**Static**

Ca si in C++, Static inseamna ca ca o var sau un obiect se initializeaza o singura data si este sters doar la incheierea programului.

* Metodele nu pot avea variabile statice declarate in ele,nici macar daca metoda e statica!Spre deosebire de C++.
* O metoda statica doar face ca ea sa fie un membru de clasa. Ele nu memorizeaza starea variabilelor din ele.
* O metoda statica nu poate accesa membri nestatici
* Putem avea si blocuri statice unde putem accesa membrii statici. Aceste blocuri statice se definesc ca membri ai clasei si mereu se executa primele.
* This sau super nu poate fi utilizat in metode statice

**Scanner**

Orice tip de input am avea, fie **nextInt**(), **next**(), **nextDouble**() si oricare altul, inafara de nextLine(), ele vor citi caractere,odata ce au gasit pe cele potrivite, pana vor ajunge la primul spatiu liber sau la \n(Enter).

De exemplu daca am avea:

Scanner in = new Scanner(System.*in*);  
 int a,b;  
 a = in.nextInt();  
 b = in.nextInt();

Daca am scrie de la tastatura 10 5 sau 10 5,

atunci **a** ar fi egal cu **10** si **b** cu **5**. Ideea e ca functiile nextXXX citesc totul pana intalnesc un spatiu gol sau un \n(Enter). Daca sunt mai multe spatii goale, ele sunt ignorate de urmatorul nextXXX(). El va continua sa caute caractere trecand peste spatiile libere.

Insa, aceasta ar putea genera o problema cand se combina nextXXX cu nextLine();

De ex:

String a,b;  
int x;  
  
System.*out*.print("a = ");  
a = *in*.nextLine();  
 System.*out*.print("x = ");  
 x = *in*.nextInt();  
 System.*out*.print("b = ");  
 b = *in*.nextLine();

Cand vom introduce valaorea lui x si vom apasa Enter, se va trece peste b, adica se va afisa b = , insa nu vom putea seta vreo valoare, caci programul se va opri.

Acest lucru are loc din cauza ca, dupa ce introducem numarul lui x, apasam pe Enter, insa nextInt() va citi totul pana la acel enter, adica pana la \n, insa acest \n va ramane salvat in buffer de input, si deci cand nextLine() va trebui sa citeasca, ea va astepta sa se scrie ceva si sa se apese Enter ca sa apara un \n, dar un \n deja a ramas si deci ea crede ca noi am apasat deja Enter si nici nu ne mai lasa sa scriem ceva. nextLine() citeste caracterele cu tot cu Enter, adica cu \n, dar restul nextXXX NU!Ele citesc pana la primul spatiu sau \n si se opresc, indiferent ce urmeaza dupa, si deci nu stocheaza acest spatiu sau \n in ele, ci le lasa.

Pentru a rezolva problema, e sufficient ca dupa nextInt(); sa punem un in.nextLine() ca sa consume acest \n;

**Prelucrarea exceptiilor**

* Spre deosebire de C++, Java poate exact ca Python prinde cu un bloc mai multe exceptii ce se sepata prin |

Catch(typeexcpetion1 | typeexception2 | e)

Astfel, e va fi anume acel tip de eroare prins

* Cand scrie throw exceptie, putem sa-I trimitem si un text in constructor.

Throw IOException(“Un text default”)

* Daca nu exista un bloc ce sa printra acel throw, compilatorul il va executa.
* Putem utiliza si un throw fara sa existe un bloc try, pe care compilatorul il va executa, doar ca va opri programul la el.

int a = 10;  
if (a ==10)  
{  
 throw new IOException("RROR");  
}  
System.*out*.println("Aici nu se ajunge");

**Nested Classes**

**Sunt clase ce se gasesc in interiorul altor clase, exact ca orice membri.**

* Pot fi private,public,protected,default,static,final,**abstract**.
* Pot accesa membrii privati ai clasei de baza, ca orice metoda.
* **Inner class**
* **Nu sunt statice!**
* **Pot implementa interfete, extinde clase**
* Crearea unui obiect de tip clasa de baza nu creaza si un obiect de tip nested in el! Pentru a putea crea un obiect de tip inner class, trebuie sa avem un obiect de tip clasa de baza.
* Clasele inner (nestatice) nu pot avea membri statici, caci inner class depind de un obiect concret. Atentie! **In noile versiuni de Java deja putem avea membri statici in inner classes!!!**
* Putem instantia o inner class, doar ca diferit.

public class test {  
  
~~a.b obj = new a.b();~~  
}  
  
class a{  
 int nr = 100;  
  
 class b{  
 void func(){  
 ++nr;  
 }  
 }  
}

Putem sa o instantiem folosind un obiect de tip clasa de baza, dar nu asa

public class test {  
a obj1 = new a();  
~~obj1.b obj = new obj1.b();~~  
}  
  
class a{  
 int nr = 100;  
  
 class b{  
 void func(){

System.*out*.println(*nr*);  
 }  
 }  
}

ci asa:

public class test {  
a obj1 = new a();  
a.b obj2 = obj1.new b();

a.b obj3 = obj1.new b();



**obj2.nr;**

}  
  
class a{  
 int nr = 100;  
  
 class b{



int x = 100;  
 void func(){

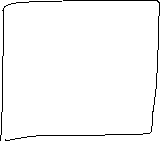
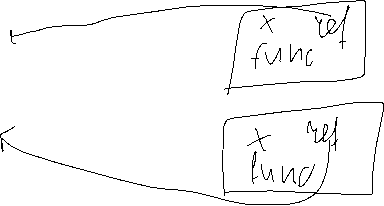
System.*out*.println(*nr*);  
 }  
 }  
}



Obiectele de tip inner class **nu fac** parte din namespace al obiectului de tip clasa de baza, dar au mereu o **referinta ascunsa** la el in namespace lor. Ele sunt amplasate undeva random prin memorie, si nu au nimic cu obiectul clasei de baza, doar referinta la el. Referinta ascunsa le permite sa accese membrii ai clasei de baza cand lucram chiar in clasa, dar nu si cand o facem in afara clasei.



**obj2.nr in main – ERROR REFERINTA E ASCUNSA**



* **Static Nested Classes:**
* Putem sa le instantiem direct folosind clasa
* Se comporta ca orice membri statici
* Pot avea membri statici
* public class test {  
    
  a.b obj = new a.b();  
  }  
    
  class a{  
   int nr = 100;  
    
   static class b{  
   void func(){  
   }  
   }  
  }
* Au un bloc in bloc al clasei de baza.

Se va crea mereu un bloc si object al acestei nested class, dar acest bloc **va face** parte din bloc al clasei de baza, care va crea un na bloc la runtime, , caci nested clasa asta doar e tot membru static. Totusi, nu putem accesa direct membrii din nested class din clasa de baza, caci nested class e un membru, nu un sub bloc ce vine de la o clasa parinte, de asta mereu va trebui sa folosim **NumeNestedClass.metoda** pentru a accesa din namespace al clasei de baza.

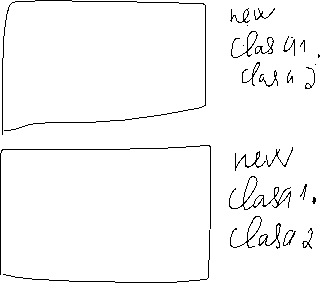
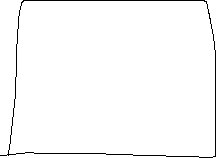
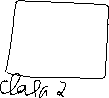
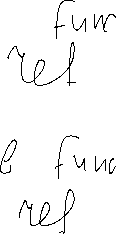
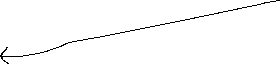
Orice instantiere a acestei nested class va duce pur si simplu la crearea unui nou obiect undeva prin memorie, ce va avea in el membri nestatici si ce va avea referinta la object class al acestei nested class. (ca crearea unui obiect simplu a carui clasa are membri statici)

class clasa1{  
int a = 100;



static int x = **500**;  
static class clasa2{  
 int b = 200;  
 static int *c* = 300;  
 public void func(){  
 System.*out*.println(*c*);  
 } }

}



clasa1 obj = new clasa1();  
  
clasa1.clasa2 obj2 = ~~obj.new clasa2();~~

**Nu putem crea in asa mod un obiect de tip nested class printr-un obiect de tip clasa de baza, caci obiectul ar trebui sa aiba 2 referinte, una la obiectul de tip clasa de baza, si alta la object al nested class, si asta ar fi o sursa de confuzie, caci nu e clar care referinta sa se foloseasca prima.**

clasa1.clasa2 obj = new clasa1.clasa2();  
~~obj.a;~~

**Accesarea unui membru static din clasa de baza printr-un obiect de tip nested class nu e posibila, caci membrul va fi cautat in** bloc **al nested class, si acolo nu va fi gasit, iar bloc al nested class are din nou o referinta ascunsa la bloc al clasei de baza,ce ne permite sa accesam membrii statici ai clasei de baza doar cat lucram din clasa.**

**Mai multe Despre Inner Class**

**Inner Class** pot fi de 2 tipuri:

* Local Inner Class
* Anonymus Class

**Local Inner Class**

* **Local Inner Classes** sunt clasele definitre intr-un bloc,nefiind membri ai unei clase, deobicei fiind vorba de metode, bucle for,if, while etc.
* Au acces la membrii din bloc, si daca sunt in bucle de ex, si la cei ai metode din care fac parte si ai clasei, ca si metoda.
* Ca si inner class simple, pot extinde clase si implementa interfate.
* Daca avem un local inner class intr-o metoda, la compilare, se creaza 2 clase: clasa de baza (baza.class) si clasa local inner (baza$locala.class de ex), si local inner class va avea o referinta catre clasa de baza,de aceea si poate accesa membrii ei, asa ca orice bucla sau metoda in principiu. Adica, la crearea unui obiect de tip clasa de baza, si la instantierea clasei de tip inner local class in metoda, instanta va contine o referita catre insusi obiectul de baza. Totusi, metoda nu are namespace, are scope sau bloc, si clasa local inner se gaseste in el, de aceea cautarea membrilor se face in ordine inversa, de la bucla la metoda si daca variabilele apelate de obiectul de tip local inner class nu sut gasite nici in scope principal la metoda, se folsoeste referinta la obiect de baza pentru a le cauta. Scope nu functioneaza la fel ca namespace.
* Nu pot fi statice! Metodele nu accepta membrii statici.

**Anonymus Class**

* Este o inner class fara nume, pentru care doar un singur obiect e creat.
* Are forma:

Test t = new Test() { ... }

Test poate fi clasa, clasa abstracta si interfata.

Intre {...} se suprascriu metodele ce trebuie suprascrise, daca e clasa abstracta sau interfata, sau pot fi suprascrise metodele unei simple clase. Aceasta clasa noua pe care o definim intre { } nu are nume, si exista doar pentru obiectul t.

* De aici reiese ca annonymus class poate implementa doar o inerfata, adica Test, caci nu putem crea un obiect de mai multe tipuri, sau poate extinde o clasa doar, nu si clasa si interfete.
* Nu poate avea constructor, asa cum nu are nume.
* Pot avea membri statici si metode ce nu le are clase de baza, ca orice clasa copil.
* Dupa declararea lor, se pune ; dupa }

interface Coach{  
 public String getDailyWorkout();  
}

Coach test = new Coach(){  
 @Override  
 public String getDailyWorkout(){  
 System.*out*.println("Hello");  
 return null;  
 }  
 static int *a* = 100;  
};

**Anonymus class sunt de 3 tipuri:**

* Extind o clasa, ca de exemplu:

Thread thread = new Thread() {

@Override

Public void run(){ System.out.println(“Hello”); };

* Extind o interfata

    Runnable r = **new** Runnable() {

            // run() method for the thread

**public** **void** run()

            {

                // Print statement when run() is invoked

                System.out.println("Child Thread");

            }

        };

* Clase anonymus din metode sau constructor

Thread t = **new** Thread(**new** Runnable() {

**public** **void** run()

            {

                System.out.println("Child Thread");

            }

        });

Aici clasa anonymus e in contsructorul lui Thread!

**Enumeration**

* Rolul la o enumeratie este de a stoca o multime de constante
* Enum in java este la fel ca o clasa. Putem deci sa avem un contructor, metode, instante etc. si chiar implementa interfete
* Orice enum tot foloseste keyword class
* Fiecare constanta din enum reprezinta un tip al acelui enum

public enum Status {  
 *ACTIVE*, *INACTIVE*}

Deci, ACTIVE si INACTIVE sunt de tip Status

* Enum elements pot fi puse in switch
* Orice constanta din enum e mereu public static final
* Orice enum entinde automat clasa java.lang.Enum class, deci e logic ca nu mai poate extinde alte clase
* metoda toString() e suprascrisa deja in Enum class, si returneaza numele la constanta
* Un enum poate avea un constructor , si acest constructor va fi executat pentru fiecare constanta din enum
* Nu putem crea obiecte de tip enum, caci constructor e private
* Putem crea metode si sa le apelam prin constante:
* public enum Status {  
   *ACTIVE*, *INACTIVE*;  
    
   public void sayHello(){  
   System.*out*.println("Hello");  
   }  
  }  
  class Main{  
   public static void main(String[] args) {  
   Status status = Status.*ACTIVE*;  
   status.sayHello();  
   }  
  }
* Metode:

- enum.values() – returneaza toate constantele enumeratiei

- enum.valueOf(“Nume”) – returneaza constanta care se numeste asa. Daca ea nu exista, o exceptie e aruncata

- constanta.ordinal() – returneaza index la constanta

* Putem liber folosi == pentru a compara constantele, asa cum oricum se folsoeste aceeasi referinta la constante

**Arrays**

* Se creaza prin:

Tip[] var = new tip[nr]

* Daca este creat un array ce nu are valori in el, Java automat va seta niste valori astfel:
* int – 0
* double – 0.0
* String – null
* boolean – false
* user defined type – null
* Putem crea intai o variabila ce are o referinta la un tip array si abea dupa sa o instantiem.

type[] tablou;

tablou = new type[nr]

* Putem direct sa initializam un array fara sa il mai instantiem

Type[] tablou = {1,2,3,4,5,6} say Type tablou = new Type[] {1,2,3,4,5,6}

Acest new Type[] este suplimentar

Dar nu se permite specificarea numarului de elemente si apoi declararea lor automata:

~~int a[] = new int[5]{1,2,3};~~

* **Tablou.clone() –** returneaza un tablou copie, nu o referinta, daca e vorba de un tablou unidimensional.Daca e vorba de un tablou multidimensional, atunci se va returna referinta catre toate tablourile din el.

**public** **class** HelloWorld{

**public** **static** **void** main(String[] args) {

**int**[] tablou1 = {1,2,3,4,5};

**int**[] tablou2 = tablou1.clone();

System.***out***.println(tablou1 == tablou2);

}

}

**public** **class** HelloWorld{

**public** **static** **void** main(String[] args) {

**int**[][] tablou1 = {{1,2,3},{4,5,6}};

**int**[][] tablou2 = tablou1.clone();

System.***out***.println(tablou1 == tablou2);

System.***out***.println(tablou1[0]==tablou2[0]);

tablou1[0][0] = 10;

tablou1[0][1] = 20;

tablou1[0][2] = 30;

**for**(**int** i : tablou2[0]) {

System.***out***.println(i);

}

}

}

* Un array bidimensional poate avea in el tablouri de diferite dimensiuni. Acest lucru se face prin:

**int**[][] bi = **new** **int**[][] {

**new** **int**[] {1,2,3},

**new** **int**[] {10,20,30,40},

**new** **int**[] {100,200,300,400,500} };

sau

**int**[][] bi = {

**new** **int**[] {1,2,3},

**new** **int**[] {10,20,30,40},

**new** **int**[] {100,200,300,400,500} };

sau

**int**[][] bi = {

{1,2,3},

{10,20,30,40},

{100,200,300,400,500}

};

Putem de asemenea doar sa specificam lungimea si atat. Putem sa alocam memorie fiecarui tablou si dupa declarare, doar ca va trebui neaparat sa specificam cate tabele vor fi in tabloul bidimensional si abea dupa putem specifica lungimea fiecaruia!

* Pentru a crea un tabel cu tabele de diferite dimensiuni, putem sa oferim o valoare doar la numarul de tabele;

int a[][] = new int[5][];

dar va trebui apoi sa specificam cate elemente are fiecare array din tabel!

**ArraysClass**

* Toate metodele din aceasta clasa sunt statice si ne pot ajuta sa facem anumite lucruri ce ar fi greu de facut cu tablouri simple.
* Nu putem crea propriu zis obiecte de tip ArraysClass, dar putem sa utilizam metodele din ea.
* **Arrays.asList(array)** – returneaza un obiect de tip List bazat pe valorile lui array. E important sa trimitem Arrays de tip obiect, nu date primitive, caci lista contine un sir de obiecte, si un tablou primitiv are date primitive in el, el fiind doar un singur obiect de date primitive. De altfel, va trebui sa definim lista ca de tip int[] de ex, daca e int, si ea va avea doar un element care va si si tabloul de tip int

Integer[] tablou = {1,2,3,4,5};

List <Integer> lista = Arrays.*asList*(tablou);

System.***out***.println(lista);

**int**[] tablou = {1,2,3,4,5};

List <**int** []> lista = Arrays.*asList*(tablou);

System.***out***.println(lista); // returneaza locatia in memorie a tabloulu

* **Arrays.binarySearch(array, element)** – returneaza pozitia elementului gasit. Daca nu il gaseste, va returna cu semnul minus pozitia unde ar fi trebuit sa fi fost elemental daca s-ar fi gasit in array, dar pozitia va incepe de la -1, nu de la 0. Trebuie de folosit metoda **Arrays.sort(array)** inainte de a o utiliza, de altfel rezultatul va fi unul gresit.
* **Arrays.binarySearch(array, poz1,poz2,element) –** cauta elememtul de la poz1 pana la poz2
* **Arrays.compare(array1,array2) –** nu compara lungimile lor, ci compara intai primele lor elemente(pozitia 0), si daca sunt egale, compara urmatoarele(pozitia 1) si tot asap ana gaseste prima pereche de elemente ce nu sunt egale. Daca array1[x] > array2[x], returneaza 1, in caz contrar -1
* **Arrays.compareUnsigned(array1, array2)** – compara tablouri de date primitive fara a tine cont ca sunt pozitive sau negative.
* **Arrays.copyOf(array, nr)** – returneaza un nou Array ce contine un numar de **nr** elemente, primele fiind cele din array de copiat. Daca array de copiat are mai putine elemente decat nr, noul array returnat va adauga 0 unde nu a avut ce copia(pentru tabluri de tip date primitive) sau null(pentru tablouri de tip obiecte)

**int**[] a = {1,2,3,4,5};

System.***out***.println(Arrays.*toString*(Arrays.*copyOf*(a,8)));

* **Arrays.copyOfRange(array,poz1,poz2) –** copie elementele lui array de la poz1 pana la poz2, fara poz2. Daca poz2 e mai mare ca nr de elemente a lui array, ele iar vor fi inlocuite cu 0 sau null
* **Arrays.toString(array)** – returneaza o reprezentare cu toate elementele din array.
* **Arrays.deepToString(array)** – returneaza o reprezentare la toate elementele din array, dar daca e multidimensional, va arata oricum fiecare element. toString, in caz de un array multidimensional, ar arata doar locatiile fiecarui tablou, dar aceasta metoda arata si elementele fiecarui tablou di tabloul de tablouri.
* **Arrays**.**equals(array1,array2)** – compara fiecare element din arrayuri, si daca gaseste 2 care nu sunt egale, returneaza false, in caz contrar true
* **Arrays.fill(array,valoare)** – umple array de elemente valoare, adica va contine in el doar un elemente pe toate pozitiile.
* **Arrays**.**hasCode(array)** – returneaza un HashCode
* **Arrays**.**mismatch(array1,array2)** – returneaza prima pozitie in care array1 si array2 au elemente ce difera
* **Arrays**.**sort(array,**comparator**(optional)) –** sorteaza array
* **Arrays**.**sort(array,poz1,poz2)** – sorteaza array de la poz1 pana la poz2, fara poz2, restul elementelor pastrandu-si pozitia
* **Arrays**.**sort(array,poz1,poz2,comparator)** – sorteaza array de la poz1 pana la poz2 pe baza la comparator, fara poz2, restul elementelor pastrandu-si pozitia
* **Arrays.stream(array) –** returneaza un stream cu toate elementele lui arry.

**Stream**

* **Este o secventa de obiecte ce pot suporta anumite operatii**
* **Nu modifica obiectul original, ci o copie a lui**

**Functional Interfaces**

* **Functional Interface** – inetrafata care contine doar o singura metoda abstracta
* Poate avea orice metode **default si static**
* Putem folosi anotatia @**FunctionalInterface**, in fata unei interfate, ca se asiguram ca nu va avea mai mult de o metoda abstracta. Folosirea ei e optionala.
* @FunctionalInterface  
  interface interfata{  
   void func();  
  }
* Daca declaram mai mult de o metodata sau nici una(abstracta), vom primi eroare.

**Lamba Expressions**

* **Lamba Expressions** - Sunt instante ale interfetelor functionale(care au doar o metoda abstracta)
* **Orice lambda expression este un obiect, si mai exact o instanta interfatei functionale. Deci, lambda expression creaza obiectul.**
* **Lamba Expressions sunt doar o imbunatatire la anonymus class, caci daca de ex interfata are doar o metoda abstracta, anonymus class poate parea neclar.**
* Functia implementata de lamba expression nu apartine niciunei clase.
* O lambda expresie poate fi trimisa ca orice obiect.
* Lamba expresia e ca o clasa ce implementeaza interfata, si funrizeaza un corp concret metodei abstracte.

@FunctionalInterface  
interface Interfata{  
 void func();  
}  
  
public class test {  
 public static void main(String[] args) {  
 Interfata obj = () -> System.*out*.println("Hello");  
 obj.func();  
 }  
}

* **Putem accesa si metodele default.**
* **Putem sa nu mai scriem tipul parametrilor, asa cum el si asa e clar si evident**

@FunctionalInterface  
interface Interfata{  
 int func(int a, int b);  
  
 default void show(){  
 System.*out*.println("Hello");  
 }  
}  
  
public class test {  
 public static void main(String[] args) {  
 Interfata obj = (x,y) -> {  
 int c = x + y;  
 return c;  
 };  
 System.*out*.println(obj.func(10,20));  
 obj.show();  
 }  
}

* Utilizarea {} este necesara cand avem mai mult de o isntructiune in metoda
* Daca metoda returneaza ceva, si nu utilizam {}, atunci expresia scrisa va fi si cea returnata.
* @FunctionalInterface  
  interface Interfata{  
   int func(int a, int b);  
  }  
    
  public class test {  
   public static void main(String[] args) {  
   Interfata obj = (x,y) -> x+y;  
   System.*out*.println(obj.func(10,20));  
   }  
  }
* Daca avem nevoie de mai multe instructiuni in metoda, folosim {} si return la final
* **Putem folosi lamba la un ArrayList pentru a afisa elementele sale:**

public class test {  
 public static void main(String[] args) {  
 ArrayList<Integer> array = new ArrayList<>();  
  
 array.add(10);  
 array.add(24);  
 array.add(5);  
 array.add(28);  
  
 array.forEach((n)->{  
 if(n%2==0)  
 System.*out*.println(n+" este par");  
 });  
 }  
}

Aici trimitem un obiect de tip lambda expression, si el va selecta fiecare element si il va trimite acestei lambda expression.

* Putem crea metode ce iau ca parametru un obiect de tip interfata, dar care va fi defapt un obiect de tip lambda expression, a carui metoda asbtracta e suprascrisa si poate fi utilizata.

Lambda Expressions au grija sa creeze un nou obiect. Ele fac exact acelasi lucru ca si anonymus class, evident daca are o metoda abstracta, doar ca ne simplifica codul, fara a mai fi necesar new Type {...} . Ele pot fi trimise si ca parametru foarte usor:

public class Test {  
 public static void main(String[] args){  
 Person person1 = new Person();  
 *printResult*(person1, (person) -> person.getAge()>80);  
 }  
 public static void printResult(Person person,Filter1 filter){  
 if(filter.isOld(person))  
 System.*out*.println("This person is old");  
 else  
 System.*out*.println("This person is young");  
 }  
}  
interface Filter1{  
 boolean isOld(Person person);  
}  
class Person{  
 private int age;  
  
 public int getAge() {  
 return age;  
 }  
  
 public void setAge(int age) {  
 this.age = age;  
 }  
}



Deci, unde am scris **(person)->person.getAge()>80** s-a creat automat un nou obiect, ce are un anonymus class cu metoda isOld() suprascrisa. Java doar stie ca metoda printResult cere ca al 2 parametru un obiect de Filter1, si deci stie ca aceasta este o interfata functionala si va sti deci ca trebuie sa creeze un obiect ce va avea suprascrisa metoda din interfata si va mai avea si alte metode default in el.

* Lambda expression pot accesa variabilele din metoda in care se afla sau din clasa unde sunt declarate fara probleme.
* public class Test {  
   public static void main(String[] args){  
   Person person1 = new Person();  
   int x = 80;  
   *printResult*(person1,person -> person.getAge()>x);  
   }  
   public static void printResult(Person person,Filter1 filter){  
   if(filter.isOld(person))  
   System.*out*.println("This person is old");  
   else  
   System.*out*.println("This person is young");  
   }  
  }



**Lamba Expression nu creaza un nou scope. Dar asta nu inseamna ca nu putem declara variabile chiar in lambda expression**

* **In lambda expression nu putem modifica valorile variabilelor in afara ei:**
* public class Test {  
   public static void main(String[] args){  
   Person person1 = new Person();  
   int x = 80;  
   *printResult*(person1,person -> {x=50; return person.getAge()>x;});  
   }  
   public static void printResult(Person person,Filter1 filter){  
   if(filter.isOld(person))  
   System.*out*.println("This person is old");  
   else  
   System.*out*.println("This person is young");  
   }  
  }



[Lambda Expressions (The Java™ Tutorials > Learning the Java Language > Classes and Objects) (oracle.com)](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html)

**Method References**

* In Java, doar expresiile Lambda pot fi folosite pentru a trimite metode prin referinta. Altfel nu e posibil
* Sintaxa generala:

Object::methodName

* Method reference e folosit pur si simplu pentru a crea o lambda expression pe baza unei metode deja existente. De ex:

public class Main {  
 public static void main(String[] args) {  
 Test test = () -> System.*out*.println("Hello!");  
 }  
}  
  
@FunctionalInterface  
interface Test{  
 void hello();  
}

Aici am creat o lambda expression, dar daca de exemplu metoda asta pe care noi am folosit-o in lambda expression deja e creata undeva, nu ar avea sens sa o cream iar in lambda expression, de ex:

public class Main {  
 public static void main(String[] args) {  
 Test test = Message::*sayHello*;  
 }  
}  
  
@FunctionalInterface  
interface Test{  
 void hello();  
}  
  
class Message{  
 public static void sayHello(){  
 System.*out*.println("Hello!");  
 }  
}

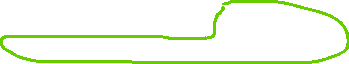


Aici noi pur si simplu cream o lambda expression pe baza la o metoda ce deja exista. Am trimis drept referinta o metoda deja existenta, si atat.Metoda abstracta din interface va fi suprascrisa si va avea referinta la o metoda deja existenta.

* Odata trimisa aceasta referinta, obiectul nou creat va folosi o metoda deja existenta, deja creata, folosind referinta le ea.
* Anonymus class, lambda expression, method reference, toate cam fac acelasi lucru, doar ca method reference e mult mai usor de folosit in unele cazuri.
* Exista 4 tipuri de metod references:

1. **Static method reference** - adica se ia o metoda statica de la o clasa.

public class Main {  
 public static void main(String[] args) {  
 Test test = (s) -> Message.*sayHello*(s);  
 }  
}  
@FunctionalInterface  
interface Test{  
 void hello(String s);  
}  
  
class Message{  
 public static void sayHello(String s){  
 System.*out*.println(s);  
 }  
}



Dar, putem simplifica asta:



Test test =Message::*sayHello*;

**Atentie! Metoda statica data prin referinta trebuie sa corespunda ca prototip cu cea abstracta din interfata!**

1. **Method from a particular object –** adica pur si simplu trimitem metoda unui obiect creat, si atat

public class Main {  
 public static void main(String[] args) {  
 Message m = new Message();  
 Test test = s -> m.sayHello(s);  
 }  
}  
@FunctionalInterface  
interface Test{ void hello(String s);}  
  
class Message{  
 public void sayHello(String s){System.*out*.println(s);}  
}

Dar asta poate fi prescurtat:

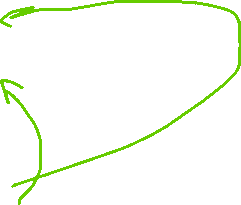
Test test = m::sayHello;

1. **Constructor method reference** –

Are sintaxa: **Clasa::new**

* Chiar daca o clasa are mai multi constructori, se va identifica automat cel potrivit, daca exista
* Constructorul tot este o metoda, si deci putem sa il trimitem, ca si cum el ar fi o metoda ce returneaza referinta la obiectul creat

public class Main {  
 public static void main(String[] args) {  
 Factory1 factory1 = () -> new Employee();  
 Factory2 factory2 = (s) -> new Employee(s);  
 }  
}  
  
interface Factory1{  
 Employee createEmptyEmployee();  
}  
interface Factory2{  
 Employee createNamedEmployee(String s);  
  
}  
class Employee{  
 private String firstName;  
 public Employee(){  
 System.*out*.println("Default constructor");  
 }  
 public Employee(String s){  
 this.firstName = s;  
 }  
}



asta poate fi inlocuit cu:

Factory1 factory1 = Employee::new;  
Factory2 factory2 = Employee::new;

1. **Reference to an Instance Method of an Arbitrary Object of a Particular Type(referinta la o metoda nestatica folosind clasa)**

Putem sa trimitem si o referinta a unei metode dintr-o clasa, care nu este statica. Dar, aici apare o problema. Problema ar fi ca daca de ex, avem o clasa:

class Shop{  
 private String name;  
  
 public Shop(String name) {  
 this.name = name;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 @Override  
 public String toString() {  
 return "Shop{" +  
 "name='" + name + '\'' +  
 '}';  
 }  
}

Si vom incerca sa cream o interfata functionala asa:

@FunctionalInterface  
interface Functional{  
 String func();  
}

O astfel de declaratie se va solda cu o eroare:

Functional f = Shop::getName;

Asta e din simplul motiv ca la apelarea acestei metode, nu e clar asupra carui obiect ea se apeleaza. Asta nu e o metoda statica, deci are neaparat nevoie sa aiba un obiect asupra careia sa fie executata. Java a hotarat sa adauge o solutie pentru a putea oferi referinta la metode non statice ale oricarei clase, si solutia e descrisa mai jos

* Daca dorim sa oferim o referinta la o metoda nestatica dintr-o clasa, este necesar sa mai adaugam un argument suplimentar pe prima pozitie, care sa fie obiectul asupra caruia se executa metoda. Chiar daca metodele nu vor corespunde ca prototip, Java se va asigura sa rezolve problema. Ea va folosi primul argument, care este obiectul aspra caruia se apeleaza metoda, pentru a putea apela metoda.

De ex:

public class Reference {  
 public static void main(String[] args) {  
 Shop shop = new Shop("Metro");  
 Functional<Shop> f = Shop::getName;  
 System.*out*.println(f.func(shop));  
 }  
}  
@FunctionalInterface  
interface Functional<T>{  
 String func(T t);  
}

getName() nu are nici-un parametru, dar func() vedem ca are

Totusi, acest parametru va fi folosit de Java pentru a apela metoda asupra obiectului trimis ca parametru, si asa nu va aparea o eroare, dar numai in asa situatie prototipul la metode poate diferentia. Va fi necesar sa trimitem obiectul asupra caruia vrem sa se execute metoda.

* Daca oferim o referinta la o metoda nestatica care are parametrii, pur si simplu obiectul asupra caruia se va apela metoda va fi mereu primul argument:
* public class Reference {  
   public static void main(String[] args) {  
   Shop shop = new Shop("Metro");  
   Functional<Shop> f = Shop::setName;  
   f.func(shop,"Jora");  
   }  
  }  
  @FunctionalInterface  
  interface Functional<T>{  
   void func(T t,String name);  
  }

**Acest prim argument care e obiectul va fi folosit pentru this.**

**Wrapper Class**

Sunt clase care stocheaza date primitive. Sunt mai bune ca datele primitive, din cauza ca putem sa le modificam cu metode si le putem trimite ca referinte la metode, caci in java datele primitive sunt trimise ca copii doar.

* **Boxing** – transformarea datelor primitive in obiecte

Integer a = Integer.valueOf(100);

* **Unboxing** - transfomarea obiecteor de tip wrapper in date primitive.

Integer a = 20;

int b = a.intValue();

* **Autoboxing** – transformarea automata a datelor primitive in obiecte de tip wrapper

Integer a = 20;

* **Autounboxing** – transformarea automata a obiectelor de tip wrapper in date primitive.

Integer a = 20;

int b = a;

* **Metode**:
* **valueOf(String/data primitive)** –Metoda statica. Fiecare wrapper class, cu exceptia lui Character, au o asa metoda, care e menita sa transforme mai mult stringul in wrapper obj, de ex valueOf(“20”); Returneaza un obiect de tip wrapper. Daca un obiect de tip Integer, sau clasa Integer o apeleaza, se va transforma string/data primitiva in Integer, daca Double o cheama, in Double si tot asa.
* **valueOf(String/data primitive, int baza) –** trasnforma textul in numar in baza data, ce poate fi de la 2 la 32, dar automat e baza 10, sistemu zecimal.
* **byte/short/int/float/double/longValue() –** **returneaza valaorea de tip primitive** transformata in tipul ales.

Float a = 5.6f;

Int b = a.intValue();

* **parseInt/Float/..(String)** – Metoda statica. Converteste un string in data primitivea, nu wrapper obj, spre deosebire de valueOf(); si nu poate lua in parametru date primitive. La fel ca valueOf(), converteste stringul in acel tip de data primitive in dependenta de clasa apelata. Adica, Integer poate apela parseInt, nu si parseDouble, sau Double poate apela doar parseDouble() si tot asa.
* **parseInt/Float/..(String, baza)**
* **toString() –** orice wrap obj are o asa metoda care converteste valoarea pastrata in string.
* **toString(valoare) –** este o metoda statica ce trebuie apelata de clasa,nu de vreun obiect.Fiecare wrapper clasa are o asa metoda, ce ia ca parametru o data de tipul characteristic ei.

**De ce downcasting nu e automat?**

Upcasting e automat, deoarece cand avem un obiect de tip parinte egal cu un obiect de tip copil, gen:

Parinte a = new Copil();

compilatorul e sigur ca nu are cum fi vreo eroare, dar cand avem un obiect de tip copil egal cu unul de tip parinte, aici pot aparea erori daca obiectul de tip parinte nu are referinta la unul de tip copil. Gen:

Parinte p = new Copil();

Copil c = p;

Obiectul p posibil sa nu aiba ref la unul de tip copil si sunt sanse sa fie o eroare, de aceea Java obliga sa specificam downcasting pentru a arata ca stim ca acesta la sigur are ref la obiect de tip copil.