**Encapsulation**

(hiding implementation)

* **Encapsulation**
  + Process of wrapping code and data together into a single unit
  + Objects fields must be private
  + Use getters and setters for data access
* Keyword **this**
  + **This** is a 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()

}

* + **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

**Access Modifiers**

* **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 itself
* **Protected** Access Modifier
  + Can be accessed only by the subclasses in other package ( child from other packages)

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
* **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

* **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

**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 the console.

* Don`t couple your class with the Console
* Contributor of your class have to think about handle Exceptions
* Constructors use private setter with validation logic

public Person(String firstName, String lastName,

Integer age, Double salary) {

setFirstName(firstName);

setLastName(lastName); **Validation is happen inside of setter**

setAge(age);

setSalary(salary);

}

* + Guarantee valid state of object in its creation
  + Guarantee valid state for public setters

**Mutable and 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

* 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]**

* 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
  + For securing our collection we can return Collections.unmodifiableList()
    - Add new methods for functionality over list
    - Return safe collections

class Team {

private List<Person> players;

public **addPlayer**(Person person) {

this.players.add(person);

}

public List<Person> getPlayers() {

return **Collections.unmodifiableList**(players);

}

}

**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() }

* 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)

}