

петък, 24.02.2017

Encapsulation

Енкапсулацията ни помага да скрием имплементацията на нашия клас както и информацията в него.


Java OOP Basics - Encapsulation - февруари 2017 - Венцислав Иванов

Encapsulation

SOFTWARE UNIVERSITY
FOUNDATION


- Process of **wrapping** code and data together into a single **unit**
- Objects fields **must be private**

```
class Person {  
    private int age;  
}
```



- Use **getters** and **setters** for data access

```
class Person {  
    public int getAge()  
    public void setAge()  
}
```



Основното **свойство** на енкапсулацията е да обием нашия код в един обект и само той да може да борава със своята чувствителна информация. Идеята е, че когато създаваме обект ние преценяваме колко имплементацията му, колко от методите и променливите да покажем пред целия свят. **Трябва максимално много да държим заключена нашата информация и нашите методи**. Или казано по друг начин колкото другите обекти знаят по-малко за нас, толкова по-добре. **__ПОЛЕНЦАТА ОТ НАШИТЕ КЛАСОВЕ ТРЯБВА ЗАДЪЛЖИТЕЛНО ДА БЪДАТ PRIVATE. ТОВА Е ПЪРВО И НАЙ-ОСНОВНО ПРАВИЛО НА ЕНКАПСУЛАЦИЯТА__**. Достъпването до информацията от тези поленца се осъществява чрез два метода. **Достъпването** се осъществява чрез **getter-a** (гетъра), а **презаписването** през **setter-a** (сетъра). На горната картинка са показани гетъра и сетъра. Имената им са същите като на полетата само че отпред започват евентуално с get или set в зависимост от метода, следвани от името на полето с главна буква. Такава е конвенцията. Тук трябва само да се прецени, че гетъра винаги ни връща някакви данни, а сетъра понеже слага, а не взема информация, то той в повечето случаи е void. Това не е по конвенция и трябва да се прецени, докато се дизайнва(проектираме) класа.

За валидация на данните има 2 варианта или тя да е сетъра, или в конструктора.

Keyword this

Java OOP Basics - Encapsulation - февруари 2017 - Венцислав Иванов

Keyword this

SOFTWARE UNIVERSITY
FOUNDATION

- **this** is reference to the **current object**
- **this** can refer current class instance variable

```
public Person(String name) {
    this.name = name;
}
```

- **this** can invoke current class method

```
private String getFirstName() { return this.fname }
public String fullName() {
    return this.getFirstName() + " " + this.getLastName()
}
```

Ключовата дума **this** в един клас показва конкретната инстанция на този клас.

Примерно ако инстанцираме един обект от клас Person (показан на горната картинка), ключовата дума **this** е само за този Person. Ако след това създадем още един обект от същия клас, то **this** ще се отнася вече за него. Когато използваме **this** с някаква променлива, която е част от обекта, тогава ние всъщност сочим към **полето**. Също така чрез думичката **this** можем и да извикваме методи, тоест такива на конкретния обект.

Java OOP Basics - Encapsulation - февруари 2017 - Венцислав Иванов

Keyword **this** (2)



- **this** can invoke current class constructor

```
public Person(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
}
public Person (String fname, String lname, Integer age) {
    this(fname, lname);
    this.age = age;
}
```

- **this** can be pass like argument in method or constructor call
- **this** can be returned from method

Когато долният клас наследява горния, ние можем да го извикаме посредством **this**. Ако имаме 2 конструктора (както е показано на горната снимка), чрез **this** можем да викнем конкретния конструктор. Има случаи макар и по-редки, когато **this** от метод в наш клас трябва да работи с целия наш клас. (Това ще се учи по-подробно в Java OOP Advanced).

Имаме и методи, които искаме когато се връща някакъв резултат, то той да бъде точно този обект. Правят се разни промени по state-а (състоянието) на обекта и накрая се връща абсолютно целия обект.

Ключовата дума **this** се използва най-основно за:

1. за променливи;
2. за методи;
3. за конструктори.
4. може да бъде подаден като аргумент (по-подробно в Java OOP Advanced)

5. може да бъде returned(върнат като резултат) (Също по-подробно в Java OOP Advanced).

Access modifiers - private, public, protected , default.

Java OOP Basics - Encapsulation - февруари 2017 - Венцислав Иванов

Private Access Modifier



- Main way that an object encapsulates itself and hides data from the outside world

```
class Person {  
    private String name;  
    Person (String name) {  
        this.name = name;  
    }  
}
```

- Class and interfaces **cannot** be private
- Can only be accessed within the declared class itself

private -> Основния начин, по който енкапсулираме данни, защото private ни дава ниво на достъп само от конкретния клас и скрива информацията от всички останали. **Важно е да се отбележи, че класове и интерфейси НЕ МОГАТ да бъдат private.** Private може да бъде достъпен само и единствено от конкретната инстанция на този клас.

Java OOP Basics - Encapsulation - февруари 2017 - Венцислав Иванов

Protected Access Modifier



- Can be accessed only by the subclasses in other package

```
class Team {  
    protected String getName ()  
    protected void setName (String name)  
}
```

- **Protected** access modifier cannot be applied to class and interfaces
- Preventing a **nonrelated** class from trying to use it

Protected -> Дава ни шанс нашите child-ве, тоест наследниците на нашия клас да пипат в нашите методи. Това се случва единствено при едно странично наследяване, когато наследникът не е в същия package. **Protected** дава достъп на нашите наследници да пипат по нашите методи. Нямаме право, никое от поленцата да бъде **protected**. Ако нашия наследник трябва да пипа чувствителна информация, трябва да направим **protected setter** или **getter**, а поленцата си остават **private**. (Показано е на горната картинка). Също както и **private**. класовете и интерфейсите не могат да бъдат

protected. Protected се използва единствено за методи, които искаме нашите наследници да знаят за тях, но само те.

Default access modifier - той е когато не пишем никакъв modifier.

Java OOP Basics - Encapsulation - февруари 2017 - Венцислав Иванов

Default Access Modifier



- Do not explicitly declare an access modifier

```
class Team {  
    String getName ()  
    void setName (String name)  
}
```

- Available to any other class in the same package

```
Team rm = new Team("Real");  
rm.setName("Real Madrid");  
System.out.println(rm.getName());  
//Real Madrid
```

В този случай достъп имат другите участници в този package, независимо дали са наши деца или не. **Лекторът съветва да не го използваме и да не стигаме до ниво да се питаме дали ни е този или в друг package.** Девелопъра, който ще седне да пише по нашия код след като ние спрем, не трябва да му оставяме такава дилема. Кое, къде, в кой package се намира. Нещата могат много да се объркат. Примерно вземаме класа Team от горната картинка и го местим в друг package и .. край.. моят клас Team става неизползваем, докато не му се наслагат modifier-и навсякъде и да се изгуби допълнително време в глупости.. Default-а може да бъде използван върху класове и интерфейси. Конкретния клас може да бъде инстанциран само в останалите членове на конкретния package.

Public access modifier

Java OOP Basics - Encapsulation - февруари 2017 - Венцислав Иванов

Public Access Modifier



- A Class, method, constructor **declared** inside a **public** class can be **accessed** from **any class** belonging to the **Java Universe**

```
public class Team {  
    public String getName ()  
    public void setName (String name)  
}
```

- Imports are needed if we try to access public class in different package
- The **main()** method of an application has to be public

Дава достъп на всеки до нашия клас. За интерфейси и класове това е най-често срещаният modifier. Въпреки, че е public ако сме в друг package , все още се нуждаем от import.

Структура на даден клас:\\ 1.Винаги се запозва с декларирането на полета, които винаги са private и са най-отгоре.

```
private String firstName;  
private String lastName;  
private int age;  
private Double salary;
```

2.Създаване на един или повече конструктори.

```
public Person(String firstName, String lastName, Integer age, Double salary) {  
    this.firstName = firstName;  
    this.lastName = lastName;  
    this.age = age;  
    this.salary = salary;  
}
```

3.След това са методите, и в частност най-отгоре са винаги гетърите и сетърите като се редуват, тоест гетър-сетър на дадено поле. След тях вече започваме със static, private, public методи.


```
public String getFirstName() { return this.firstName; }  
  
private String getLastName () { return this.lastName; }  
  
public int getAge() { return this.age; }  
  
public Double getSalary() { return this.salary; }
```

Константите трябва да бъдат в друг трети общ клас. Лекторът съветва да се пазим от static. Колкото по-малко static имаме в приложението си, толкова по-щастливи ще бъдем.

Validation

За какво още е важна енкапсулацията ? - Важна е за т.нар. валидация.

Validation



- Data validation happen in setters

```
private void setSalary(Double salary) {  
    if (salary < 460) {  
        throw new IllegalArgumentException  
            ("Salary cannot be less than 460 leva");  
    }  
    this.salary = salary;  
}
```

Better throw exception,
than print to Console

- Don't couple your class with Console
- Contributor of your class have to think about handle Exceptions

Всеки един обект си има състояние(state) - това са неговите стойности, които той поддържа във всеки един момент от живота на нашия object. От нас като програмисти

зависи да поддържахме нашия обект винаги във валиден state. Това нещо обикновено се осъществява чрез сетърите, където валидираме дали нашите данни са валидни и гарантираме, че нашият state не може да е невалиден. Това се случва по 2 начина:

- Като валидираме нашите данни още на ниво в конструктора. (Тук има един по-специфичен случай, когато се използва ключовата дума final).
- Но по-често използваното е на ниво setter(сетър) на даденото поле.

В конкретния пример, който е с salary от горната картинка, тази променлива няма как да е под 460 лв чрез тази проверка, която правим и след нея имаме 2 варианта:

- Единия вариант е да напишем `System.out.println("");` и да напишем в конзолата някакво съобщение, в случая `Salary cannot be less than 460 lv`.
- **Вторият ни вариант е да хвърлим exception. (throw new Exception).**

По-правилният и по КПК(Качествен Програмен Код) вариант е да хвърлим exception. Абсолютно задължение е на user-а на нашия клас е да си прихване(handle-не) exception-ите.

Validation (2)

Constructors use private setter with validation logic

```
public Person(String firstName, String lastName,
               Integer age, Double salary) {
    setFirstName(firstName);
    setLastName(lastName);
    setAge(age);
    setSalary(salary);
}
```

Validation is happen inside of setter

- Guarantee valid state of object in its creation
- Guarantee valid state for public setters

В примера от горната картинка просто в конструктора сме си извикали сетърите и по-този начин изнасяме валидацията на данните у тях. Като във всеки сетър си пишем валидация към конкретното поле. По този начин гарантираме, че то винаги ще бъде във валиден state.

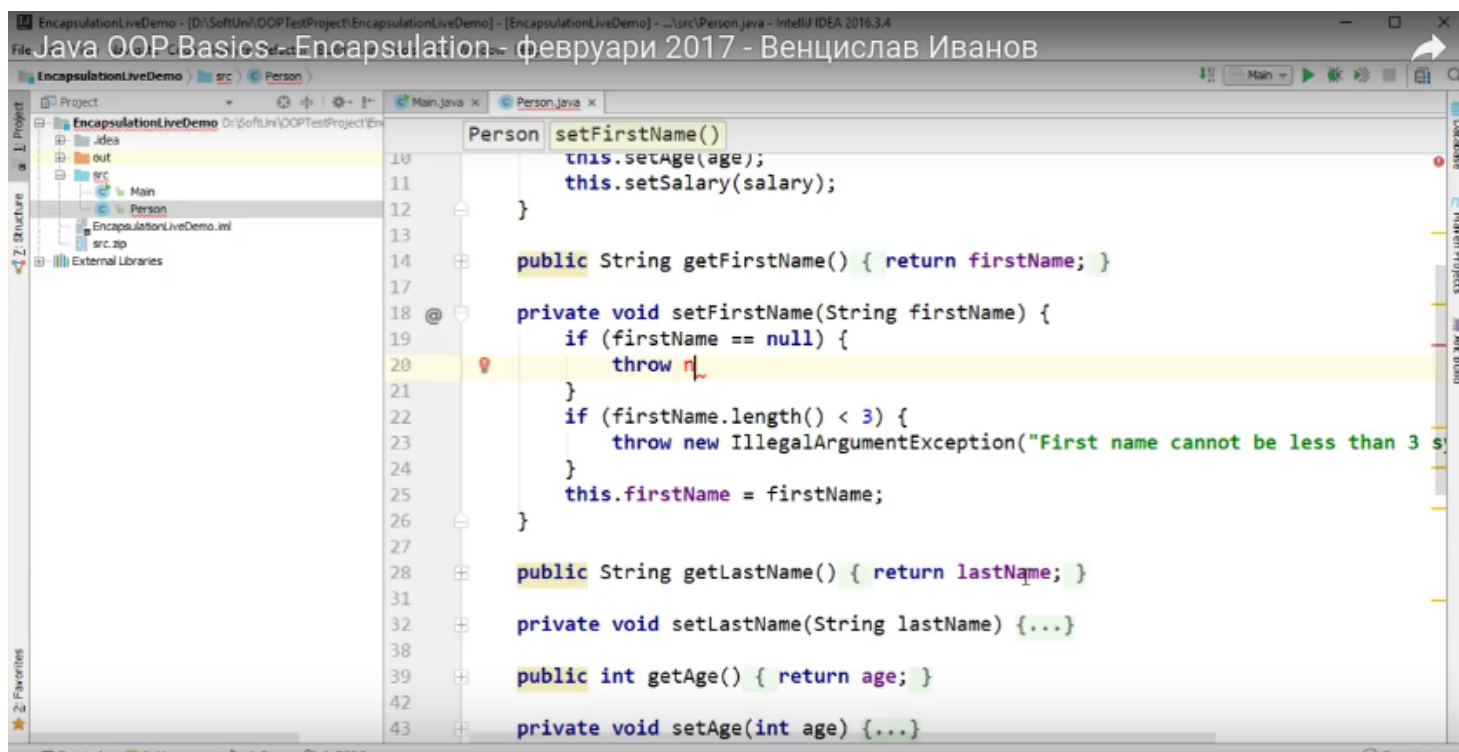
В следващата картинка е показан един интересен случай:

```
18 private void setFirstName(String firstName) {
19
20     if (firstName.length() < 3) {
21         throw new IllegalArgumentException("First name cannot be less than 3 s
22     }
23     this.firstName = firstName;
24 }
```

Exception in thread "main" java.lang.NullPointerException
at Person.setFirstName(Person.java:19)
at Person.<init>(Person.java:8)
at Main.main(Main.java:21) <5 internal calls>

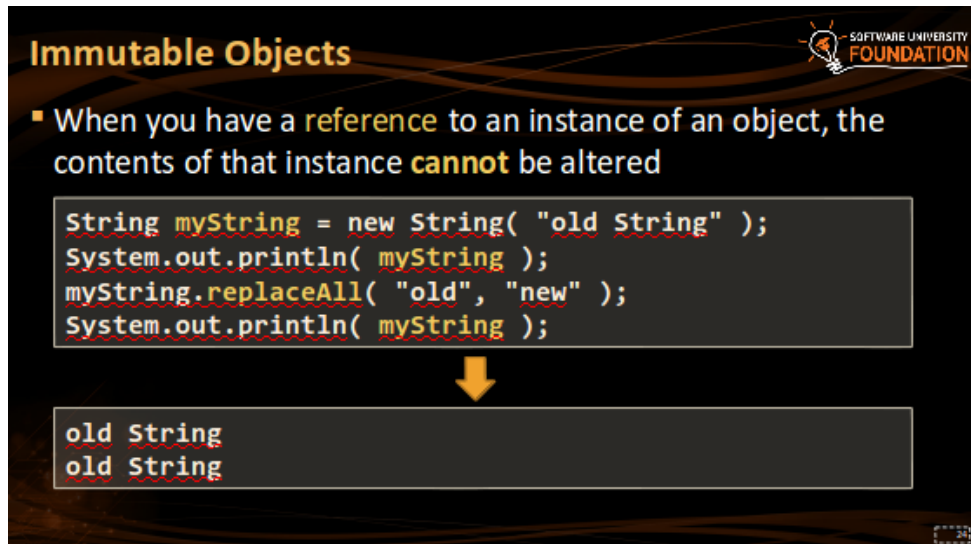
Понеже String е nullable тип, тоест който може да приема null като стойност и в момента

exception-а, който се хвърля на картинката по-горе е точно на реда преди проверката - 20 ред и въобще не стига до нашия exception. За това в такива случаи, най-адекватната валидация е още първата проверка да проверява дали стойността на String променливата не е null. По принцип за String полета, най-първата проверка трябва да бъде точно за null стойност, защото в противен случай ни хвърля NullPointerException. (Това е показано на следващата картинка).



```
10  public void setFirstAndLastName(String firstName, String lastName) {
11      this.setAge(age);
12      this.setSalary(salary);
13  }
14  public String getFirstName() { return firstName; }
15
16  public void setFirstName(String firstName) {
17      if (firstName == null) {
18          throw new NullPointerException("First name cannot be null");
19      }
20      if (firstName.length() < 3) {
21          throw new IllegalArgumentException("First name cannot be less than 3 s");
22      }
23      this.firstName = firstName;
24  }
25
26  public String getLastName() { return lastName; }
27
28  public void setLastName(String lastName) { ... }
29
30  public int getAge() { return age; }
31
32  public void setAge(int age) { ... }
```

Immutable Objects



Immutable Objects

- When you have a **reference** to an instance of an object, the contents of that instance **cannot** be altered

```
String myString = new String( "old String" );
System.out.println( myString );
myString.replaceAll( "old", "new" );
System.out.println( myString );
```

↓

```
old String
old String
```

Обектите могат да бъдат разделени на два вида - Immutable(които не могат да се променят) и Mutable(които можем да променяме). Пример за обект, който не можем да му променим състоянието, това е String-ът. На примерът на горната снимка се вижда как String-ът myString си остава един и същ, тоест непроменен.

Още един пример, който е на следващата картинка: Имаме стринг променлива f със стойност "hello" и след това към същата се опитваме да добавим стойността "pesho". И сега ако изпринтираме променливата ще излезе като "hello pesho" и може да си

сега ако изтрием променливата ще излезе като нещо ново и може да си помислим, че сме променили string-а. Само че не е така. Идеята е, че стрингът пази в heap-а референция към гат паметта къде точно се пази този стринг и в момента, и когато напишем '+' на стринга, тоест конкатенираме го, точно в този момент в heap-а тази референция към конкретната клетка в рама се изтрива и стрингът се записва в изцяло нова клетка. **На кратко: ние замествахме целия стринг. НЕ ГО МОДИФИЦИРАМЕ, ПРОМЕНЯМЕ ДАННИТЕ МУ КАТО СТРИНГ, А ГО ЗАМЕСТВАМЕ С ИЗЦЯЛО НОВИЯТ СТРИНГ.**

```
String f = "hello";  
f += "pesho";
```

Mutable objects

Mutable Objects

- When you have a **reference** to an instance of an object, the contents of that instance **can** be altered

```
Point myPoint = new Point( 0, 0 );  
System.out.println( myPoint );  
myPoint.setLocation( 1.0, 0.0 );  
System.out.println( myPoint );
```

↓

```
java.awt.Point[0.0, 0.0]  
java.awt.Point[1.0, 0.0]
```

Когато имаме референция(адрес) към инстанция на обект, съдържанието в инстанцията може да се променя, както е показано на горната картинка. Идеята е, че този тип данни muttables, могат да променят състоянието на обектите и се запазват на едно и също място в рам паметта. Просто сменят стойността си. Mutable class са всички колекции, които познаваме като например - List<T>.

Mutable fields Да кажем че имаме едно поленце, което ни е лист от Person - List<Person> , какво се случва когато върнем (return) чрез getter листа от Person наречен players. По този начин ние връщаме референцията, което означава, че върху този лист ние можем да правим всичко.(Примерът е показан на следващата картинка).Това не е грешно. Но тук всичко опира до дизайн(проектирането) на самият клас и дали желаем някой да може да редактира по някакъв начин листа ни. Просто трябва да знаем, че когато го върнем без нищо както е на картинката, ние реално връщаме правата за използването на нашия user. Тоест той може да направи с нашия лист абсолютно всичко. Въпросът е да знаем, че му даваме целите права върху информацията на самия обект.

Mutable Fields

- private** mutable fields are still don't encapsulated

```
class Team {  
    private String name;  
    private List<Person> players;  
  
    public List<Person> getPlayers() {  
        return this.players;  
    }  
}
```

- In this case **getter is setter too**

Ако не искаме да даваме на потребителя възможност да пипа по нашия лист, тоест не искаме да му дадем правата да редактира обекта ни, тогава трябва да му върнем **`Collections.unmodifiableList(players)`**. Последното връща заключен лист, в който не можем да ползваме повечето от методите на листа. Може да се итерира(обхожда) и да му се вземат стойностите, но не могат да се редактират. Проблемът до който води това е , че не могат да се добавят нови елементи в листа. Не може да се напише примерно `players.add(player)`. Затова изнасяме нов метод в нашия клас, който добавя `players` на ниво `object` в нашия клас, а на света връща лист, който е на практика заключен. Не може да се променя. Всичко това е показано на следващата картинка.

Mutable Fields (2)

For securing our collection we can return `Collections.unmodifiableList()`

```
class Team {
    private List<Person> players;
    public addPlayer(Person person) {
        this.players.add(person);
    }
    public List<Person> getPlayers() {
        return Collections.unmodifiableList(players);
    }
}
```

Add new methods for functionality over list

Return safe collections

И все пак зависи с каква цел връщаме листа. Ако е само, за да се покажат данните в нето, то по-добър вариант би бил да се върне `iterable`. Последното дава възможност единствено да се итерира, разглежда и нищо повече.

Keyword final

Keyword final

`final class` can't be extended

```
public class Animal {}
public final class Mammal extends Animal {}
public class Cat extends Mammal {}
```

`final method` can't be overridden

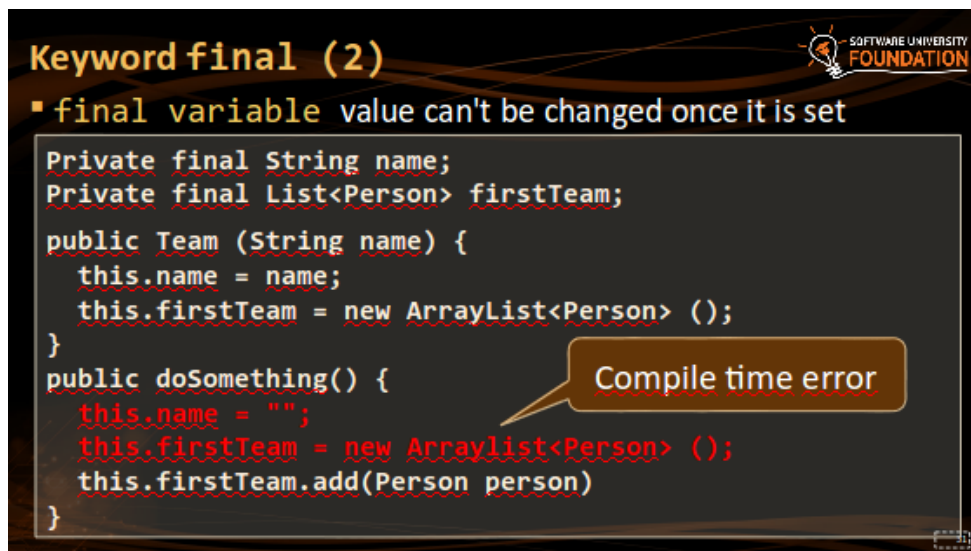
```
public class Animal {
    public final move(Point point) }
public class Mammal extends Animal {
    @override
    public move() }
```

Тази ключова дума може да се използва на няколко места в java:

- Класове(Classes) - забранява този клас да бъде наследяван;
- Методи(Methods) - в този случай забранява методът да бъде override-нат(презаписан). Примери са показани на горната картинка.;
- Променлива(Variable) .

Когато се използва върху променлива има два варианта:

- Единият вариант е да се инициализира на момента при създаването на променливата.
- Вторият начин е да се дефинира променливата като final, но да се инициализира в конструктора. Веднъж инициализирана(тоест и е зададена някаква стойност) повече не може по никакъв начин да се променя тази стойност.



Keyword final (2)

▪ **final variable** value can't be changed once it is set

```
Private final String name;  
Private final List<Person> firstTeam;  
public Team (String name) {  
    this.name = name;  
    this.firstTeam = new ArrayList<Person> ();  
}  
public doSomething() {  
    this.name = "";  
    this.firstTeam = new ArrayList<Person> ();  
    this.firstTeam.add(Person person)  
}
```

Compile time error

SOFTWARE UNIVERSITY
FOUNDATION