**Optional return**

* return este optional intr-o metoda, asa cum ultima instructiune e mereu returnata:
* def test(){  
   int a = 100;  
   a;  
  }  
    
  println(test())

desi putem si folosi return

* Daca folosim void in loc de def sau un tip concret, metoda va returna null mereu

**def**

* def este defapt Object
* Parametrii unei metode pot fi scrisi si fara def
* void test( a, b){  
    
  }

**Omitting pharanteses**

* Apelarea metodelor nu necesita mereu paranteze, de ex:

void test( a, b){  
println a+b  
}  
  
test 1, 2

* Daca avem o metoda intr-o metoda, cea din interior necesita ()
* int test( a, b){  
   return a+b;  
  }  
    
  println test 1 2



**dar asa**

println test(1, 2)

* Pentru fieldurile publice, groovy creaza automat getteri si setteri
* class Test{  
   int a = 100;  
  }  
    
  def obj = new Test();  
    
  obj.setA(0);  
  println(obj.getA())

**dar**

class Test{  
 private int a = 100;  
}  
  
def obj = new Test();  
  
obj.setA(0);  
println(obj.getA())



* Cand folosim object.field = 100, se foloseste setField() automat si la get e la fel
* Daca avem doar un default constructor, putem sa specificam parametrii cu numele si valoare la crearea obiectului:
* class Test{  
   Integer a = 100;  
   Integer b;  
     
    
  }  
    
  def obj = new Test(a: 100, b:1000);  
    
  println(obj)

intai va fi apelat constructorul default, apoi setterii

* Daca mai avem un constructor definit,in afara de cel default, asta nu mai e posibil:
* class Test{  
   Integer a = 100;  
   Integer b;  
    
   Test(a){  
   this.a = a;  
   }  
   Test(){  
     
   }  
    
  }  
    
  def obj = new Test(a: 100, b:1000);  
    
  println(obj)



**Anotatii**

* @NullCheck – e pus la nivel de metoda sau constructor sau clasa, pentru a nu permite parametrilor sa fie null. Nu merge pe field, dar daca e pus la class level, se aplica asupra la toti constructorii si metodele
* @NullCheck  
  class Test{  
   Integer a = 100;  
    
   void setA(Integer a){  
   this.a = a;  
   }  
  }  
    
  def obj = new Test();  
    
  obj.setA(null);  
  println(obj.getA())

Caught: java.lang.IllegalArgumentException: a cannot be null

java.lang.IllegalArgumentException: a cannot be null

at test.pack2.Test.setA(MainClass.groovy)

at test.pack2.MainClass.run(MainClass.groovy:16)

* @ToString – groovy are si asa anotatie
* @EqualsAndHashCode – exact ca la lombok
* @Immutable – fieldurile sunt facute private si final si nu se creaza setters
* @Builder
* @Synchronized

**with and tap**

* with – permite sa nu mai tot repetam object.

server.name = application.name

server.status = status

server.sessionCount = 3

server.start()

server.stop()

vs:

server.with {

name = application.name

status = status

sessionCount = 3

start()

stop()

}

class Test{  
 private Integer a = 100;  
 private Integer b;  
  
 Integer getA() {  
 return a  
 }  
  
 void setA(Integer a) {  
 this.a = a  
 }  
  
 Integer getB() {  
 return b  
 }  
  
 void setB(Integer b) {  
 this.b = b  
 }  
}  
  
def obj = new Test(a: 100, b:1000);  
  
obj.with **{** setA(1);  
 println(getB());  
 setB(2);  
 println(getB())  
**}**

deoarece with este un closure, ultima instructiune e mereu returnata.

* **tap** – exact ca with, dar returneaza it, adica propriu zis obiectul

println obj.tap **{** setA(1);  
 println(getB());  
 setB(2);  
 println(getB())  
**}**

1000

2

test.pack2.Test(1, 2)

**switch**

* In groovy, switch accepta nu doar primitive si string, ci si closures, object type, etc.
* switch (obj){  
   case **{**it.getA() > 100**}**:  
   println("greater than 100");  
   break;  
   case String:  
   println("A string");  
   break;  
   case **{**it.getB() < 10000**}**:  
   println("less than 10000")  
   break;  
  }

**Truth**

* false este:

- null

- void

- 0

- ‘’

- empty collection

println(0 && '' && null && new ArrayList<>())

false

**Script**

Groovy nu necesita o clasa cu main() pentru a rula codul, dar chiar si daca folosim direct cod, el oricum va crea clasa si main(), va extinde clasa Script si va apela din main metoda run() ce e mostenita de la Script

println 'Hello'

int power(int n) { 2\*\*n }

println "2^6==${power(6)}"

e echivalent cu

import org.codehaus.groovy.runtime.InvokerHelper

class Main extends Script {

int power(int n) { 2\*\* n}

def run() {

println 'Hello'

println "2^6==${power(6)}"

}

static void main(String[] args) {

InvokerHelper.runScript(Main, args)

}

}

**String for enum finding**

enum Test{  
 *CAR*,  
 *BUS*}  
  
Test element = "CAR";  
println(Test.*CAR* === element)

= e echivalent cu Test.valueOf()

**asType method**

* asType method supraincarca operatorul as

class Celsius{

double value;

def asType(Class other){

if(other == Farentheit){

return new Farentheit(value: this.value\*(10/9))

}

}

}

class Farentheit{

double value;

}

Celsius obj1 = new Celsius(value: 20);

Farentheit obj2 = obj1 as Farentheit;

println obj2.value

Supraincarcam operatorul in clasa Celsius, daca vrem sa folosim as pe obiecte de tip celsius. E important sa folosim Class ca argument, sa putem specifica mai multe tipuri de obiecte la conversie.

**Optional return**

* Ultima instructiune dintr-o metoda sau closure e mereu returnata, dar folosirea lui return e adesea preferata

**asBoolean()**

* class Test{  
   Integer value;  
   def asBoolean(){  
   return value>=1000;  
   }  
  }  
  Test obj = new Test();  
  if(!obj)  
   println("Hello")
* permite sa modificam cum groovy modifica true sau false pentru un obiect

**def**

def is, in Groovy, strictly equivalent to using Object.

**metaClass**

* Orice clasa are un obiect static metaClass. El permite sa efectuam diferite operatii asupra clasei, si sa adaugam membri la runtime
* class Test{  
    
  }  
  Test.metaClass.a = 100;  
  Test obj = new Test()  
  obj.a = 100;  
    
  println(obj.a)