**Autoboxing**

* Groovy tot foloseste datele primitive, de ex

def a = 10

e o primitiva, dar el face mereu boxing in wrapper class cand e necesar. De ex:

def a = 10.toString()

El va face automat conversia din int in Integer

class Foo {

static int i

}

assert Foo.class.getDeclaredField('i').type == int.class

assert Foo.i.class != int.class && Foo.i.class == Integer.class

**Classes**

* Putem avea intr-un fisier mai multe clase publice.
* Mereu oricum se creaza o clasa cu main() identica cu numele la fisier
* Iata de ce, nu putem crea si o clasa cu acelasi nume ca a fisierului si script in fisier:
* class MyClass{  
   static void main(String[] args) {  
   println("Hello")  
   }  
  }  
  println("what")



* **Inner class pot accesa membrii privati din clasa parinte**

class Outer {

private String privateStr

def callInnerMethod() {

new Inner().methodA()

}

class Inner {

def methodA() {

println "${privateStr}."

}

}

}

**Constructor**

* Putem crea obiecte folosind un constructor definit prin 3 metode:

1. Folosind new
2. Folosind un array si as
3. Folosind doar un array

Cand folosim un array, se vor lua elementele sale si se vor pune automat in constructor in ordinea lor

Test obj1 = new Test(1,2);  
Test obj2 = **[**1,2**]** as Test;  
Test obj3 = **[**1,2**]**

toate 3 sunt identice

**Map Constructor**

class Test{  
 int a;  
 int b;  
  
}  
Test obj1 = new Test(a:1,b:2);

* Grooovy, daca nu avem niciun constructor, pe langa cel default, mai creaza unul ce ia un Map<> unde key sunt numele la variabile si value valorile lor si le atribuite
* Dcaa definim noi vreun constructor, macar si default, map constructor nu mai e generat
* Putem si asa face, e tot aceeasi:
* Test obj1 = new Test([a:1,b:2]);

**Method**

* Nu putem trimite parametrii cu nume la o metoda:
* def method(a,b,c){  
    
  }  
  method(a:100,b:5,c:2);



* **Daca primul sau ultimul parametru e un map, putem evita []**
* def method(Map map, a){  
    
  }  
  method(a:100,b:5,c:2, 100);

**e echivalent cu**

method([a:100,b:5,c:2], 100)

* **Metodele suporta default value pentru parametrii:**

def method(Map map, a=100){  
  
}

* In groovym ... sau [] ca ultim argument in metoda e tot aceeasi:

def method(int... arg){  
 return arg;  
}  
println(method(1,2,3,4))

e similar cu

def method(int[] arg){  
 return arg;  
}  
println(method(1,2,3,4))

* In caz de supraincarcare, metoda cu un parametru e aleasa:
* def method(int[] arg){  
   return "array";  
  }  
  def method(int arg){  
   return "single parameter"  
  }  
  println(method(1))

single parameter

* Daca avem 2 metode, de ex, ce contin o superclasa si o interfata implementata a unei clase, si o apelam cu o un obiect de tip copil, in Java vom avea eroare de ambiguitate, dar groovy va alege interfata mereu in loc de clasa:
* *method*(new Test());  
    
   static void method(Parent obj){  
   System.*out*.println("Parent class");  
   }  
   static void method(Interface obj){  
   System.*out*.println("Interface");  
    
   }  
    
  class Parent{  
    
  }  
  interface Interface{  
    
  }  
  class Test extends Parent implements Interface{  
    
  }

Interface

* O variabila sau field in genere poate sa nu aiba tip, sa nu fie cu def declarata:
* a = 100;  
    
  println(a)

dar asta e o practica rea!

* groovy creaza si apeleaza automat getterii si setterii cand incercam sa atribuim direct valori unui field public a unui obiect, insa asta nu mai e valabil daca accesam field direct din clasa

class Person {

String name

void name(String name) {

this.name = "Wonder $name"

}

String title() {

this.name

}

}

def p = new Person()

p.name = 'Diana'

assert p.name == 'Diana'

p.name('Woman')

assert p.title() == 'Wonder Woman'

|  |  |
| --- | --- |
|  | this.name will directly access the field because the property is accessed from within the class that defines it |
|  | similarly a read access is done directly on the name field |
|  | write access to the property is done outside of the Person class so it will implicitly call setName |
|  | read access to the property is done outside of the Person class so it will implicitly call getName |
|  | this will call the name method on Person which performs a direct access to the field |
|  | this will call the title method on Person which performs a direct read access to the field |

**Exceptions**

* In groovy, checked exceptions nu mai trebuie neaparat prinse in metoda parinte sau in block try catch.
* Nici macar throws nu poate obliga prinderea unei exceptii la metoda apelanta
* method();  
    
  def method(){  
   exception();  
  }  
    
  def exception() throws IOException{  
     
  }

**properties field**

* Orice clasa si obiect au un field properties, ce permite sa aflam detalii despre fields
* .**properties** – returneaza un map ce contine numele la field si valoarea lui din obiect
* class Test{  
   def field1;  
   def field2;  
   static def *staticVar*;  
   static def method(){  
    
   }  
  }  
  Test obj = new Test();  
    
  println(obj.properties)

[field1:null, field2:null, staticVar:null, class:class test.pack2.Test]

* .**properties.keySet()** – returnraza un set ce contine numele la fielduri.

println(obj.properties.keySet())

[field1, field2, staticVar, class]

* Daca o clasa il apeleaza, se returneaza numele la fieldurile statice, daca un obiect, atunci cele nestatice

**Pseudo properties**

* Daca nu avem un field, dar avem metode getters si setters, ce incep cu get/set, putem sa nu le apelam, dar sa le folosim ca si cum ar fi fielduri de care ele se ocupa.
* Groovy, cand vede ca folosim un field, automat incearca sa gaseasca metode get sau set pentru el, indiferent ca asa field exista sau nu
* class Test{  
   def setInexistent(def value){  
   println("Setter");  
    
   }  
   def getInexistent(){  
   return "Inexistent field";  
   }  
  }  
  Test obj = new Test();  
  obj.inexistent = "Hello";  
    
  println(obj.inexistent)

**Trait**

* Trait sunt un fel de interfete, dar care permit sa declaram metode si fielduri exact ca intr-o clasa
* Nu pot fi instantiate
* Field nu mai sunt automat final static
* Metodele nu mai sunt automat abstract, ci default
* Trait accepta doar private si public
* Clasele le implementeaza cu implements
* Trait pot implementa interfete cu implements
* Trait pot extinde alte trait cu extends
* Daca o clasa implementeaza mai multe traits ce au o metoda identica, mereu se va lua metoda ultimului trait

**this in Interfaces**

* Ca si in Java, putem folosi cuvantul this in interfete, doar ca el va fi evident obiectul instantiat si va permite accesul la tot ce se afla in scope al interfetei
* interface Interface{  
   default def me(){  
   return this;  
   }  
  }  
    
  class Test implements Interface{  
    
  }  
    
  Test obj = new Test();  
  println(obj === obj.me())

true