**Block**

* Multime de cod intre { } din orice clasa, metoda sau constructor
* Un bloc poate fi pus in alt bloc, si e numit nested bloc
* public class Main {  
   public static void main(String[] args) {  
   System.*out*.println("Hello world!");  
   {  
   System.*out*.println("A block");  
   {  
   System.*out*.println("Another nested block");  
   }  
   }  
   }  
  }
* Variabilele dintr-in bloc sunt locale. Deci, pot fi accesate doar in acel block, sau in blocuri inner, dar nu in blocuri externe:
* public class Main {  
   public static void main(String[] args) {  
   System.*out*.println("Hello world!");  
   int a = 100;  
   {  
   int b = a+10;  
   }  
   int c = b;  
   }  
  }



* **Nu putem declara o variabila intr-un bloc care deja a fost declarata intr-un bloc parinte**
* public class Main {  
   public static void main(String[] args) {  
   System.*out*.println("Hello world!");  
   int a = 100;  
   {  
   int b = a+10;  
   int a = 100;  
   }  
   }  
  }



**Local Block**

* Un local block e ste un bock dintr-o metoda sau constructor sau alt bloc
* public class Main {  
   public static void main(String[] args) {  
   System.*out*.println("Hello world!");  
   {  
   System.*out*.println("A block");  
   }  
   }  
  }

**Instance Initialization Block**

* Ele se executa dupa static blocks.
* Se executa pentru fiecare obiect creat
* Dupa ele, se executa constructorul
* **aceste blocuri mereu sunt executate ca primele instructiuni din constructor, caci java se va asigura sa le execute deodata dupa ce constructorul a initializat itemii. Deci, ele vor fi puse automat inaintea la orice instructiuni ce noi le avem in constructor.**

**Static Initialization Block**

* Se declara asa:

static {

}

* Sunt executate doar o singura data, si anume **la crearea clasei in memorie**
* Ele mereu se executa primele.
* Nu pot accesa membri sau metode nestatice**, asa cum ele apartin clasei,** nu vreunui obiect!
* class A{  
   int a;  
   public A(int a){  
   this.a = a;  
   }  
   static {  
    
   a = 1000;  
   }  
  }

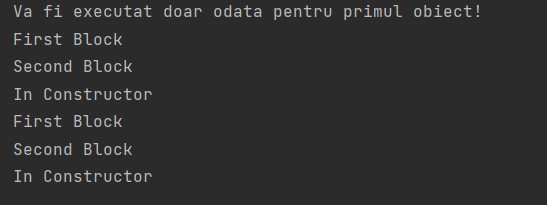


**Order**

1. Static blocks(doar la primul obiect)
2. Instance blocks
3. Constructor

public class Main {  
 public static void main(String[] args) {  
 Test test1 = new Test();  
 Test test2 = new Test();  
 }  
}  
class Test{  
 public Test(){  
 System.*out*.println("In Constructor");  
 }  
 {  
 System.*out*.println("First Block");  
 }  
 {  
 System.*out*.println("Second Block");  
 }  
 static{  
 System.*out*.println("Va fi executat doar odata pentru primul obiect!");  
 }  
  
}







**In practica, blocurile de clasa se folosesc in clasele anonyme, asa cum ele nu pot avea constructori. In rest, ele nu se prea folosesc.**

public class Main {  
 public static void main(String[] args) {  
 Thread thread = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 System.*out*.println("Hello");  
 }  
 static{  
 System.*out*.println("First block");  
 }  
 {  
 System.*out*.println("Second block");  
 }  
 });  
 }  
}

**Data more info**

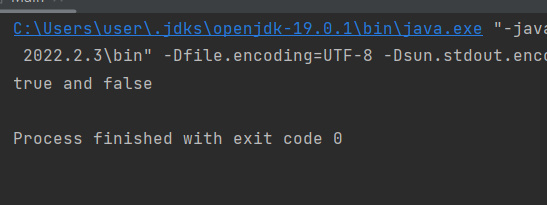
* Putem folosi \_ pentru a delimita cifrele mari:

int a = 1\_000\_000\_000;

* Nu putem da parse la un String ce contine \_
* int a = Integer.*parseInt*("1\_000\_000");



* orice text care contine true, ca “true” sau “TruE” e transformat in true, in rest se transforma false:
* Boolean a = Boolean.*parseBoolean*("TruE");  
  Boolean b = Boolean.*parseBoolean*("fefef");  
  System.*out*.println(a + " and "+b);



* Orice operatii cu date de tip short/byte mereu creaza un int!

short a = 5;

shoty b = 10;

a+b -> int nu short!!!

**Referinte**

* Upcasting si downcasting nu modifica obiectul din memorie, ci doar referinta pastrata spre el
* Chiar daca o variabila are null, oricum putem accesa membrii statici ai clasei cu ea:
* public class Main {  
   public static void main(String[] args) {  
   Test test = null;  
   test.*func*();  
   }  
  }  
  class Test{  
   public static void func(){  
   System.*out*.println("Static Method");  
   }  
  }



**Destroying objects**

* System.gc() si Runtime.getRuntime().gc() – vor **sugera** lui compilator ca poate sterge obiectele spre care nu mai sunt referinte. Totusi, ele nu garanteaza ca compilatorul va face asta 100%
* rareori configuram manual Garbage Collector
* Daca avem 2 obiecte ce au ceva referinte unul la altul, dar in code nu mai avem nicio referinta la ele,adica nu avem referinte in stack la ele, Garbage Collector va identifica aceasta situatie si tot va stere obiectele

**If**

* if/while/do while... accepta doar boolean!
* E posibil si asa ceva de facut:
* boolean a = false;  
  if(a = true)  
   System.*out*.println("Yes");

intai lui a i se asigneaza true, apoi se executa if cu valaorea lui a

**Switch**

* switch poate lua ca parametru doar date primite sau wrapper a lor si String.
* Test test = new Test();  
    
  switch (test){  
    
  }



**Break**

* break poate fi folosita si pentru a opri nu doar bucla in care se gaseste, ci si altele ce sunt parinti
* Pentru asta folosim label, adica putem pune in fata la fiecare bucla for/while etc. un Nume: si apoi dam break Nume;

MyLabel: for(int i = 0;i<10;i++)  
 MySecond: for(int x = 0;x<10;x++)  
 if(i == 5)  
 break MyLabel;  
 else  
 System.*out*.println(i);



* **Asa, break nu va opri bucla in care el se gaseste, ci cea a carui label cel scris apartine**

**String Pool**

* Stringurile ce nu sunt create cu new, dar au valoare identica, pointeaza catre acelasi obiect.
* De ex:
* public static void main(String[] args) {  
   String a = "Hello";  
   String b = "Hello";  
   String c = new String("Hello");  
    
   System.*out*.println(a==b); // true  
   System.*out*.println(a==c); // false  
  }

**== compara referintele la obiecte!**

* **Stringurile sunt imutabile!**
* **Metodele care modifica stringul vor returna un nou string, nu vor modifica cel ce a apelat metoda!**

String a = "Hello";  
a.replace('e','i');  
//a inca e "Hello"

**Constructors**

* Daca o clasa nu are nici un constructor, Compilatorul automat va pune un constructor default default, adica

numeClasa() { }

* **Daca clasa copil nu apeleaza nici un constructor din clasa parinte, compilatorul mereu va pune in constructorul la parinte super()**
* class A{  
     
  }  
  class B extends A{  
   public B(){  
   super();  
   }  
  }

