцип, на метода може да се зададе произволно наименование. Трябва да се знае, че **main**() е наименование, запазено за метода, с който започва изпълнението на програмата. Ключовите думи в Java също не трябва да се използват като наименования на методи. След наименованието на всеки метод има отваряща и затваряща скоба, между които се разполагат аргументи, ако има такива. По този начин се различават методите от обикновените променливи. В следващия пример е дефиниран обект Circle (окръжност) с полета - координатите на центъра (centerx и centery) и радиуса (radius). Освен тях е дефиниран и метод Area за изчисляване на площта на окръжността.

```
// Файл CircleUse
class Circle {
        public double centerx;
        public double centery;
        public double radius;
        public double Area () {
                return Math.PI*this.radius*this.radius ;
        }
}
class CircleUse {
public static void main (String args[]) {
Circle first = new Circle();
double Circle_area ;
// стойности на елементите на обект first
        first.centerx = 3.5 ; first.centery = 4.7 ;
        first.radius =4.3 ;
        Circle_area = first.Area() ;
        System.out.println ("area : "+ Circle_area) ;
        }
}
```

От представения пример се вижда, че методите се описват посредством името на метода, спецификация за достъп (**public** или **private**) и тип на връщаната стойност (в случая **double**). Изпълнимите операции и декларации за метода (тяло на метода) се записват между двойка фигурни скоби ‘{}’. Връщаната стойност от метода се записва след служебната дума **return**. За определяне на площта на кръга се използва дефинираната в класа Math променлива PI.
За да се изясни напълно начинът на създаване на методи на класове трябва да се изясни ролята на ключовата дума **this**. Когато се извиква един метод, на него автоматично му се предава един подразбиращ се аргумент, представляващ референция (обръщение) към извикващия обект. Тази референция се нарича **this** (този). С референцията **this** се извършва обръщение към променлива или метод в обекта в който се намирате. Конструкцията **this.**radius означава, че променливата radius е инстанционната променлива на обекта за който се използва метода. В рамките на класа всички членове са достъпни. Следователно, изразът който връща стойност в метода Area(), може да се напише и във вида: **return** Math.PI*radius*rad ;. Това е съкратен запис на използвания в примера оператор. Използването на **this** позволява да се декларират локални променливи със същите имена както имената на някои инстанционни променливи. Тогава за да се различават тези типове променливи, инстанционните променливи се използват с референцията **this**.
В класовете могат да се използват и параметризирани методи. Това се постига чрез използване на аргументи (параметри), които се записват в скобите след наименованието на

метода. Синтаксисът на деклариране на параметрите е същият както при деклариране на обикновена променлива. Параметрите са валидни в областта на действие на метода и се явяват локални променливи за него. В горния клас Circle може да се добави още един метод, който да определя обема на цилиндър образуван от зададената окръжност и височина зададена като параметър за метода:

```
public double Volume (double height) {
        return Math.PI*this.radius*this.radius*height ;
```

Този метод след това може да бъде използван в класа CircleUse за определяне на обем на цилиндър:

```
System.out.println ("Volume : "+ first.Volume(5)) ;
```

3. Конструктори и деструктори на класове.

Конструкторите на класове са методи, които инициализират обектите при тяхното създаване. Конструкторът има същото име, както името на класа. В синтактично отношение конструкторът е като метод, само че не връща стойност. Затова не се задава тип на връщания резултат. Обикновено конструктор се използва за да се зададат начални стойности на дефинираните в класа инстанционни променливи или да се извършат с тях някои начални инициализиращи действия.

Всички класове имат конструктори, независимо дали те са декларирани явно или не, тъй като Java автоматично осигурява подразбиращ се конструктор, който инициализира всички членове на обекта с нули. Когато обаче има деклариран от потребителя конструктор, подразбиращият се конструктор се игнорира. Ето пример на дефиниран конструктор за класа Circle:

```
// File PointUse_K.java
// import java.lang.* ;
class Circle {
        public double centerx;
        public double centery;
        public double radius;
        // Конструктор
        Circle (double x, double y, double r) {
                centerx=x ; centery=y; radius=r; }
        public double Area () {
                return Math.PI*radius*radius ;}
        public double Volume (double height) {
                return Math.PI*radius*radius*height ;
        }
}
class CircleUse_K {
public static void main (String args[]) {
Circle first = new Circle(3.5,4.7,5.5);// създаване и инициализация на обект
double Circle_area ;
        Circle_area = first.Area() ;
        System.out.println ("area : "+ Circle_area) ;
        System.out.println ("Volume : "+ first.Volume(5)) ;
        }
}
```

Конструкторът в този пример е параметризиран, тъй като началните стойности на инстанционните променливи се задават като параметри при създаване на обекта. Тъй като конструктора е метод, в него могат да се предвидят и други действия, освен задаване на начални стойности.

Освен конструктори в обектно-ориентираното програмиране се използват и **деструктори**. Ако ролята на конструкторите е да конструира даден обект (да го инициализира), то ролята на деструктора е да извърши определени действия, когато даден обект се освободи (затвори).

Тъй като за обектите се заделя динамично памет (по време на изпълнение на програмата), то е необходимо след освобождаване на обекта да се освободи тази памет. В Java съществува модул за събиране на боклук (garbage collector), който има грижата да освободи заетата памет. Затова, декларирането на деструктори за класовете не е задължително, но когато трябва да се извършат някои допълнителни действия при освобождаване на обект, може да се създаде специален метод – деструктор.

4. Предефиниране на методи.

В Java е възможно два или повече методи от един клас да имат едно и също име, стига декларациите на техните параметри да са различни. Когато има дефинирани повече от един метод с еднакви имена, се казва че методите са предефинирани (***overloaded***). Предефинирането на методи е един от начините за реализиране на полиморфизъм.

За да се предефинира един метод е необходимо да се реализира нова версия на метода. Компилаторът поема грижата да изпълни правилния метод по време на изпълнение на програмата в зависимост от условията за използване на метода. За да може компилаторът да разпознае кой от методите с еднакви имена да използва се налагат някои ограничения при синтактичната реализация на предефинираните методи. Едно от най-важните изисквания е типът и/или броят на параметрите на всеки предефиниран метод да е различен. Не е достатъчно два метода да се различават по типа на връщания резултат. Компилаторът може да направи избор за правилния метод само по типа и броя на параметрите на метода. При извикване на даден метод се изпълнява тази версия от предефинираните методи, която съответства по тип и брой на параметрите на извикващата инструкция.

Предефинираните методи могат да бъдат илюстрирани със следния пример:

```
// File OverloadUse.java
class Overload {
        void ShowParam () {
        System.out.println(" No parameters !!") ;  }
        void ShowParam (int i) {
        System.out.println(" One parameter" + i );  }
        void ShowParam (double x) {
        System.out.println(" One parameter" + x ) ;  }
        void ShowParam (int i, int j) {
        System.out.println(" Two parameters" + i +"  "+j) ;  }
        }
class OverloadUse {
        public static void main(String args[]) {
        Overload  obj = new Overload() ;
        obj.ShowParam () ;
        obj.ShowParam(5) ;
        obj.ShowParam(3.12) ;
        obj.ShowParam(5,6) ;
        }
```

}В този пример са декларирани 4 предефинирани метода, които извеждат различни съобщения в зависимост от параметрите при извикване на метода. Компилатора избира подходящия метод след анализ на подаваните в извикващата инструкция параметри. Това е един вид полиморфизъм, защото с един общ извикващ механизъм се реализират различни действия.

5 Наследявани при класовете.

Java подържа наследяване, като дава възможност един клас да внедри друг клас в своята декларация. Това се извършва с помощта на ключовата дума **extends**. Класът, който използва **extends** се нарича подклас, а класът който се внедрява в декларацията се нарича надклас. При наследяването подкласът надгражда (разширява) надкласа. Голямото предимство на наследяването е, че веднъж създаден клас може да бъде използван за

създаване на произволен брой класове със сходни свойства. Следващият пример илюстрира наследяването при класовете.

```
// File Figures.java
class TwoDimFig {
        public double width;
        public double height;
        public void ShowFig () {
        System.out.println (" Width is :" + width +
              " height is :"+height) ;}
}
class Rectangl extends TwoDimFig {
        Rectangl (double w, double h) {
                width = w; height = h; }
        public double area () {
        return width * height ;}
        }
class Triangle extends TwoDimFig {
        Triangle (double w, double h) {
                width = w; height = h; }
        public double area () {
        return width*height/2 ;}
}
class Figures {
        static public void main (String args[]) {
        Rectangl f1 = new Rectangl (4.0,8.0) ;
        Triangle f2= new Triangle (4.0,8.0) ;
        System.out.println(" Rectangle area :"+f1.area()) ;
        System.out.println(" Triangle area :"+f2.area()) ;
        }
}
```

В примера TwoDimFig е надклас за Rectang и Triangle. Той описва равнинни фигури с две димeнсии (width и height), а подкласовете Rectang и Triangle специфицират (разширяват) декларациите съответно за правоъгълник и триъгълник. Класовете Rectang и Triangle използват дефинициите за размери на фигурата от надкласа (width и height), а в тях са добави дефиниции за изчисляване на площта на фигурата (те са различни за двете фигури).

В подкласовете са добавени конструктори за инициализация на обектите. При наследяването може да се използва и конструктори, дефинирани в надкласа, но това става при спазване на определени правила. Засега е напълно достатъчно да се познава използването на конструктори за всеки подклас.

При наследяването може да се използва не само проста схема подклас – надклас, но и по сложни схеми. Възможно е създаване на йерархични структури съдържащи неограничен брой наследявания. Те се използват за по-сложни програмни приложения.