

сряда, 22.02.2017

Defining Classes

Abstract data types(Абстрактни типове данни) - преди да създаден даден клас трябва преди това да си зададем въпроси като какво представлява един клас ? Как да си създадем такъв ? Откъде да тръгнем ? Как да го дефинираме ? и др. И това става с т.наречените абстрактни типове данни. Или с други думи казано това е някакво обобщение на нещо от реалния свят. Описание на нещо, което ние ще можем да го напишем като код след това и да го използваме като някакъв клас.

Примерно да се абстрагираме от това, че в Java има някакъв тип данни, който се нарича String() и просто си говорим за нещо, което се нарича String. И какво представлява то ? Един низ от символи, които се пазят в един масив. И когато ние дефинираме такъв абстрактен клас трябва да помислим какво да има той. Примерно:

- Един стринг трябва да ни дава възможност да го създаваме;
- Да можем примерно да видим какво му е дължината - int length;
- Да можем да видим какъв знак има на дадена позиция в string-а - char charAt(int index);
- Да можем примерно да видим дали стрингът е пълен или празен - boolean isEmpty();
- И др. такива неща, които можем да използваме.

▪ Data type whose representation is hidden from the client

String ADT – indexed sequence of chars:

```
String()
int length()
char charAt(int index)
boolean isEmpty()
```

// many others...

В момента обсъждаме какво трябва да има този String преди въобще да го направим на клас. Тоест **един абстрактен тип данни ни показва всичко, което един потребител трябва да има, за да може да използва нещо в компютърния свят.**

```
String()
int length()
char charAt(int index)
boolean isEmpty()
```

На горната картичка е показано описание на String-а , което задължително трябва присъства ако направим един такъв клас, за да може потребителят да го използва като клас.

Абстрактният тип данни ни позволява да скрием имплементацията от потребителите. Тоест ние като използваме един String не ни интересува как

работи отдолу. Не ни интересува, че символите се пазят в масив, или че този масив си има индекси и оттам си вземаме позицията на всяка една буквичка, не ни интересува как той разбира дали стрингът е пълен или празен и др.. Просто знаем, че има такова поведение и как да го използваме.

Какво ни дава тази концепция за абстрактен тип данни ? Дава ни независимост от езика, на който пишем. Тоест можем да говорим за неща от компютърния свят като стекове и опашки, масиви и др. без въобще да ни интересува на какъв език пишем. Просто знаем, че имат някакво поведение и ние можем да го използваме без да знаем на какъв език е направено, как работи то ? и тн.

Още един пример:

Java OOP Basics - Defining Classes - февруари 2017 - Петър Пенев

Abstract Data Type (2)

▪ You don't need to know the implementation to use an ADT

The diagram shows two abstract data types. On the left, a circle contains a silhouette of a dog, with the text 'Dog:' below it. To its right is a box containing the class definition 'Dog()' and methods 'String getName()', 'void bark()', and 'void sleep()'. On the right, a circle contains a computer monitor, with the text 'Computer:' below it. To its right is a box containing the class definition 'Computer()' and methods 'void turnOn()', 'void turnOff()', and 'String getSpecs()'.

Може да имаме абстрактен клас куче както е показано на горната картичка. И сега за нашата програма ни е необходимо по някакъв начин да можем да създаваме кучето, трябва ни начин да вземем неговото име -> `String getName();`; Трябва ни някакъв начин да го накараме да лай -> `void bark();`; и също така ни трябва някакъв начин да го накараме да спи -> `void sleep();`; И сега като разполагаме с това вече можем да го имплементираме, на който си език искаме. Същото е и за примера с компютъра на горната картичка. За нашата програма ни трябва единствено как да го създаваме ?, Как да го пускаме ?, Как да го изключваме ? И как да му вземем характеристиките ? Отново както и с миналия пример , не ни интересува имплементацията, нито на кой език ще го пишем. Просто ни трябва да го дефинираме по някакъв подобен начин. Ако е за някоя друга програма може да ни трябват съвсем различни неща.

Defining Classes (Създаване на класове) **Какво представлява класът ? - Класът представлява просто имплементация на един абстрактен тип данни.** Просто това, което сме обсъждали като идеи, го напишем на дадем език за програмиране(който е обектно-ориентиран и поддържа класове) тогава имаме конкретна имплементация на абстрактен тип данни.

Пример: Искаме да си създаме клас Зар. Примерно за нашата програма сме уточнили , че зарчето трябва да брой страни, ще има тип, ще има метод, който ще го хвърля. Класът се дефинира както е показано на следващата снимка.

Java OOP Basics - Defining Classes - февруари 2017 - Петър Пенев

Defining Simple Classes

▪ Class is a concrete implementation of an ADT

▪ Classes provide structure for describing and creating objects

```
class Dice {  
    ...  
}
```

Класовете ни дават възможността да опишем с тях някаква структура, по която после да си създаваме обекти от въпросния клас. Класът ни дава описанието на обекта, а самият обект е просто екземплярен от класа.

ВАЖНО НЕЩО Е ВСЕКИ ПЪТ КОГАТО СИ ПРАВИМ КЛАС, ТОЙ ДА БЪДЕ В ОТДЕЛЕН ФАЙЛ. ТОВА Е ДОБРАТА ПРАКТИКА.

Java OOP Basics - Defining Classes - февруари 2017 - Петър Пенев

Defining Simple Classes

The diagram shows a code snippet for defining a class named 'Dice'. It highlights the 'Keyword' (class), 'Class name' (Dice), 'Class body' (the code block), and an ellipsis (...). To the right, a yellow box contains a screenshot of a 'Create New Class' dialog box. The 'Name:' field is set to 'Dice' and the 'Kind:' dropdown is set to 'Class'. Buttons for 'OK' and 'Cancel' are at the bottom.

- Class is a concrete implementation of an ADT
- Classes provide structure for describing and creating objects

ДРУГО ВАЖНО НЕЩО Е ИМЕНУВАНЕТО НА КЛАСОВЕТЕ. КОНВЕНЦИЯТА ГЛАСИ ДА СЕ ИЗПОЛЗВА PascalCase. Това означава първата буква от класа винаги да бъде главна и след това всяка от думите нататък трябва да бъде с главна буква. Също така е важно да се използват някакви описателни съществителни, като е добре да се избягват различни съкращения освен ако последното не е по-известно от самото описание. Това е единственото изключение.

Следващата картичка показва няколко примера за добро именуване на класове.

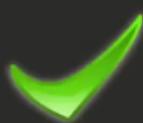
Java OOP Basics - Defining Classes - февруари 2017 - Петър Пенев

Naming Classes

The diagram shows a list of naming conventions for classes:

- Classes should be PascalCase
- Use descriptive nouns
- Avoid abbreviations (except widely known, e.g. URL, HTTP, etc.)

```
class Dice { ... }  
class BankAccount { ... }  
class IntegerCalculator { ... }
```



Примери за лошо именуване на класове са показани на следващата снимка.

```
class TPMF { ... }  
class bankaccount { ... }  
class intcalc { ... }
```



Class members(Членове на класа) Един клас може да има в себе си различни неща - конструктор, полета, методи. Полетата на класа представляват просто едни променливи, които съдържат състоянието на обекта от класа, от който сме го създали. Нещата заобградени с жълто в следващата картичка се наричат полета.

Java OOP Basics - Defining Classes - февруари 2017 - Петър Пенев



- Class is made up of state and behavior
- Fields store state
- Methods describe behaviour

```
class Dice {  
    int sides; // Fields  
    String type;  
}
```

Тези полета въобще пазят състоянието на обекта.

Също така методите могат да има и поведение. Поведението се дефинира чрез методи. Това е показано на следващата картичка, където се вижда как методът е дефиниран в класа.

- Methods describe behaviour

```
class Dice {  
    int sides; // Fields  
    String type;  
  
    void roll(){ ... } // Method  
}
```

Creating an object - Създаване на обект от клас, който вече сме създали.

Когато вече сме описали структурата, тоест направи ли сме вече шаблона(класа) идва време да си направим екземпляр от класа или т.нр. обект. Как се прави това ? Просто се изписва на първо място името на класа. След това името на променливата и после чрез ключовата дума **new** се създава нов обект от този клас.

Един клас може да има много инстанции, екземпляри от класа или обекти.(И трите неща означават едно и също в случая). В следващата картичка са показани 2 нови обекта от клас Dice.

Java OOP Basics - Defining Classes - февруари 2017 - Петър Пенев

Creating an Object

SOFTWARE UNIVERSITY FOUNDATION

- A class can have many instances (objects)

```
class Program {  
    public static void main(String args) {  
        Dice diceD6 = new Dice();  
        Dice diceD8 = new Dice();  
    }  
}
```

Variable stores a reference

Use the **new** keyword

Application in a separate file

21:03 / 2:35:38

HD YouTube

Object reference - референция към обекта Какво става когато създавам обект от даден клас ? Създаваме си променлива, създаваме си обекта с ключовата дума **new** , обаче какво става с променливата, която ни пази обекта. В случая diceD6. Тя не пази целия обект, а само една референция, която се запазва в програмния стек. (*Стекът ни пази информация за локалните променливи и докъде е стигнало изпълнението на програмата. Когато имаме някакви по-големи и по-обемни неща, които няма как да слагаме в стека, защото той има ограничен размер. Те се слагат на друго място в паметта наречено Heap*) Този пример е показан на следващата картичка.

Java OOP Basics - Defining Classes - февруари 2017 - Петър Пенев

Object Reference

SOFTWARE UNIVERSITY FOUNDATION

- Declaring a variable creates a **reference** in the stack
- new** keyword allocates memory on the heap

```
Dice diceD6 = new Dice();
```

Reference has a fixed size

diceD6 (1540e19d)

type = null
sides = 0

HD YouTube

Classes Vs Objects Каква е разликата между класовете и обектите ?

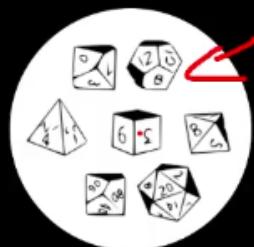
Java OOP Basics - Defining Classes - февруари 2017 - Петър Пенев

Classes vs. Objects



- Classes provide structure for describing and creating objects
- An object is a single instance of a class

Dice is...



Dice ADT

Dice (Class)

D6 Dice
(Object)

Винаги първо тръгваме от абстрактния тип данни и след него към имплементацията на дадения клас. В случай клас зарче. Тя(имплементацията) просто описва структурата. И вече от този клас можем да създадем обект, който репрезентира точно структурата на класа(просто създаваме екземпляр от класа).

Java OOP Basics - Defining Classes - февруари 2017 - Петър Пенев

Classes vs. Objects (2)



Objects

object
diceD6

type = "six sided"
sides = 6

Class name

Classes

class
Dice

Class data

type: String
sides: int

roll(...)

Class actions
(methods)

Class Data Как можем да пазим информация в класа ?

Това се постига чрез полета. Те са просто променливи като всяка една от тях си има тип и име. И понеже са деклариирани в тялото на класа те са негови членове. Едно поле може

- Class fields have type and name

```
class Dice {  
    String type; ←  
    int sides;  
    int[] rollFrequency;  
    Person owner;  
  
    ...  
}
```

МНОГО Е ВАЖНО ДА ВНИМАВАМЕ КАК СИ КРЪЩАВАМЕ КЛАСОВЕТЕ. НЕ САМО Е ДОБРА ПРАКТИКА ДА СЕ КРЪЩАВАТ ОПИСАТЕЛНО И С ДОБРИ ИМENA, НО И НА ИЗПITAЩЕ ПРОВЕРЯВАТ ЗА ТОВА.

Modifiers(Модификатори) - **Какво представляват?** Това са ключови думи, които се слагат преди променливата или дори преди самия клас, които ни казват каква ще бъде неговата видимост(Това представлява откъде дадената променлива или клас ще бъде видим. Примерно някой потребител използва дадения обект или клас, какво ще може да вижда от него и дали изобщо ще може да вижда самия клас.)

- Classes and class members have modifiers
- Modifiers define visibility

```
public class Dice {  
    private int sides;  
    public void roll(int amount);  
}
```

Примерно в примера от горната картичка дадения клас Dice отпред името на класа е зададен модификаторът `public`, който казва, че този клас ще бъде видим навсякъде. Тоест ако някой използва нашата библиотека, в която сме си направили този клас, той ще може да вижда този клас и да го използва. Примерно на дадения по-горе пример полето, което пази броя на стените на зарчето - `private int sides;`, е с модификатор `private`, което означава че само класът може да вижда това поле. Потребител няма как да има достъп до него. Следващите няколко картички демонстрират видимостта на променливите чрез модификаторите `public` и `private`.

The screenshot shows the IntelliJ IDEA interface with the project 'Java OOP Basics' open. The 'src' folder contains 'BankAccount.java'. The code defines a public class 'BankAccount' with two public fields: 'id' (int) and 'balance' (double). The code editor has syntax highlighting and a status bar indicating 'Platform and Plugin Updates'.

```
public class BankAccount {  
    public int id;  
    public double balance;  
}
```

The screenshot shows the IntelliJ IDEA interface with the project 'Java OOP Basics' open. The 'src' folder contains 'Main.java'. The code defines a public class 'Main' with a main method. Inside the main method, a new 'BankAccount' object is created and assigned to 'account'. The 'id' field is set to 1 and the 'balance' field is set to 15. A printf statement is used to print the account ID and balance. The code editor has syntax highlighting and a status bar indicating 'Platform and Plugin Updates'.

```
public class Main {  
  
    public static void main(String[] args) {  
        BankAccount account = new BankAccount();  
        account.id = 1;  
        account.balance = 15;  
  
        System.out.printf("Accout %d, balance: %.2f",  
                          account.id,  
                          account.balance);  
    }  
}
```

С модификаторът `public` се вижда ясно как имаме достъп до полетата и как лесно можем да им променяме или задаваме стойности.

На следващите картинки е пример с модификатор `private`.

The screenshot shows the IntelliJ IDEA interface with the project 'Java OOP Basics' open. The 'src' folder contains 'BankAccount.java'. The code defines a public class 'BankAccount' with two private fields: 'id' (int) and 'balance' (double). The code editor has syntax highlighting and a status bar indicating 'Platform and Plugin Updates'.

```
public class BankAccount {  
    private int id;  
    private double balance;  
}
```

The screenshot shows the IntelliJ IDEA interface with a Java project named 'DCDemo'. The 'Main.java' file is open, displaying the following code:

```
public class Main {  
    public static void main(String[] args) {  
        BankAccount account = new BankAccount();  
        account.id = 1;  
        account.balance = 15;  
  
        System.out.printf("Accout %d, balance: %.2f",  
                           account.id,  
                           account.balance);  
    }  
}
```

Тук ясно се вижда, че с модификатор `private` вече нямаме директен достъп до полетата и не можем пряко да им сменяме стойностите както си искали за разлика от предния пример.

МНОГО ВАЖНО!!!!!!!!!!!!!! ВСИЧКИ ПОЛЕТА, КОИТО ДЕФИНИРАМЕ, ТРЯБВА ДА БЪДАТ

PRIVATE!!

Това се прави с цел, когато правим някакъв качествен клас, да скрием колкото можем повече информация от потребителя. Тоест имплементацията на този клас, потребителя не би трябвало да го интересува.

Methods - След като сме скрили от потребителя всички полета, идва въпросът по какъв начин да му дадем възможност той да си променя състоянието (state) на обекта. Това става с т.нар. **гетъри и сетъри**(getters and setters), които са просто методи.

Getters and Setters

Как да си променяме състоянието (state-a) на обекта, когато сме си декларирали полетата като `private`? Това става с едни методи, които по конвенция се наричат гетъри и сетъри(getters and setters). Срещат се и с имената accessors и mutators.

The diagram illustrates the creation of accessors and mutators (getters and setters) for a `Dice` class. The code shown is:

```
class Dice {  
    private int sides;  
    public int getSides() {  
        return this.sides;  
    }  
  
    public void setSides(int sides) {  
        this.sides = sides;  
    }  
}
```

Annotations explain the code:

- A callout bubble points to the `sides` field declaration with the text "Field is hidden".
- A callout bubble points to the `getSides()` method with the text "Getter provides access to field".



Constructors

Java OOP Basics - Defining Classes - февруари 2017 - Петър Пенев

Constructors



- Special methods, executed during object creation

```
class Dice {  
    int sides;  
  
    public Dice() {    }  
        this.sides = 6;  
    }  
}
```

Overloading default constructor

Това са едни специални методи, които нямат return type (тоест не връщат никакъв резултат), и които ни служат за инициализиране на самия обект. При създаването на някакъв обект конструкторът е първият метод, който се изпълнява. Конструкторите понеже са методи, могат и да приемат параметри. Ако си направим свой собствен конструктор, то нашия презписва този, който е по подразбиране (default) и ще се извика нашия вместо този по подразбиране. На следващата картичка е показан конструктор, който приема параметри.

Java OOP Basics - Defining Classes - февруари 2017 - Петър Пенев

Constructors (2)



- You can have multiple constructors in the same class

```
class Dice {  
    int sides;  
  
    public Dice() {  
        this.sides = 6;  
    }  
  
    public Dice(int sides) {  
        this.sides = sides;  
    }  
}
```

new Dice

Ако искаме да извикаме примерно първия конструктор от картицата по-горе, просто трябва да напишем **new Dice()**; Ако искаме обаче да извикаме втория, понеже той очаква да му подадем някаква стойност, защото приема параметри, трябва да го извикаме по следния начин **new Dice(2);**

Другото много важно нещо, което правят конструкторите е, че те ни задават първоначалното състояние на обектите, тъй като те са методите, които първи се извикват при създаването на обект от даден клас.

Object Initial State

- Constructors set object's initial state

```
class Dice {
    int sides;
    int[] rollFrequency;

    public Dice(int sides) {
        this.sides = sides;
        this.rollFrequency = new int[sides];
    }
}
```

Constructor Chaining

- Constructors can call each other

```
class Dice {
    int sides;
    public Dice() {
        this(6);
    }
    public Dice(int sides) {
        this.sides = sides;
    }
}
```

Calls constructor with parameters

6 should be declared in a final variable

Static members Представляват членове от нашия клас, които са споделени с него. Те не са конкретно към никой създаден обект, а са споделени за всички обекти. Примерно ако искаме да си запазим бройката на всички създадени акаунти някога в нашата програма. Това не е информация, която да се пази в инстанцията на нашия клас. Не би трябвало един обект да знае колко такива преди него са създадени. Затова можем да имаме такава променлива в класа, която да бъде обща за класа, а не за инстанциите на обектите. Това става с ключовата дума **static**. (Показано е в следващата картичка)

Static Members

- Static members are shared class-wide

```
class BankAccount {
    private static int accountsCount:
```

}

Static Members

- Static members are shared class-wide

```
class BankAccount {  
    private static int accountsCount;  
  
    public BankAccount() {  
        accountsCount++;  
    }  
    ...  
}
```

Като си направим подобна променлива, тя не може да бъде достъпена от инстанцията на обекта. Как се достъпва това ? С името на класа, в случая от горната картичка пишем името на класа - BankAccount и след него веднага дописваме '.' и ще ни се появи статичната променлива.