**?.**

def obj = new Test(a: null,b: 100);  
println(obj?.a?.*MAX\_VALUE*);

Daca vreun element e null, null e returnat

**?:**

def result = name != null ? name : "Unknown"

poate fi mai usor scris ca

def result = name ?: "Unknown"

**.&**

* Acest operator ne permite sa returnam un pointer al undei functii si deci sa o executam
* Orice referinta la o functie este de tip Closure
* def function(String arg){  
   println(arg);  
  }  
    
  Closure closure = this.&function;  
    
  closure("Hello");
* Acest operator face ca metoda sa fie aleasa la runtime, nu compiletime, de aceea, daca folosim overloading, oricum metoda potrivita se va alege la runtime pentru fiecare call

def function(String arg){  
 println("Un string");  
}  
def function(int arg){  
 println("Un int")  
}  
**~**

* Acest operator permite crearea unui obiect de tip regexp, si anume Pattern
* Poate fi folosit si cu // si ‘’ su “” etc.
* def regExp = ~/1001.\*/  
  println(regExp instanceof Pattern)

true

**=~**

* Apeleaza automat metoda mattcher() a unui Patterm si returneaza un Matcher
* Pattern p = ~/.\*test/  
  Matcher m = "This is a test" =~ p;  
    
  println(m.matches())

**==~**

* Returneaza un boolean, deci apeleaza mattcher().matches()

Pattern p = ~/.\*test/  
Boolean m = "This is a test" ==~ p;  
  
println(m)

**\*.**

* Operatorul dat, numit spread operator, e ca o alternativa pentru map() in stream, desi e mai limitat
* El permite returnarea unei noi liste ce contine elemente create dintr-un field a tuturor elementelor unei liste
* class Student{  
   String firstName;  
   String lastName;  
   String age;  
  }  
    
  def list = [new Student(firstName: "Mititiuc",lastName: "Eduatd",age: 21),  
   new Student(firstName: "Petru",lastName: "Turcan",age: 20),  
   new Student(firstName: "Goncear",lastName: "Alex",age: 25),  
  ]  
  println(list\*.firstName)

[Mititiuc, Petru, Goncear]

list\*.firstName va returna o lista noua ce contine elemente formate din firstName la fiecare obiect din lista

* este echivalenta cu **collect({it.field})**
* println(list.collect(**{**it.firstName**}**))
* este thread safe. Daca un element din lista e null, va arunca null, nu exception

assert null\*.make == null

**\***

* separa toate elementele unei liste
* def function(num1, num2, num3){  
   return num1+num2+num3;  
  }  
    
  println function(\*[1,2,3])

6

deci va face ca [1,2,3] sa devina 3 valori separate

def array = [0,1,2,3]  
println([\*array,4,5,6])

[0,1,2,3,4,5,6]

* Putem sa il folosim si pentru map asa \*:map
* def map = [one : 1, two : 2]  
  println([zero : 0, \*:map, three : 3])

[zero:0, one:1, two:2, three:3]

**.. Range**

* Groovy are o clasa IntRange, ce ofera metoda **collect**() pentru a returna o List
* IntRange range = new IntRange(1,5);  
  println(range.collect())

[1, 2, 3, 4, 5]

* Operatorul .. e o prescurtare

IntRange range = 0..10;  
println(range instanceof IntRange)

true

* Putem folosi si < intre ..
* IntRange range = 0<..<10;  
  println(range.collect())

asa 10 si 0 sunt excluse

**<=> prescurtare la compareTo()**

println(1 <=> 10)

-1

**?[] este echivalent cu ?. doar e ca pentru array**

* Daca index oferit nu exista returneaza null
* def array = [1,2,3,4]  
  println(array?[8])

null

* Daca incercam sa setam un index inexistent, nu se intampla nimic
* def array = [1,2,3,4]  
  array?[10] = 10;  
  println(array)

**() = call()**

* Daca un obiect are metoda call(), nu mai trebuie sa folosim .call(), ci doar ()
* class Test{  
   def call(a){  
   println(a);  
   }  
  }  
    
  def obj = new Test();  
  obj(10)

**Operator overload**

* In groovy putem suprascrie operatorii. Doar folosim metodele plus(other), minus(other) etc.
* Tipul argumentului depinde de ce vrem noi sa punem dupa un operator

| **Operator** | **Method** | **Operator** | **Method** |
| --- | --- | --- | --- |
| + | a.plus(b) | a[b] | a.getAt(b) |
| - | a.minus(b) | a[b] = c | a.putAt(b, c) |
| \* | a.multiply(b) | a in b | b.isCase(a) |
| / | a.div(b) | << | a.leftShift(b) |
| % | a.mod(b) | >> | a.rightShift(b) |
| \*\* | a.power(b) | >>> | a.rightShiftUnsigned(b) |
| | | a.or(b) | ++ | a.next() |
| & | a.and(b) | -- | a.previous() |
| ^ | a.xor(b) | +a | a.positive() |
| as | a.asType(b) | -a | a.negative() |
| a() | a.call() | ~a | a.bitwiseNegate() |

class Test{  
 int money;  
 def plus(int other){  
 return this.money + other;  
 }  
 def plus(Test other){  
 return this.money + other.money;  
 }  
}  
  
Test obj1 = new Test(money: 100);  
Test obj2 = new Test(money: 1000);  
  
println(obj1 + 100);  
println(obj1 + obj2);