Curs 1 PPOO

Conf. univ. dr. Cristian CIUREA

Departamentul de Informatică și Cibernetică Economică

cristian.ciurea@ie.ase.ro

Structură evaluare

Curs:

- 60% (examen la calculator)

Seminar:

- 20% (proiect, una din 5 teme alocată din oficiu)
- 20% (test grilă)

Să ne cunoaștem...

Academia de Studii Economice Din București	**63**
Facultatea de Cibernetică, Statistică și Informatică Economică	58
Facultatea de Comerț	1
Facultatea de Contabilitate și Informatică de Gestiune	3
Facultatea de Finanțe, Asigurări, Bănci și Burse de Valori	1
Universitatea "Alexandru Ioan cuza" Din Iași	**1**
Facultatea de Informatică	1
Universitatea Din București	**1**
Facultatea de Matematică și Informatică	1
Universitatea Politehnica Din București	**3**
Facultatea de Automatică și Calculatoare	2
Facultatea de Electronică, Telecomunicații și Tehnologia Informației	1
(blank)	**3**
(blank)	3

Total = 13 studenti non-CSIE

Să ne cunoaștem…

- http://www.cristianciurea.ase.ro/JavaTest.aspx

Ce obiective are disciplina?

- programare orientată obiect în Java;
- proiectare aplicații eficiente folosind principii de internaționalizare (I18N), documentare cod sursă (JavaDoc), design patterns (Singleton), etc.

Ce vom învăța?

- Tipuri de date, variabile
- Masive
- Structuri de control
- Transferul parametrilor
- Clase, polimorfism, clase abstracte, interfete
- String si immutable
- Exceptii try-catch
- JavaDoc
- Internationalizare
- Colectii
- Genericitate
- Fisiere, serializare
- Singleton, Callback
- Fire de executie, procese
- Lucrul cu baze de date
- Spring, Swing, Java FX

Java fundamentals

- JDK, JRE, JVM
- Concepte de bază ale JVM
- Utilizarea unui IDE – NetBeans/Eclipse/IntelliJ
- Structura unei aplicații Java
- Pachete
- Pași de dezvoltare a unei aplicații Java
- Compilare în linie de comandă
- Tipuri de date în Java
- Variabile în Java
- Boxing / Unboxing
- Masive în Java
- Stiva și Heap

JDK, JRE, JVM

- **JDK - Java Development Kit**: formează subsetul SDK, care are responsabilitatea pentru scrierea și rularea programelor Java.

- **JRE - Java Runtime Environment**: conține API-uri adecvate, împachetate împreună cu JVM.

- **JVM - Java Virtual Machine**: este o mașină virtuală, care poate executa Java bytecode. Este componenta pentru execuția codului sursă a platformei software Java.

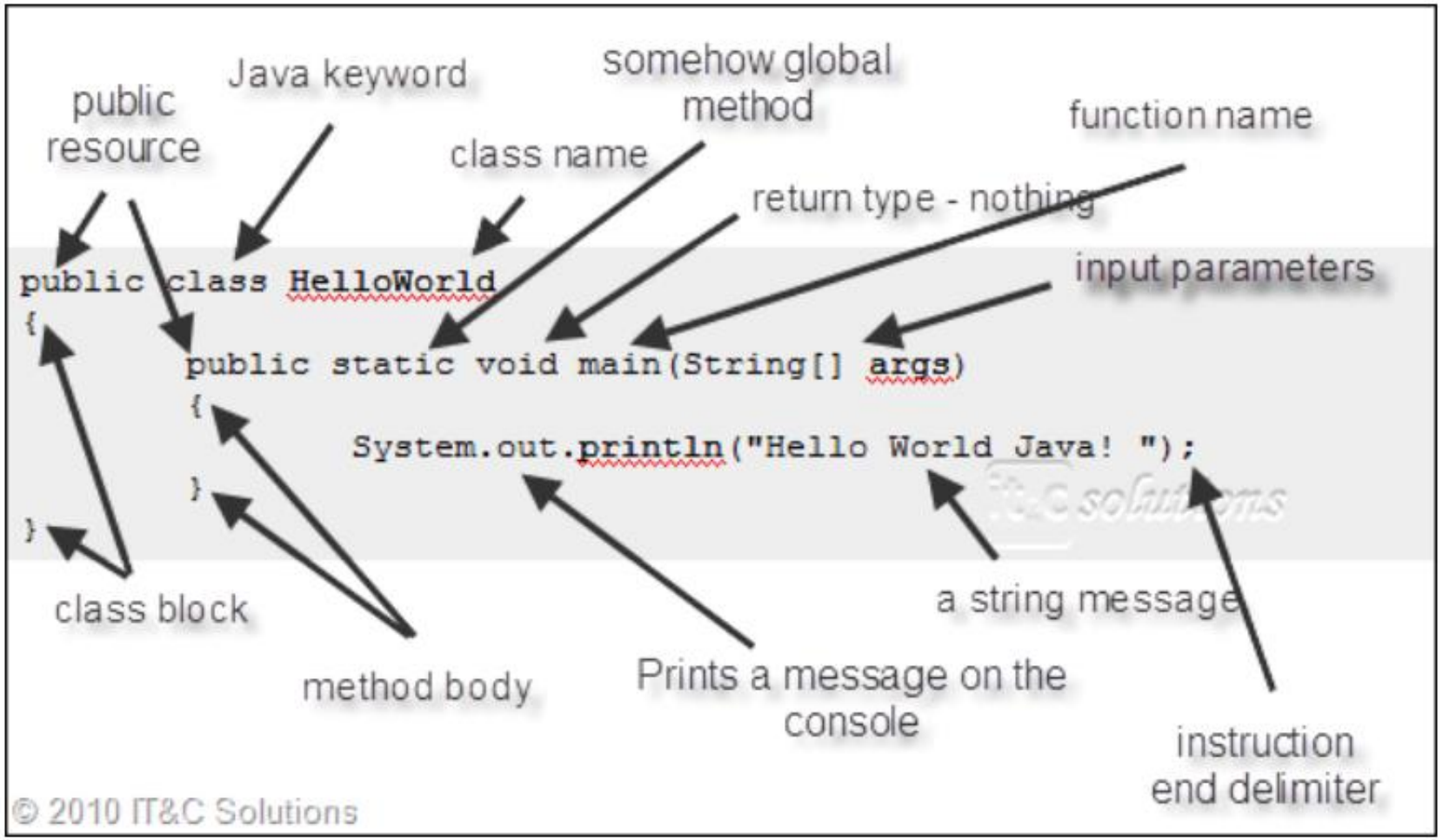
Noutăți Java 8... Java 13

- **Java 8:** expresii lambda pentru accesare elemente colectii
- **Java 9:** modularizare JDK
- **Java 10:** release March 20, 2018
  - Experimental Java-based JIT compiler
- **Java 11:** release September 25, 2018
  - Epsilon: a no-op garbage collector
  - HTTP Client has been standarized (java.net.http)
  - support for Unicode 10
- **Java 12:** release March 19, 2019
  - Switch Expressions, and many others
- **Java 13:** release September 17, 2019
  - hundreds of smaller enhancements and thousands of bug fixes

Utilizarea unui IDE

- **NetBeans:** https://netbeans.org/
- **Eclipse:** https://www.eclipse.org/
- **IntelliJ IDEA:** https://www.jetbrains.com/idea/

Structura unei aplicații Java



Structura unei aplicații Java

- comentariile pe o linie sunt definite prin **//**
- comentariile pe mai multe linii incluse între **/*** și ***/**
- delimitatorul pentru instrucțiuni este **;**
- instrucțiunile pot fi include în blocuri de cod definite prin **{ }**
- limbajul Java este case sensitive, variabila **vb** este diferită de **Vb** și **VB**

Structura unei aplicații Java

- Totul se definește în interiorul unei clase;
- Nu se pot defini variabile globale sau metode în afara clasei (ca în C/C++);
- Conținutul unei clase este încadrat între { și };
- Clasa care include funcția *main()* are aceeași denumire cu fișierul de cod sursă care o conține.

Pachete

- Un pachet este o grupare de clase înrudite, interfețe, enumerări și tipuri de adnotări care oferă protecție la acces și gestiune a spațiului de nume.
- Pentru a crea un pachet, se alege un nume pentru acel pachet și se pune o declarație de pachet cu acest nume în partea de sus a fiecărui fișier sursă care îl conține.

```
//in the Graphic.java file

package graphics;

public abstract class Graphic { ... }
```

Pachete

- Numele pachetelor sunt scrise cu toate literele mici, pentru a evita un conflict cu numele de clase sau interfețe.
- Companiile folosesc numele inversat aferent domeniului de Internet pentru a începe denumirile de pachete: **com.example.mypackage** pentru un pachet denumit **mypackage** creat de un programator din compania **example.com**.
- Pachetele aferente limbajului Java încep cu **java.** sau **javax.**

Pachete

- Pentru a importa un anumit membru în fișierul curent, se pune o declarație de import la începutul fișierului.
- `import graphics.Rectangle;`

- Pentru a importa toate tipurile conținute într-un pachet, se utilizează declarația de import cu asterisc (*).
- `import graphics.*;`

Pachete

Package class modifiers are:

- **default** (when you don't use anything) - - the class is visible in the package;
- **public** - the class is visible anywhere

different packages

```
package p1;

public class Class1
{ }

class Class2
{ }
```

```
package p2;

import p1.*;

public class Other
{
    Class1 c1;

    Class2 c2;
}
```

only the public class is visible

same package

```
package p1;

public class Class1
{ }

class Class2
{ }
```

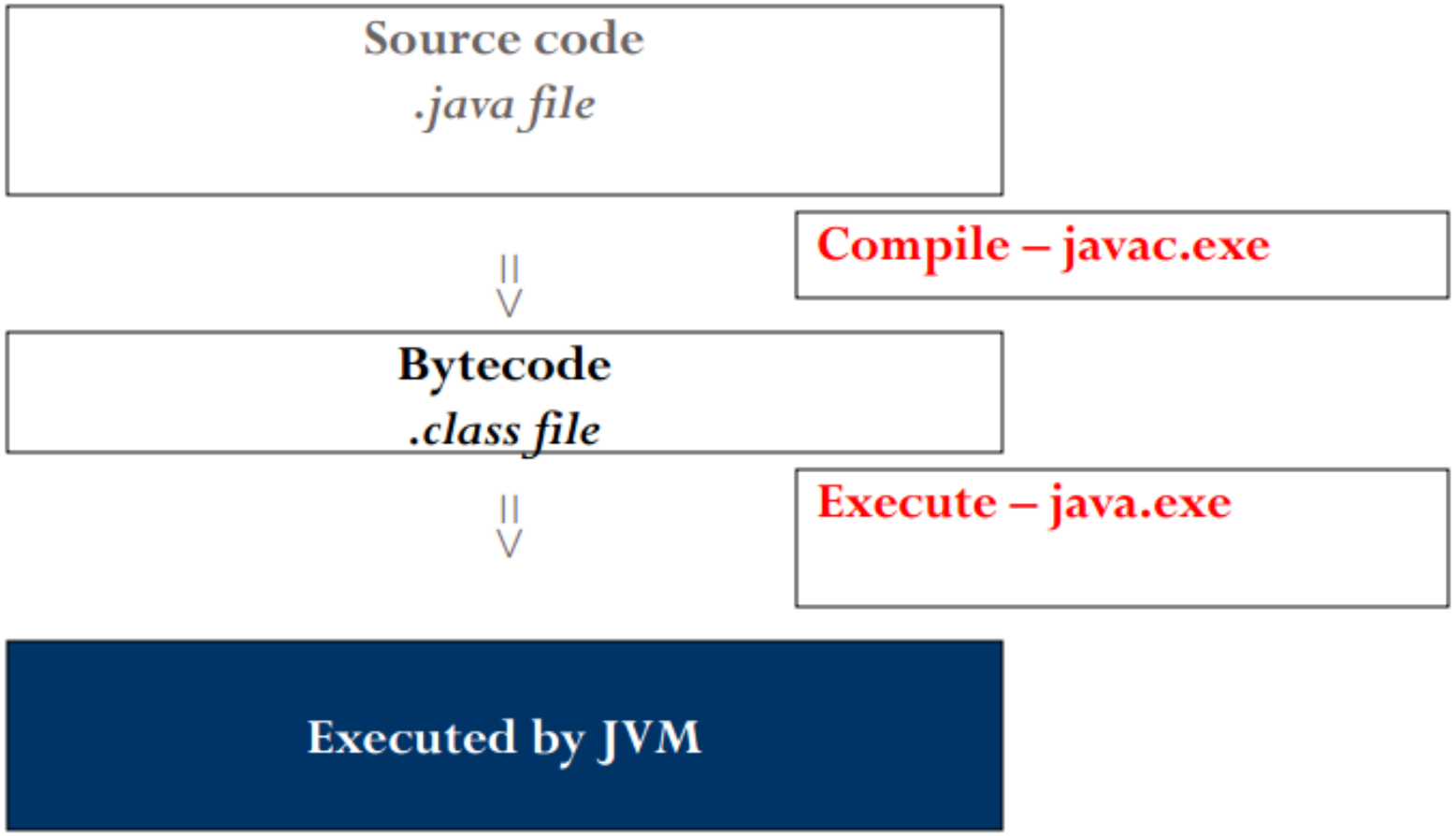
```
package p1;

public class Other
{
    Class1 c1;

    Class2 c2;
}
```

all classes are visible

Pași de dezvoltare a unei aplicații Java



Pași de dezvoltare a unei aplicații Java

Instrumente necesare:

- Un editor ASCII simplu (Notepad, Notepad++, etc.) pentru scrierea fișierelor de cod sursă;
- Compilatorul Java (**javac.exe**) pentru a compila fișierele de cod sursă (**.java**) și a obține fișiere bytecode cu extensia **.class**;
- Mașina virtuală JVM (**java.exe**) pentru a executa aplicațiile Java.

Pași de dezvoltare a unei aplicații Java

- Componentele **javac.exe** și **java.exe** se obțin prin instalarea Java SDK (JDK) de pe site-ul Oracle;
- Cele două executabile se găsesc în calea:

  C:\Program Files\Java\jdk1.7.0_71\bin

  sau

  C:\Program Files\Java\jdk-12.0.2\bin

  în funcție de versiunea JDK instalată.

Compilare în linie de comandă

C:\Windows\system32\cmd.exe

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\cristi>java -version
java version "1.6.0_33"
Java(TM) SE Runtime Environment (build 1.6.0_33-b03)
Java HotSpot(TM) Client VM (build 20.8-b03, mixed mode, sharing)

C:\Users\cristi>javac -version
javac 1.6.0_33

C:\Users\cristi>cd h:\javaex

C:\Users\cristi>h:

h:\javaex>javac.exe HelloWorld.java

h:\javaex>java.exe HelloWorld
Hello World Java!

h:\javaex>
```

Tipuri de date în Java

Value data type	Size	Range for signed values	Category
byte	**1 byte**	-128 –> 127	integer
short	**2 bytes**	-32768 –> 32767	integer
int	**4 bytes**	-2147483648 –> 2147483647	integer
long	**8 bytes**	-9,223,372,036,854,775,808 –> 9,223,372,036,854,775,807	integer
float	**4 bytes**	7 significant digits	real simple precision
double	**8 bytes**	15 significant digits	real double precision
char	**2 bytes**	'\u0000' –> '\uffff' 0 –> 65535	16 bits Unicode char
boolean	**1 bit**	true or false	logic value

Variabile în Java

- Variabile de tipuri primitive (int, float, char, boolean, etc.);
- Variabile de tipuri referențiale (obiecte, interfețe, enumerări, învelitori (wrappers) pentru tipuri primitive).

Variabile în Java

- Denumirile de variabile trebuie să înceapă cu o literă, simbolul underscore (_) sau simbolul dolar ($);
- Denumirile de variabile nu pot să înceapă cu o cifră;
- După primul caracter, se pot utiliza cifre în denumirile de variabile;
- Denumirile de variabile nu pot fi un cuvânt cheie sau rezervat din limbajul Java;
- Se pot defini mai multe variabile simultan.

Variabile în Java

- Denumirile de variabile sunt alese de programator, dar pentru eficiență se pot respecta următoarele convenții de nume:
  - Notația ungară;
  - Camel Case;
  - Java mixed case.

```
int iBooksNumber; //Hungarian Notation
int BooksNumber; //CamelCase
int booksNumber; //Java mixed case
```

Variabile în Java

- Java is strong type language;

```
float vb2 = 23.5; //compilation error -
  possible loss of precision
int vb3 = 45.6; //compilation error -
  possible loss of precision
boolean test = 23; //compilation error -
  incompatible types
```

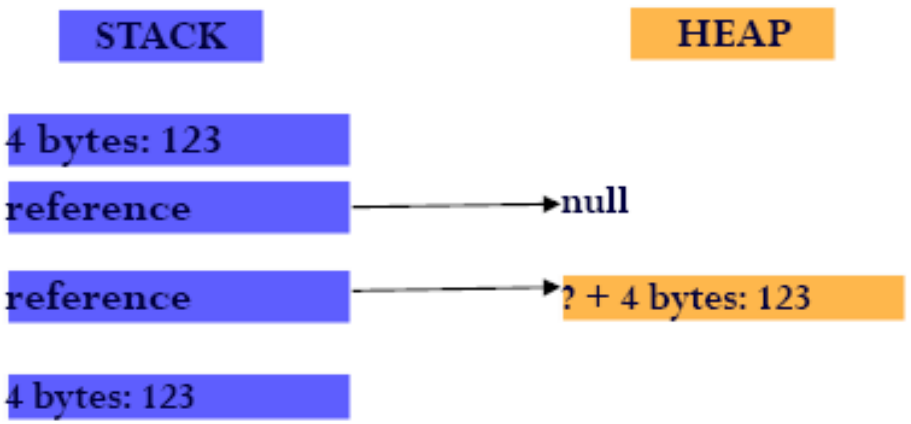
Variabile în Java

```
public static void main()
{
int sum;
//local variable declared in main
  method
sum = sum + 10;
//compiler error
//variable sum might not have been
  initialized
}
```

Boxing / Unboxing

Converting a value type into a reference type and backwards is done by **boxing / unboxing**

```
public class BoxUnbox
{
        static void main(String[] args)
        {
          int i = 123;
          Integer iObject;
          iObject = i;
          int j = iObject;
        }
}
```



Masive în Java

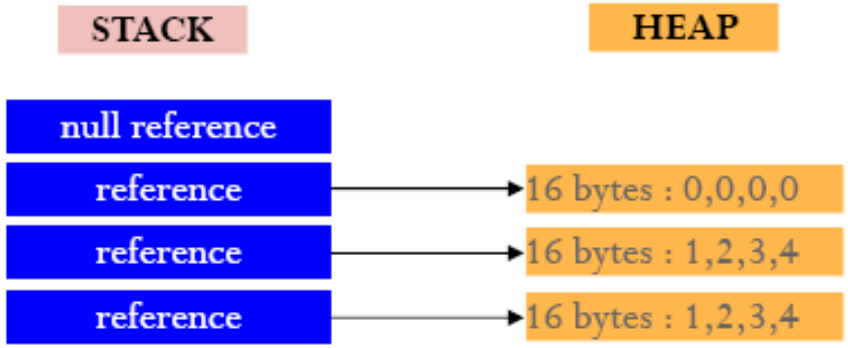
- un masiv este o formă particulară de obiect Java care este utilizat pentru a stoca o listă de elemente omogene (fiecare element al masivului are același tip ca tipul de bază);
- numărul de elemente din masiv este fix și definește lungimea sa;
- în Java, masivele sunt un tip special de obiecte - instanțe ale clasei Array;
- Java permite utilizarea operatorului index [ ] pentru a avea acces la elementele unui masiv;
- operatorul new acceptă ca parametru dimensiunea masivului.

Masive în Java

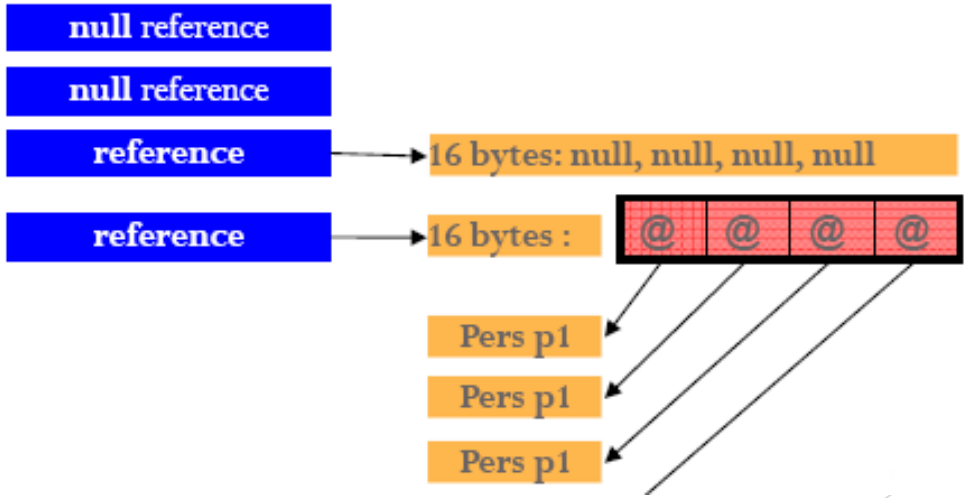
- Definirea masivelor unidimensionale (vectori) în Java:
  - **base_type[ ] array_name;**
  - **base_type array_name[ ];** //stil similar cu C/C++
- Inițializarea vectorilor în Java:
  - Se definește vectorul;
  - Se alocă spațiu de memorie pentru el;
  - Se inițializează elementele.
- Accesarea și prelucrarea vectorilor în Java:
  - Accesul la elemente cu operatorul index [ ];
  - Numărul de elemente este gestionat cu proprietatea *length*.

Masive în Java

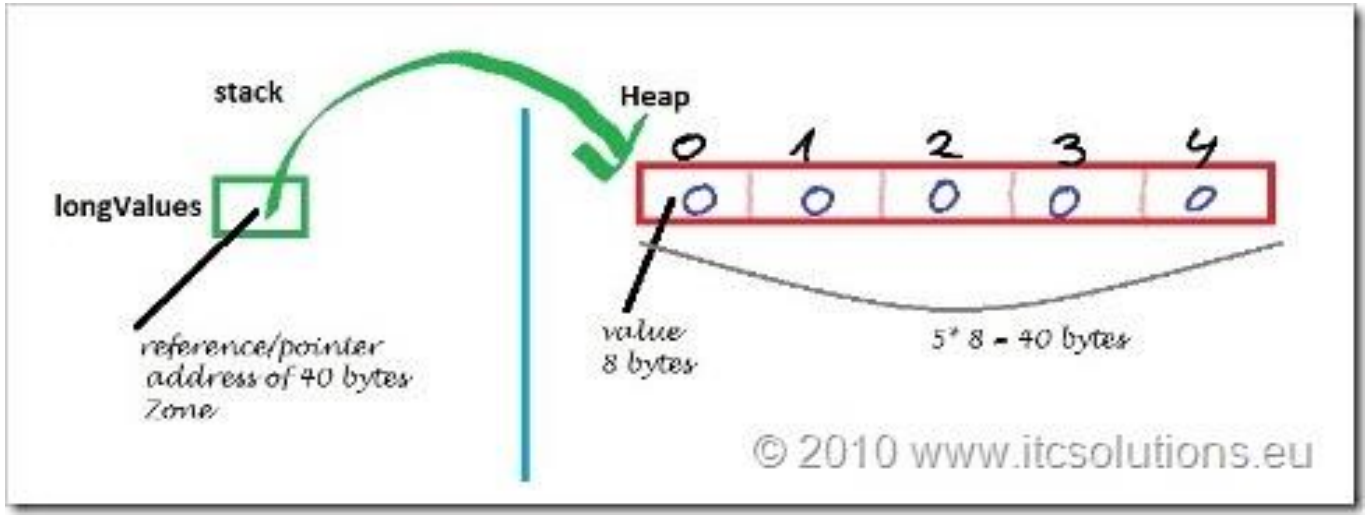
```
int [ ] vect;
vect = new int[4];
int [ ] vect2 = {1,2,3,4};
int [ ] vect3 = new int[] {1,2,3,4};
```

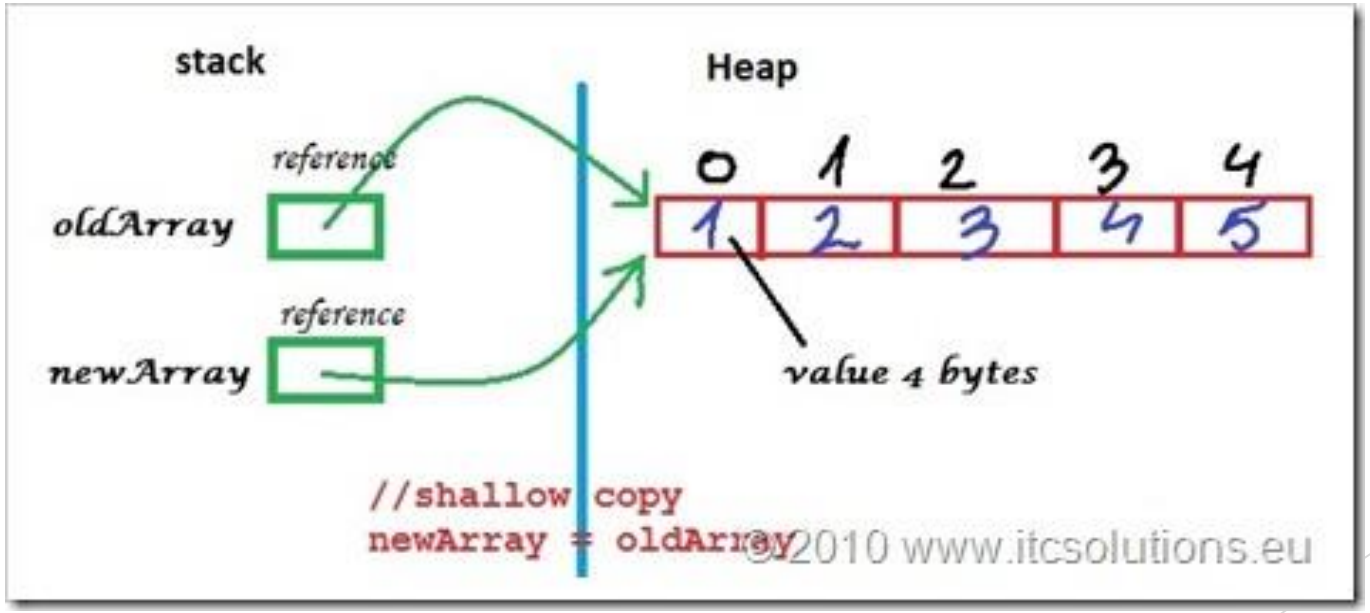



```
Pers p1;
Pers [ ] vectPers;
vectPers = new Pers[4];
vectPers = new Pers[4] {p1,p2,p3,p4};
```



Masive în Java

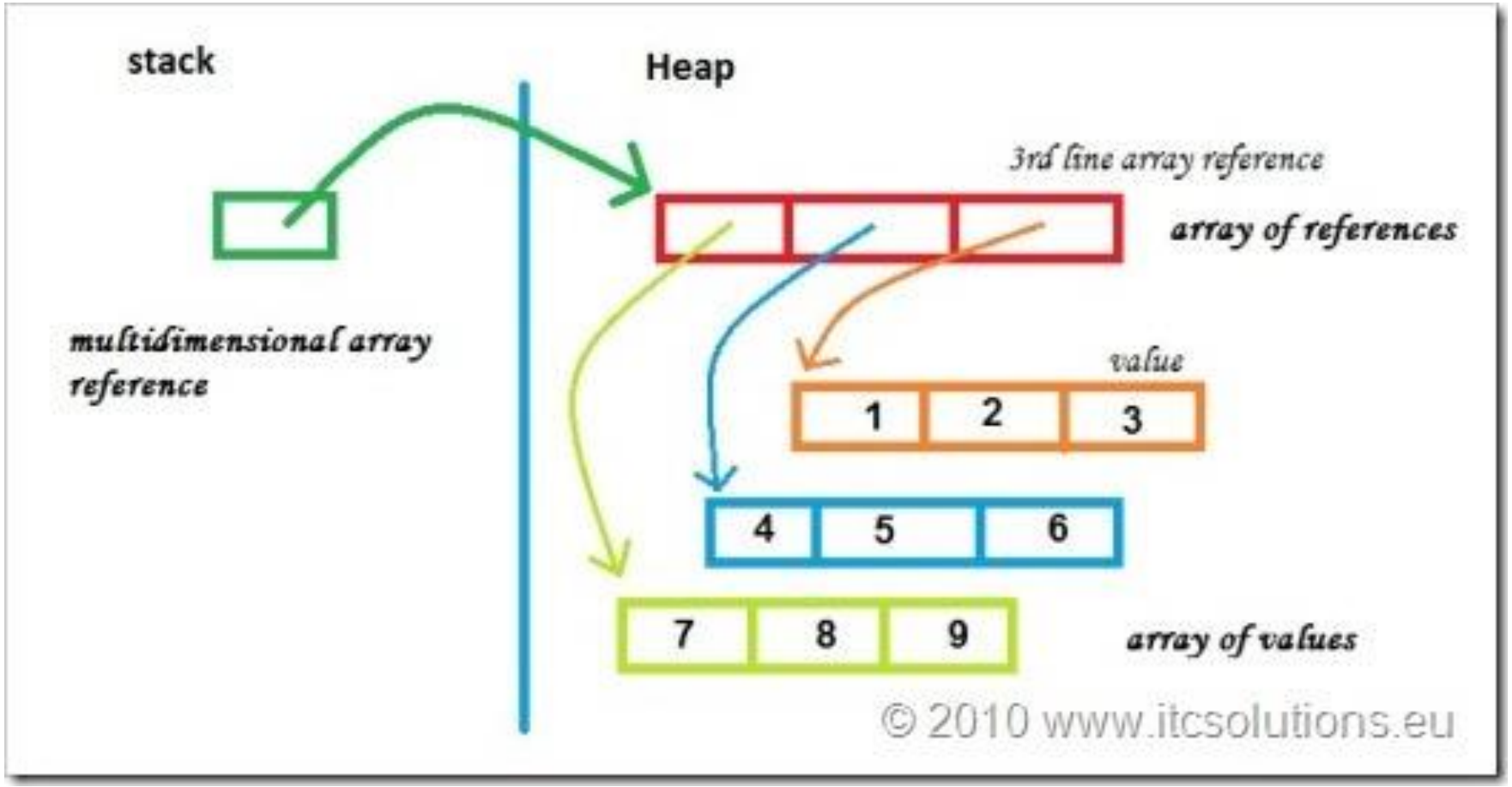




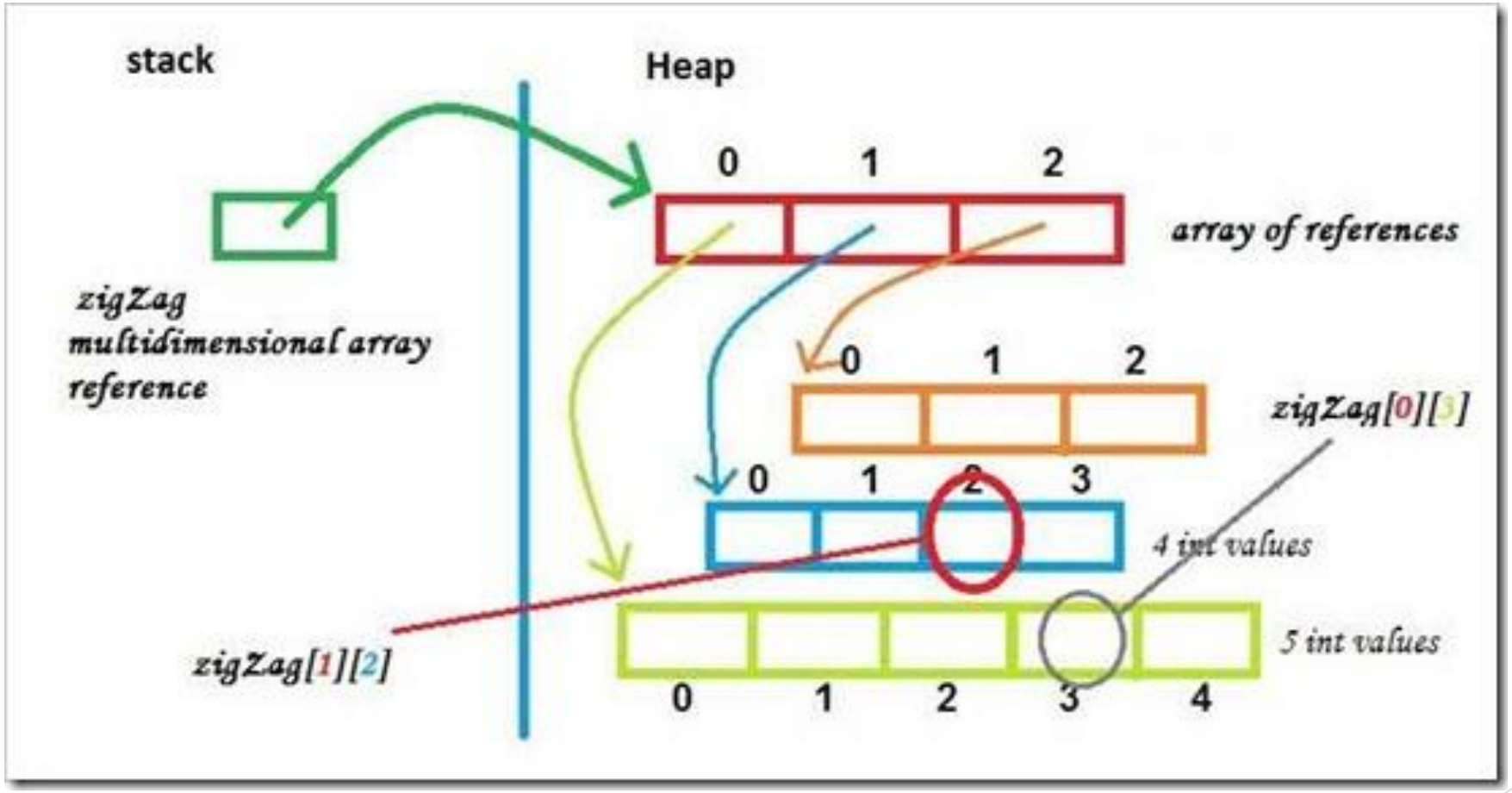
Masive în Java

- Definirea masivelor bidimensionale (matrice) în Java:
  - **base_type[ ][ ] matrix_name;**
  - **base_type matrix_name[ ][ ];** //stil similar cu C/C++
- Inițializarea matricilor în Java:
  - Se definește vectorul de vectori (liniile matricei);
  - Se alocă spațiu de memorie pentru el și se inițializează numărul de elemente aferent primei dimensiuni;
  - Se rezervă spațiu de memorie pentru fiecare vector care va stoca valori; deoarece fiecare vector este prelucrat separat, este posibil să se stabilească diferite dimensiuni și să se creeze o matrice în zig-zag;
  - Se inițializează elementele.

Masive în Java



Masive în Java



Stiva și Heap

Stiva:

- un spațiu de memorie rezervat de sistemul de operare pentru procesul aplicației;
- dimensiunea stivei este fixă și este determinată în faza de compilare pe baza declarărilor de variabile și alte opțiuni de compilare;
- este important să se stabilească faptul că stiva este limitată, iar dimensiunea sa este fixă (odată ce procesul a început, nu se mai poate modifica mărimea stivei);
- de cele mai multe ori, stiva este folosită pentru a stoca variabile ale funcțiilor sau metodelor (argumente de intrare și variabile locale);
- fiecare metodă are propria sa stivă, inclusiv metoda main(), care este, de asemenea, o funcție.
- o metodă există pe stivă numai pe timpul duratei de viață a acelei metode: din momentul apelului până la momentul returnării valorilor;

Stiva și Heap

Heap:

- un spațiu de memorie gestionat de sistemul de operare și utilizat de procese pentru a obține spațiu suplimentar la execuție;
- această zonă de memorie este globală, ceea ce înseamnă că orice proces o poate folosi (desigur, procesele nu pot citi sau scrie în zona Heap rezervată altui proces);
- rolul acestei zone de memorie este de a oferi resurse suplimentare pentru procesele care au nevoie de acest spațiu suplimentar la execuție (de exemplu, o aplicație simplă Java care construiește un vector cu valori de la consolă);
- spațiul necesar în timpul execuției unui proces este determinat de funcții cum ar fi *new* (aceeași funcție utilizată pentru a crea obiecte în Java), care sunt folosite pentru a obține spațiu suplimentar în Heap.

Stiva și Heap

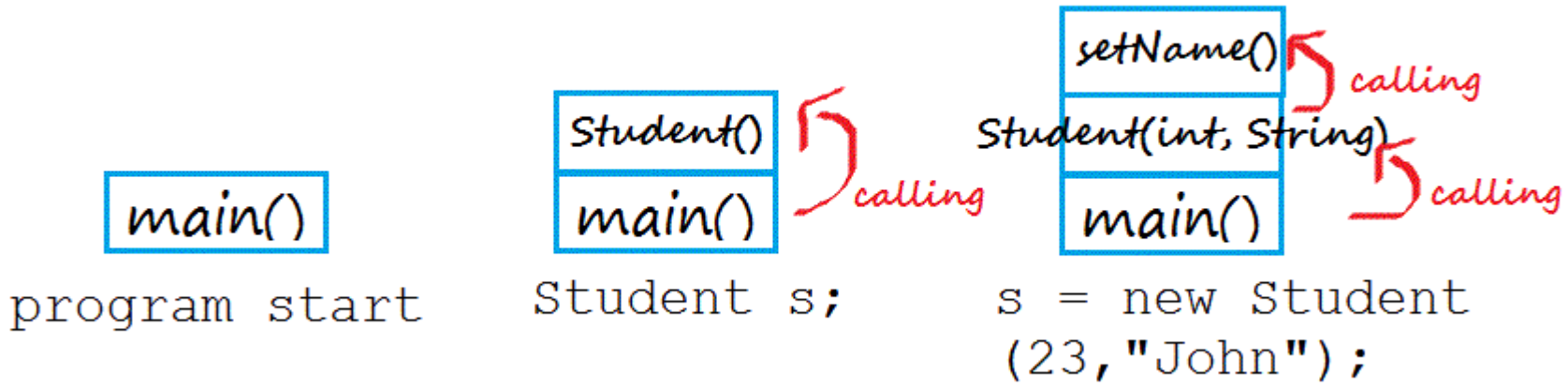
```
class Student {
    int age; //instance variable
    String name; //instance variable
    public Student() {
        this.age = 0;
        name = "Anonymous";
    }
    public Student(int Age, String Name) {
        this.age = Age;
        setName(Name);
    }
    public void setName(String Name) {
     this.name = Name;
    }
}
```

Stiva și Heap

```
public class Main {
    public static void main(String[] args) {
        Student s; //local variable - reference
        s = new Student(23,"John");
        int noStudents = 1; //local variable
    }
}
```

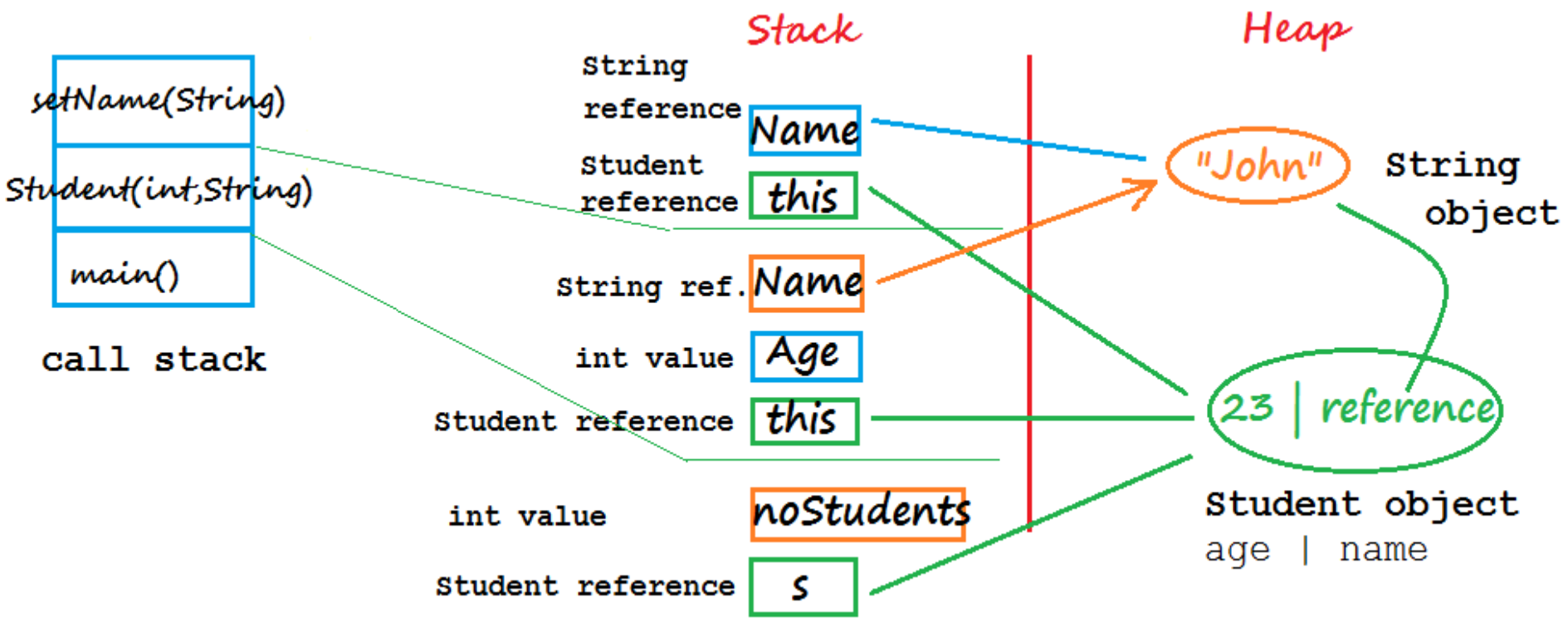
Stiva și Heap

- Exemplu apel stivă:



Stiva și Heap

- Valorile de pe stivă și din Heap:



Stiva și Heap

Regula generală în ceea ce privește locul în care variabilele sunt plasate în memorie:

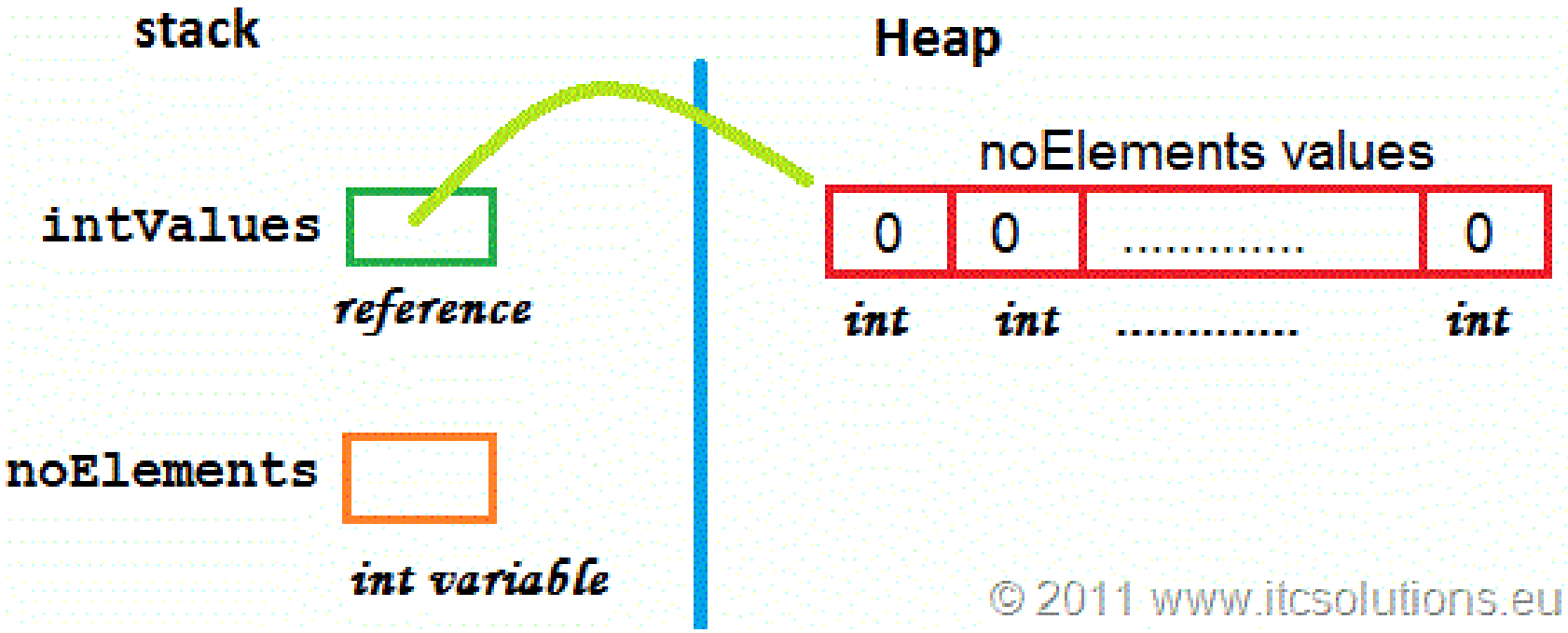
- variabilele declarate în blocuri de funcții (în Java nu se pot declara variabile globale, cum ar fi în C sau C++) sau în lista lor de parametri sunt plasate pe stivă (stiva funcției);
- orice valoare creată cu operatorul *new* este plasată în Heap.

Stiva și Heap

```
try {
    int[] intValues;
    int noElements = 0;
    System.out.println("The array elements number (0 - 255):");    //read number of elements
    noElements = System.in.read();
    intValues = new int[noElements];    //print the values - all with 0 default value
    for(int i=0; i < intValues.length; i++) System.out.print(" "+intValues[i]);
}
catch(IOException ex) {
    System.out.println(ex.getMessage());
}
```

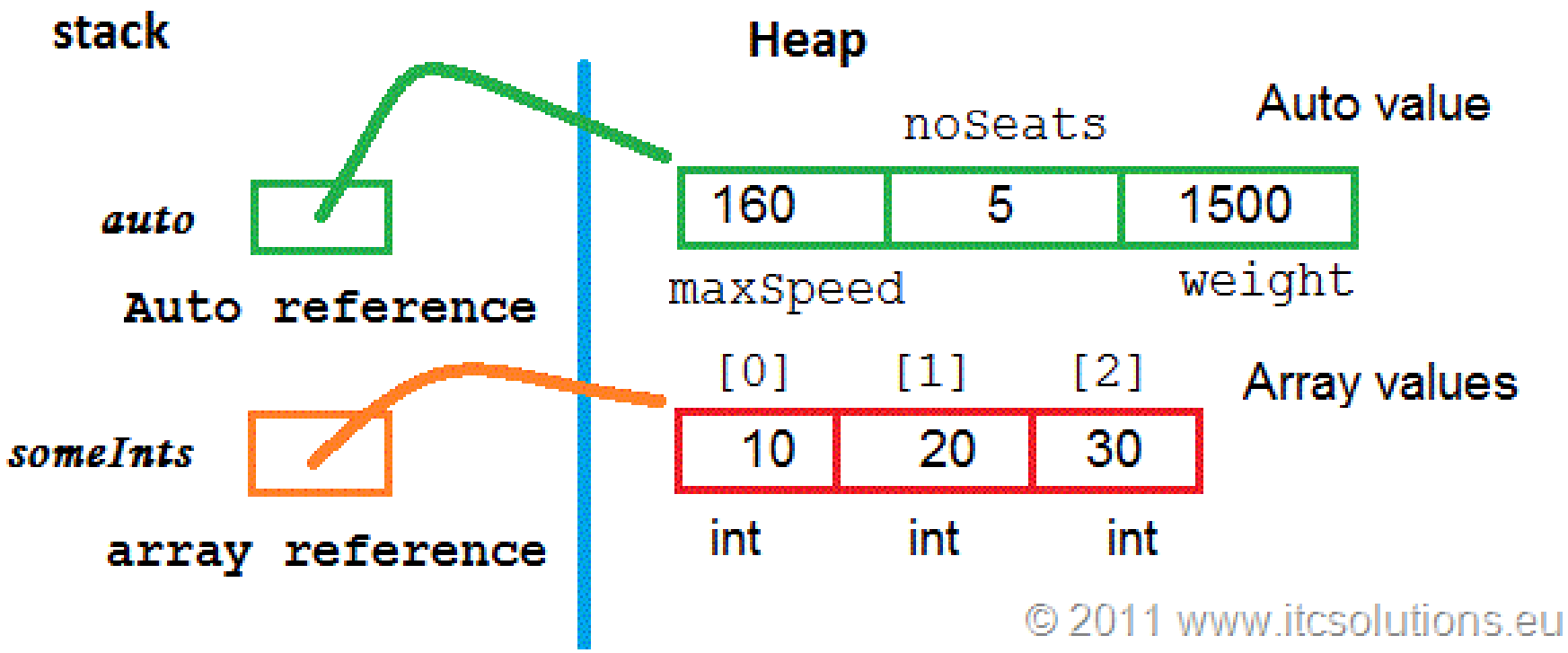
Stiva și Heap

- Valorile obiectelor și referințele în memorie:



Stiva și Heap

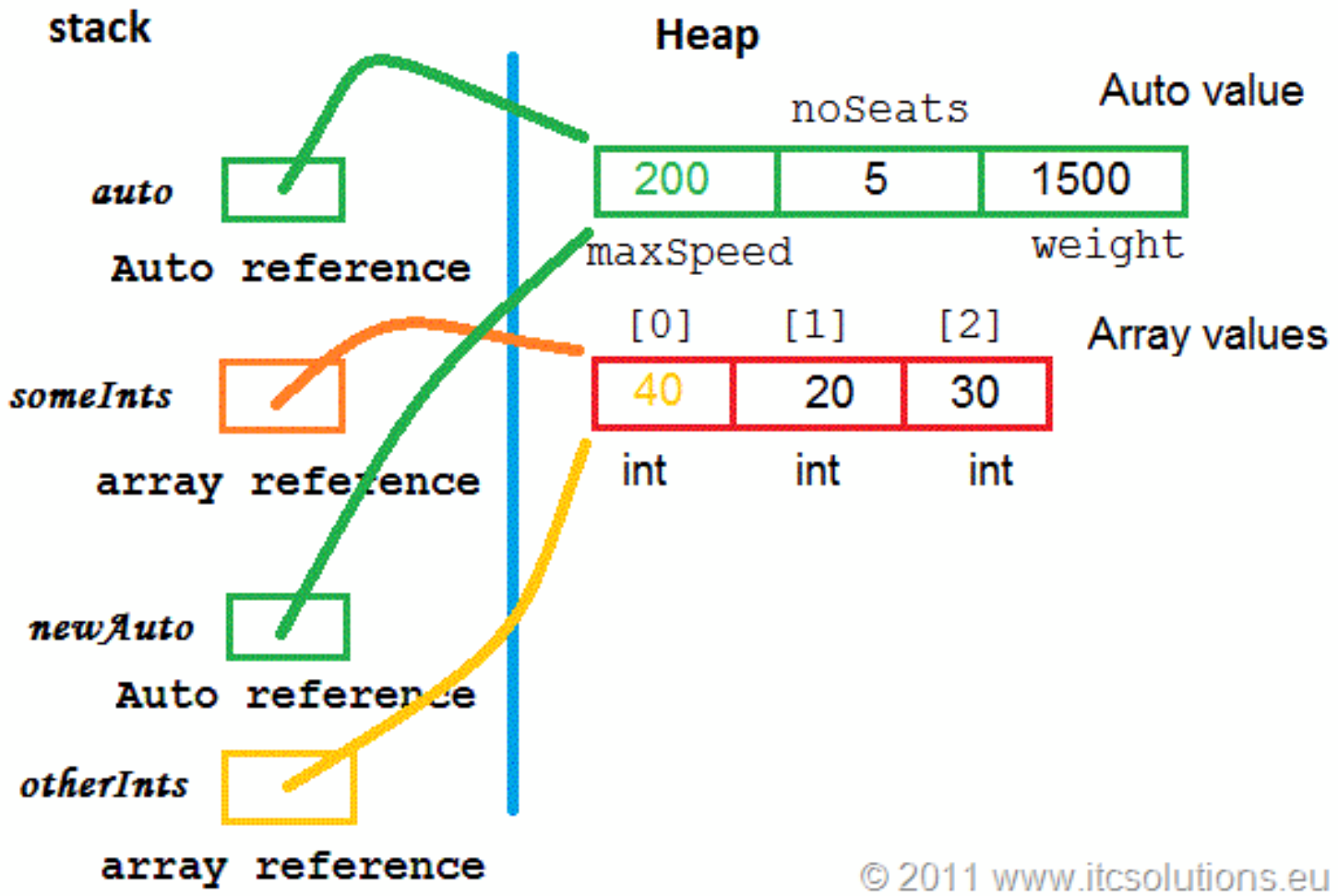
- Valorile obiectelor și referințele în memorie:



© 2011 www.itcsolutions.eu

Stiva și Heap

- Valorile obiectelor gestionate prin două referințe:



© 2011 www.itcsolutions.eu

Bibliografie

- [1] Jonathan Knudsen, Patrick Niemeyer – *Learning Java, 3rd Edition*, O'Reilly.
- [2] http://www.itcsolutions.eu
- [3] http://www.acs.ase.ro
- [4] http://docs.oracle.com/javase/tutorial/ /index.html

Curs 2 PPOO

Conf. univ. dr. Cristian CIUREA

Departamentul de Informatică și Cibernetică Economică

cristian.ciurea@ie.ase.ro

Java fundamentals

- Structuri de control
- Enumerări
- Transferul parametrilor
- Mecanismul try-catch
- String și immutable
- JavaDoc
- Internaționalizare

Structuri de control

Flow control structures:

- if-then
- if-then-else
- do-while
- while-do
- for
- enhanced-for
- switch

Structuri de control

IF-THEN

```
if (condition)
{
    < statement 1 >
    < statement 2 >
}
```

- *condition* is a boolean expression or variable that has the value *true* or *false* .For example *30> 10* or 10 == 30

- are not accepted conditions based on expressions or variables that have numeric values

Structuri de control

IF-THEN-ELSE

```
if (condition)
{
   < statement 1 >
   < statement 2 >
}
else
{
   < statement 1 >
   < statement 2 >
}
```

- an alternative is the conditional operator:

```
condition ? then_statement : else_statement
```

Structuri de control

DO-WHILE

```
do
{
   < statement 1>
   < statement 2>
} while (condition)
```

-the condition for exiting/staying in the loop is checked at the end of the loop block

- the structure will run at least once the iteration statements

Structuri de control

WHILE-DO

```
while (condition)
{
    < statement 1>
    < statement 2>
}
```

- the condition for exiting from/staying in the loop is checked before the first statement in the loop block is executed

Structuri de control

FOR

```
for(initialization; condition;
   iteration)
{
   < statement 1>
   < statement 2>
}
```

- like *do-while;*
- is more efficient, simply because the iteration and the initialization statements are included in the structure and not in the block;

Structuri de control

ENHANCED-FOR

```
for ( variable: iterable_collection ){
    < statement 1>
    < statement 2>
}
```

- used to iterate through a collection that implements the *java.lang.Iterable* interface

Structuri de control

SWITCH

```
switch (testValue) {
case constant1:
   <statement 1>
   break;
case constant2:
   < statement 2>
   break;
...
default:
```

- conditional structure with multiple branches;
- each *case* clause is closed with the *break* instruction because it provides the exit from the *switch* structure;
- break is optional;

Enumerări

- Definirea unei enumerări:
  - **enum EnumName {constant1, constant2, …, constantN};**
  - //ATENȚIE simbolul ; (punct și virgulă) de la sfârșit este opțional
- Declararea unei enumerări:
  - independent sau global, deoarece enumerările sunt văzute ca și clase
  - în interiorul unei alte clase
  - NU în interiorul metodelor

Enumerări

Reguli la definirea unei enumerări:

- simbolurile unei enumerări sau constantele sunt de obicei definite cu litere mari (cum ar fi DIESEL, BENZINA, etc.), pe baza convențiilor de nume ale Sun;
- simbolurile unei enumerări nu sunt întregi sau string-uri;
- enumerările declarate la nivel global pot fi definite doar default sau public (NU private sau protected), dar în acest ultim caz, în propriul fișier .java (aceeași regulă ca și pentru clase);
- enumerările declarate într-o altă clasă pot avea modificatori de acces (private, public, protected, default), care controlează vizibilitatea acestora în afara clasei părinte.

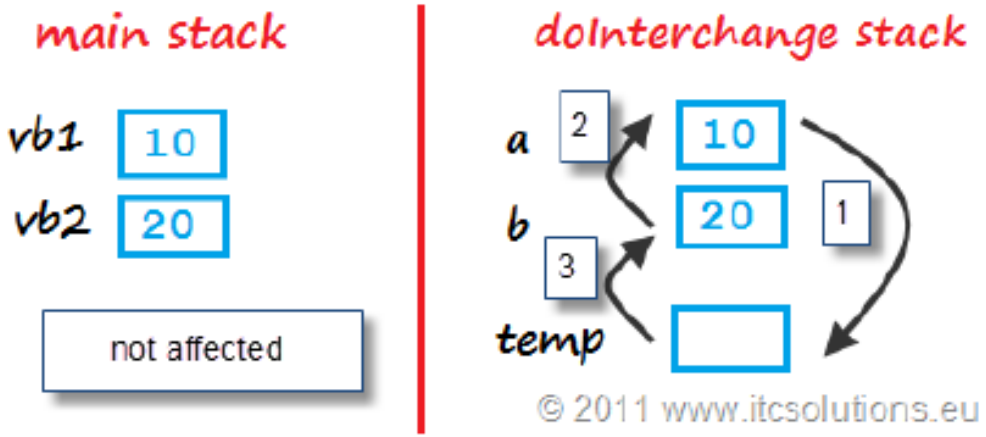
Enumerări

- enumerările sunt clase în Java și pot conține atribute, metode și constructori;
- fiecare enumerare furnizează o metodă statică, values(), utilizată pentru a itera peste constantele respectivei enumerări;
- simbolurile unei enumerări pot avea un corp constant de clasă specifică folosită pentru a înlocui o metodă generică.

Transferul parametrilor

- transfer parameters by their value

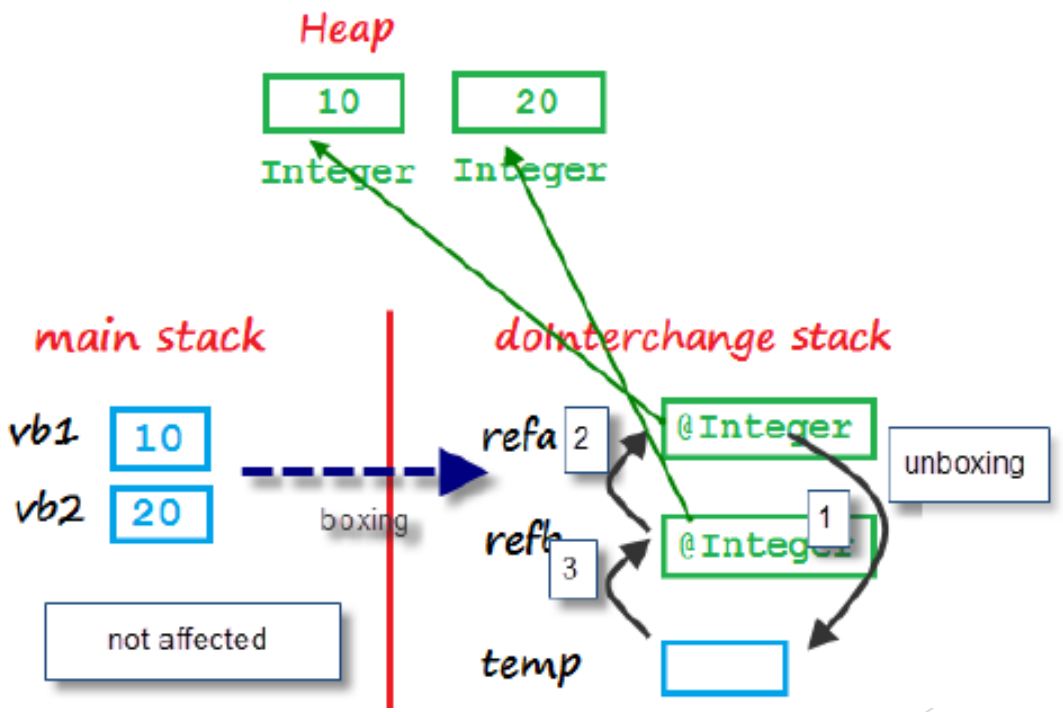
```
public static void doInterchange(int p, int q){
        int t = p;
        p = q;
        q = t;
  }
```



Transferul parametrilor

- transfer parameters using wrappers

```
public static void doInterchange(Integer ra, Integer rb){
        int temp = ra;
        ra = rb;
        rb = temp;
  }
```



Mecanismul try-catch

- Permite gestiunea situațiilor excepționale care conduc la terminarea imediată a unui program;
- Necesar pentru a realiza programe fiabile și robuste;
- Implementare prin **try**, **catch**, **finally** și **throw**.

Mecanismul try-catch

```
try
        {//statements}
catch(exception_type_1)
        { //particular statements}
catch(exception_type_2)
        {// particular statements}
catch(Exception)
        { //general statements}
finally
        {//must-do statements}
```

Mecanismul try-catch

- Blocurile **try - catch - finally** pot fi incluse în alte blocuri **try**;
- Programatorul poate să-și definească propriile excepții prin derivarea de clase din Exception;
- Funcția **throw** generează o nouă excepție.

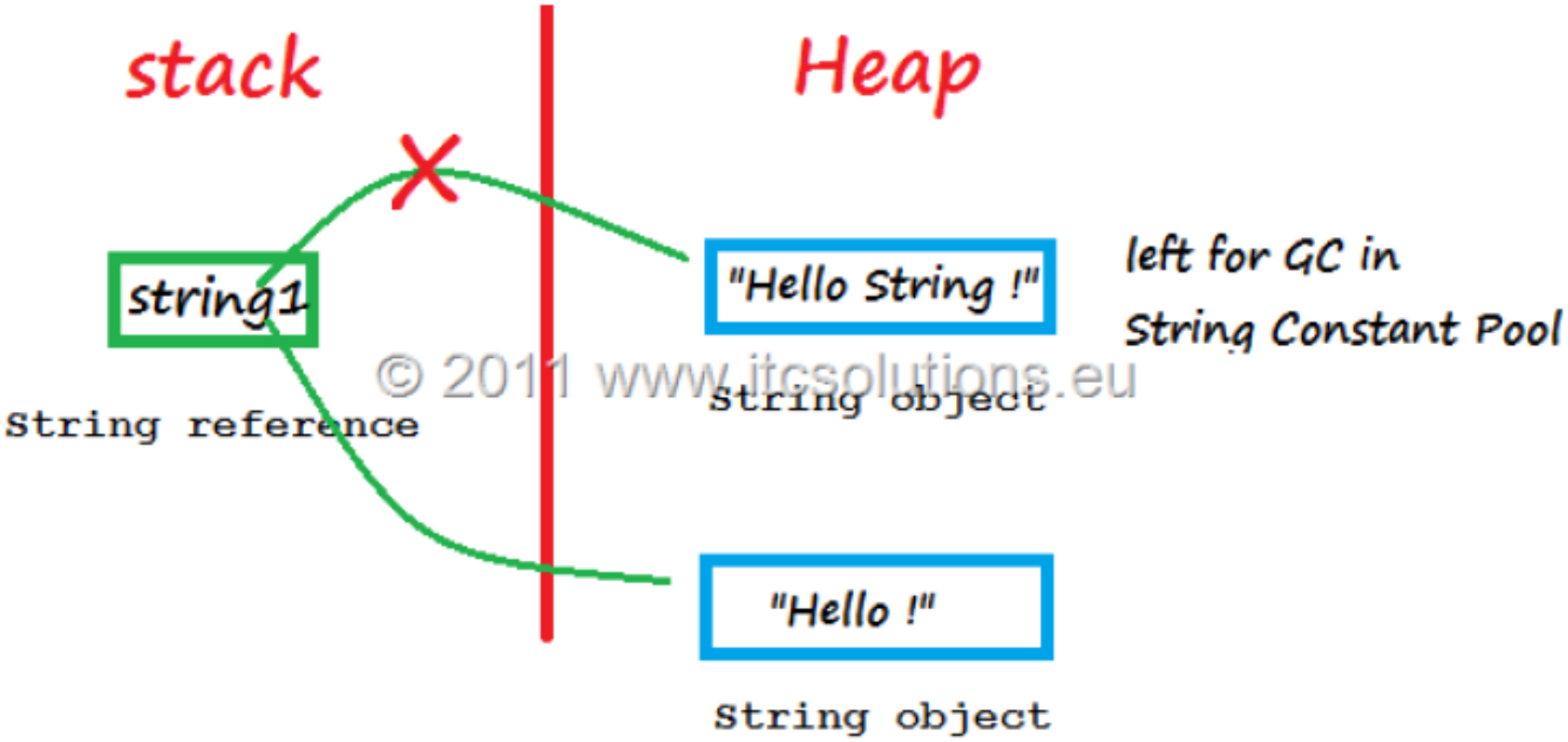
String și immutable

- În Java, fiecare caracter este o valoare Unicode pe 16 biți (2 B) și nu pe 1 B ca în C++;
- În Java, variabilele de tip string sunt gestionate prin obiecte din clasa String;
- În Java, sintaxa permite utilizarea obiectelor String ca tipuri de date primitive (se poate utiliza operatorul “=” pentru inițializarea acestora);
- În Java, variabilele de tip String sunt obiecte imuabile, adică odată ce au fost create nu-și mai pot modifica valoarea.

String și immutable

- Utilizarea operatorului “=” între două referințe String va copia valoarea referinței, nu valoarea obiectului;
- În Java, obiectele String sunt foarte speciale, pe de-o parte pentru că sunt imuabile și, pe de altă parte, pentru utilizarea eficientă a memoriei, JVM gestionează valorile String prin punerea lor într-o zonă specială de memorie denumită **String constant pool**.

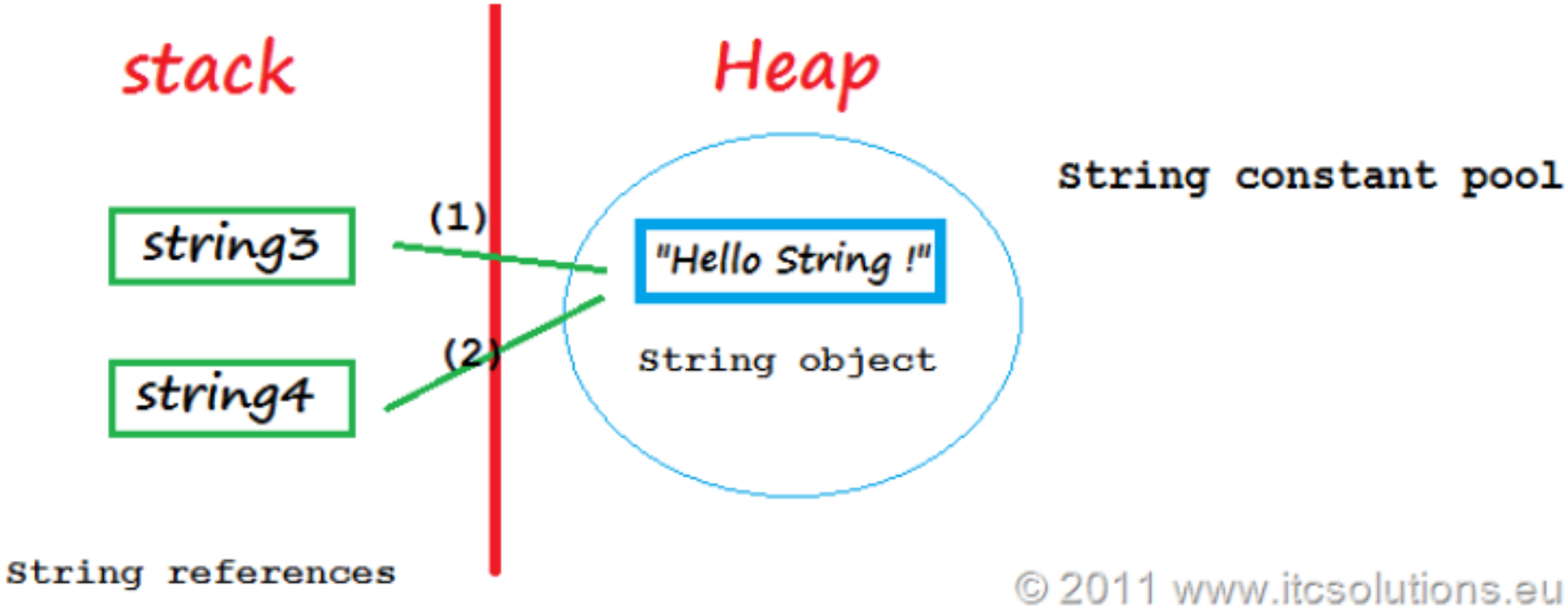
String și immutable



© 2011 www.itcsolutions.eu

String is Immutable

String și immutable



String și immutable

Method	Description
charAt()	returns the char at a given index; index takes values from 0 to length()-1;
concat()	appends a String to the end of another; the same as +
equals()	compare at case level 2 String values
length()	return the number of chars; IT IS NOT the length attribute of an array. IT IS A METHOD
replace()	replace occurences of a char with a given one
substring()	returns a substring
toLowerCase()	converts all chars to lowercase
toString()	returns the value of the String object
toUpperCase()	converts all chars to uppercase
trim()	remove whitespace from the end

These methods will NOT affect the current object.

String și immutable

StringBuilder și **StringBuffer**:

- cele două clase sunt aproximativ la fel;
- oferă o modalitate eficentă de lucru pentru operații de I/O pe fluxuri mari de date;
- valorile lor nu sunt stocate în *String constant pool*, ci se comportă ca obiecte normale.

String și immutable

Method	Description
append()	adds the argument to the end of the current object
delete()	deletes chars between a start and end index
insert()	inserts a value at a given offset
reverse()	reverse the value of the current object
toString()	returns the value of the StringBuilder or StringBuffer object

These method affects the value of the calling object, StringBuilder or StringBuffer.

JavaDoc

În activitatea de programare este important să se documenteze codul sursă, deoarece:

- vom avea o imagine clară asupra proiectelor complexe cu multe clase sau module;
- vom fi capabili, mai târziu, să înțelegem ce a fost făcut, astfel încât să putem modifica, adăuga sau șterge.

JavaDoc

- Pentru aplicațiile Java, documentația este furnizată de obicei în format HTML sub formă de arhive sau fișiere **.chm**.
- Pentru o documentație de cod sursă eficientă, proiectele JavaDoc sunt generate automat pe baza comentariilor de cod sursă cu ajutorul instrumentului **javadoc.exe**.

JavaDoc

Pentru a genera documentatia JavaDoc, comentariile de cod sursă trebuie să fie definite în conformitate cu anumite reguli:

- blocurile de comentarii JavaDoc încep cu /**
- blocurile de comentarii JavaDoc se termină cu */
- liniile de comentarii JavaDoc încep, prin convenție, cu *
- în timpul generării documentației JavaDoc, * sau spațiile de la începutul comentariului sunt ignorate
- comentariile sunt folosite pentru a genera documentația pentru metode sau clase
- deoarece documentația JavaDoc este în format HTML, comentariile pot conține tag-uri HTML folosite pentru a formata conținutul (de exemplu, <br> pentru a trece pe linia următoare)

JavaDoc

Există o serie de tag-uri JavaDoc pentru clase sau atribute speciale ale metodelor:

JavaDoc Tag	Meaning	Description
@see	Name of associated class	Class, method
@author	Author	Class
@version	Version	Class
@param	Input parameters	Method
@return	Return value	Method
@exception	Generated exception	Method
@throws	Generated exception	Method
@deprecated	Defines the element as deprecated	Class, method
@since	The API version in which this element was included	Class, method

Internaționalizare

- **Internaționalizarea (i18n)** este procesul de proiectare a unei aplicații, astfel încât să poată fi adaptată la diferite limbi și regiuni fără modificări majore.
- **Localizarea (l10n)** este procesul de adaptare a software-ului pentru o regiune sau o anumită limbă prin adăugarea de componente specifice de localizare și traducerea textului.

Internaționalizare

Un program internaționalizat are următoarele caracteristici:

- același executabil poate rula oriunde în lume;
- elementele textuale, cum ar fi mesajele de stare și etichetele componentelor GUI nu sunt hardcodate în aplicație;
- suportul pentru adăugarea de noi limbi nu necesită recompilarea aplicației;
- datele calendaristice și monedele apar în formate care sunt conforme cu regiunea utilizatorului final și limba selectată.

Internaționalizare

- Un obiect **Locale** pentru localizare este un identificator pentru o anumită combinație de limbă și regiune. În cazul în care o clasă își variază comportamentul în funcție de elementele regionale, se spune ca este **locale-sensitive**.
- De exemplu, clasa **NumberFormat** este sensibilă la setările regionale. Astfel, NumberFormat poate returna un număr 902 300 (Franța) sau 902.300 (Germania) sau 902,300 (Statele Unite).

Bibliografie

- [1] Jonathan Knudsen, Patrick Niemeyer – *Learning Java, 3rd Edition*, O'Reilly.
- [2] http://www.itcsolutions.eu
- [3] http://www.acs.ase.ro
- [4] http://docs.oracle.com/javase/tutorial/ /index.html

Curs 3 PPOO

Conf. univ. dr. Cristian CIUREA

Departamentul de Informatică și Cibernetică Economică

cristian.ciurea@ie.ase.ro

Java fundamentals

- Clase
- Shallow Copy vs. Deep Copy
- Moștenire (derivare)
- Polimorfism
- Clase abstracte
- Interfețe

Clase

Concepte de bază ale POO:

- Ce este o **clasă**? Ce este un **obiect**?
- Fiecare obiect conține date (**atribute/ câmpuri/ variabile instanță**) definite în cadrul clasei;
- O clasă definește o serie de funcții (**metode/ operații**) care aplicate unui obiect definesc **interfața** acestuia;
- Datele obiectelor sunt ascunse și pot fi accesate doar prin funcții definite în interiorul clasei => **încapsulare**;
- **Starea** unui obiect este definită prin atributele sale;
- **Comportamentul** unui obiect este definit prin metodele sale;
- Conceptul de **transmitere a unui mesaj unui obiect** este echivalent cu apelul unei metode.

Clase

- Sintaxa pentru definirea unei clase:

```
[attributes][acces_modifier] class class_name [extends
   base_class][implements interface1, interface2, ...]
{
   access_modifier attribute1;  //instance variable
   access_modifier attribute2; //instance variable
   ...
    access_modifier method1;
};
```

Clase

Modificatorii de acces:

- **public**
- **private**
- **protected**
- **default**

Atributele:

- **final**
- **abstract**

Clase

Instance members & methods visibility for access modifiers

Visibility	Public	Protected	Private	Default
Same class	X	X	X	X
Class in same package	X	X		X
Subclass in same package	X	X		X
Subclass in other package	X	X		
Class outside the package	X			

Clase

Cuvinte cheie:

- **extends**: permite derivarea clasei dintr-o altă clasă de bază;
- **implements**: permite derivarea clasei din una sau mai multe interfețe.

Clase

Atributele clasei:

- variabile instanță sau atribute ale obiectelor;
- variabile statice (un fel de variabile globale).

Metodele clasei:

- funcții constructor;
- funcții accesor (get() și set());
- alte metode de prelucrate.

Clase

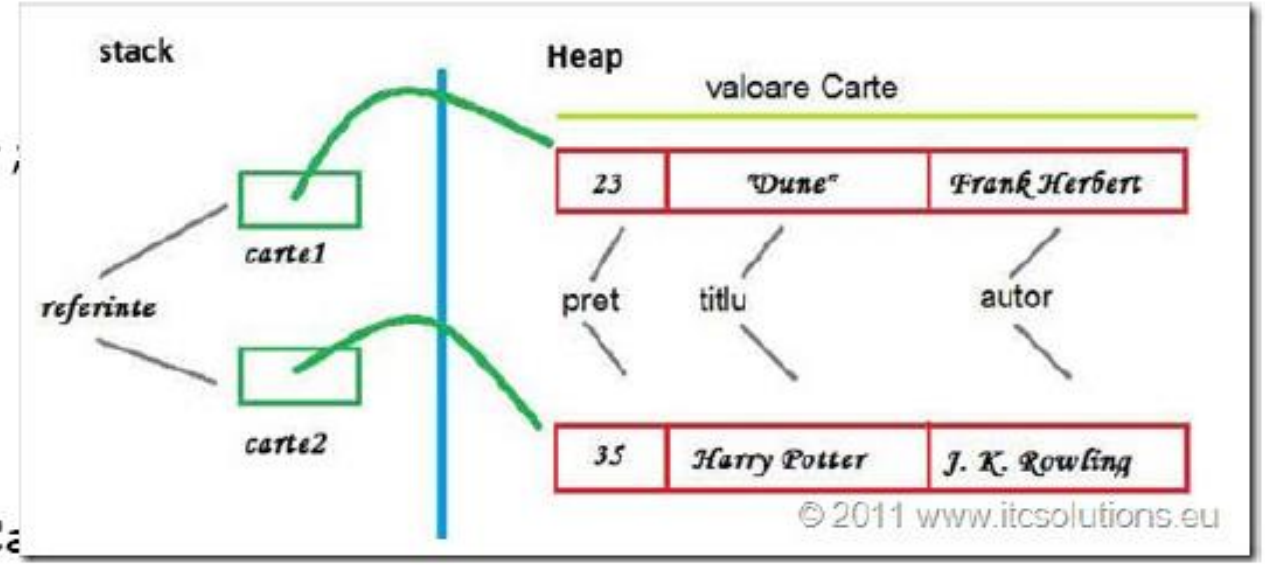
- Într-un fișier Java (.java) pot fi definite mai multe clase;
- Într-un fișier Java (.java) poate fi definită o singură clasă publică;
- Fișierul sursă Java conținând clasa publică poartă denumirea clasei publice;
- Prin compilarea unui fișier sursă Java ce include mai multe clase se obțin fișiere bytecode (.class) pentru fiecare clasă.

Clase

- Crearea obiectelor se face prin apelul operatorului **new** care apelează constructorul clasei
- Accesarea metodelor și atributelor unui obiect se realizează cu ajutorul operatorului punct (.)

Clase

```
class Book{
    //define attributes - instance variables
    float price;
    String title;
    String author;
}
...
```




```
Book book1 = new Ca
book1.price = 23;
book1.title = "Dune";
book1.author = "Frank Herbert";
```

Clase

Atributele **constante**:

- definite cu **final**;
- nu se poate modifica valoarea acestora odată ce au fost inițializate;

```
class Test
{
  public final int attribute_1 = 10;
//MUST be initialized in constructor
  public final int attribute_2;
}
```

Clase

Atributele **constante**:

- sunt inițializate în constructor sau la definire (dar nu în ambele situații);
- sunt echivalente variabilelor constante din C++.

Clase

Atributele **statice**:

- sunt definite prin cuvântul static;
- reprezintă atribute ce nu aparțin unui obiect;
- pot fi definite și **final**;
- inițializarea se face la definire sau într-un bloc de inițializare;
- sunt considerate variabile definite la nivelul clasei (un fel de variabile globale);
- sunt accesate prin numele clasei.

Clase

static attributes:

sintax:

```
class Test
{
   public static int attribute_1 = 10;
   public static final int attribute_2;
    //initialisation block
    //when is executed ?
    {
       attribute_2 = 45;
    }
}
```

Clase

Funcții statice:

- Sunt funcții care nu aparțin unui obiect;
- Sunt “funcții globale” care aparțin unei clase de obiecte;
- Au acces doar la ceilalți membri statici ai clasei;
- Sunt apelate prin numele clasei;
- Nu pot avea referința *this*.

Clase

Funcții membre:

- Constructori (se poate implementa inclusiv constructor de copiere);
- Nu există destructor (doar funcțai finalize());
- Funcții de acces (get și set);
- Nu se pot supraîncărca operatori ca în C++.

Clase

Constructors:

- syntax:

```
class class_Name {
public class_Name( ){...}
};
```

- use (because objects are managed by references, an object is created using *new* operator):

```
public static void main () {
class_Name object_1 = new class_Name();
class_Name object_2 = new class_Name(input params)
}
```

Clase

Copy constructor:

- syntax:

```
class class_name{
        public class_name(class_name existing_object){...}
            };
```

- use:

```
public static void main (){
class_name object_1 = new class_name(...);
class_name object_2 = new class_name(object_1);
}
```

Clase

Operatorul =

- Copiază bit cu bit valoarea sursei în zona de memorie destinație (cele două zone sunt identice ca tip și structură);
- În cazul obiectelor unei clase, copiază valoarea referinței obiectului sursă în referința obiectului destinație.

Clase

Proprietățile (getteri și setteri):

- Oferă acces (citire/scriere) la atributele private ale clasei;
- Presupun validarea datelor de intrare;
- Sunt definite în zona publică a clasei;
- Definite prin două metode prefixate cu **get** pentru citire și **set** pentru scriere.

Clase

```
class Test {
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

Shallow copy vs. Deep copy

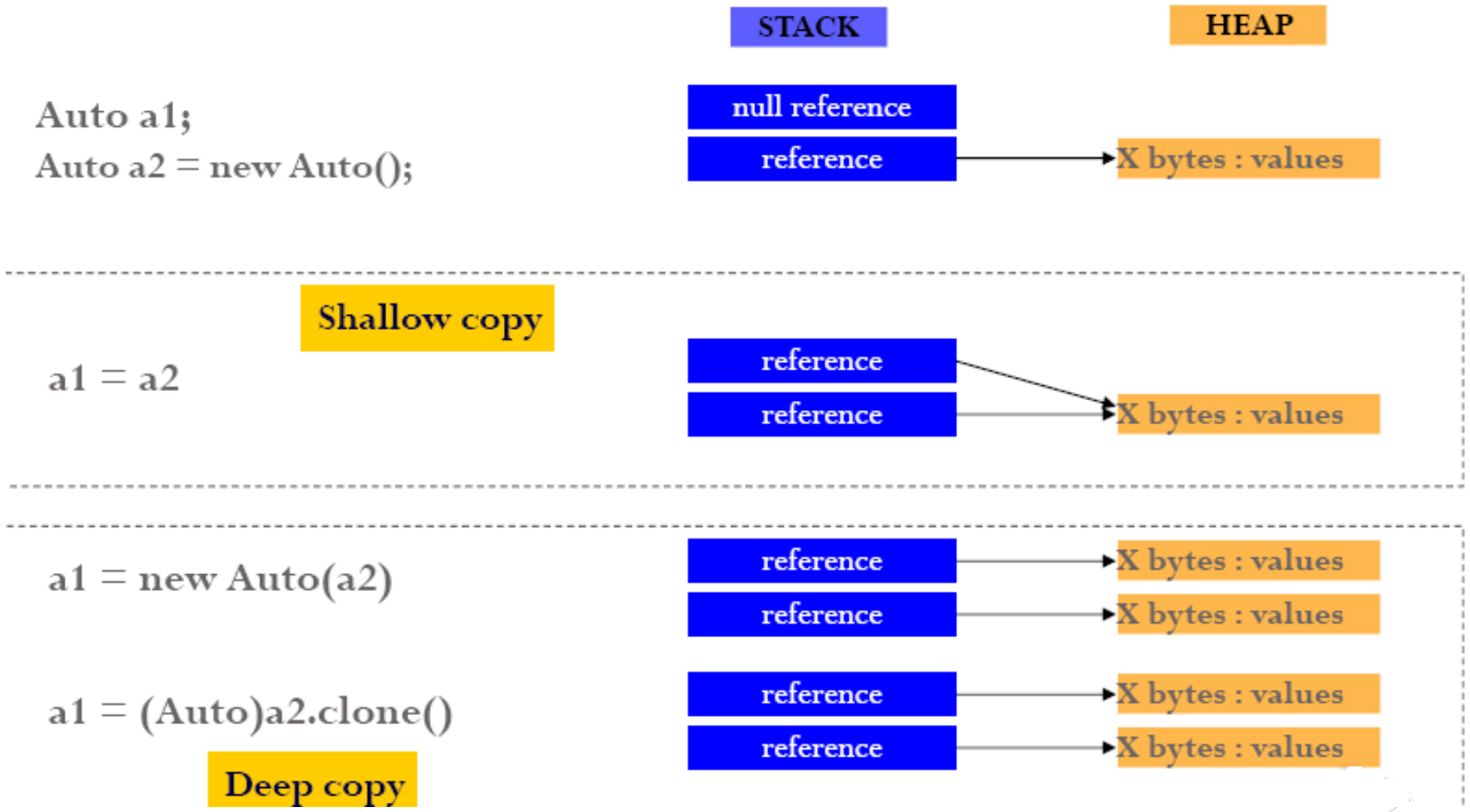
Shallow copy:

- Copiază valorile a două referințe între ele;
- Realizat implicit prin operatorul egal “=“;

Deep copy:

- Copiază valorile obiectelor (nu referințele);
- Realizat prin metode speciale cum ar fi constructorul de copiere, metoda clone().

Shallow copy vs. Deep copy



Moștenire (derivare)

- Este implementată atunci când există o relație de tipul **“is a”** între o subclasă și o clasă de bază;
- Se poate moșteni o singură clasă de bază;
- Apelul constructorului clasei de bază se face cu *super*.

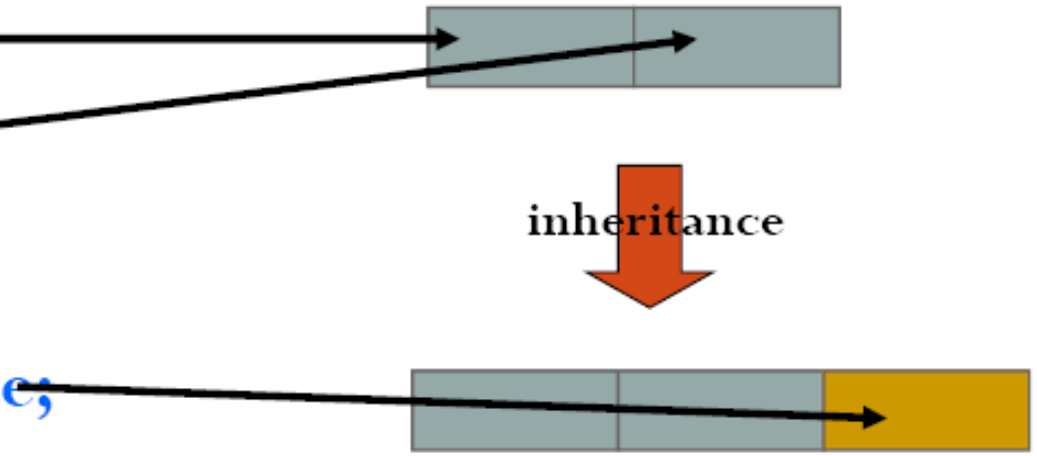
```
class SpecialProduct extends Product
{       private float _discount;
                ...
}
```

```
public SpecialProduct(double cost, float profit, float discount)
        {
                super(cost,profit);
                if(discount>0) _discount=discount;
        }
```

Moștenire (derivare)

through inheritance the subclass gets all the methods and attributes

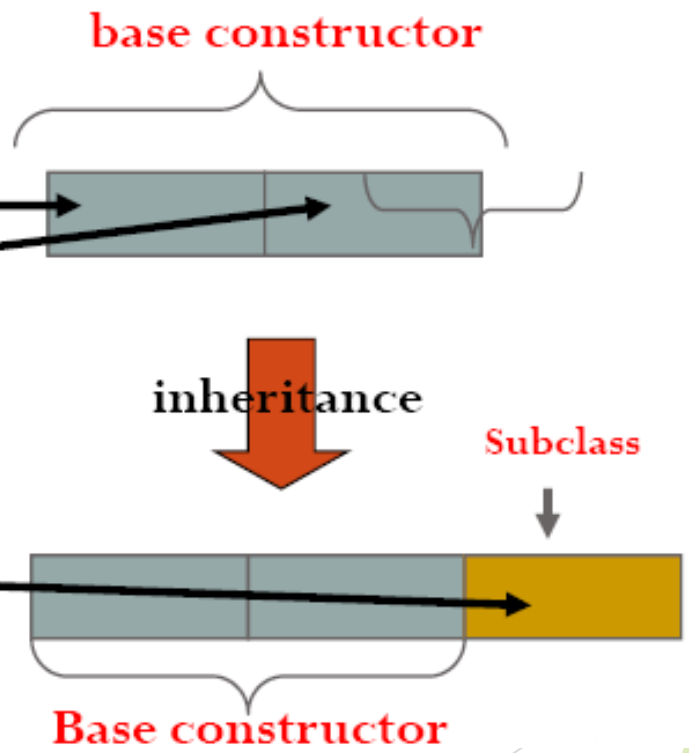
```
class Base{
        int attribute1;
        int attribute2;
};
class Subclass: Base{
        int new_attribute;
};
```



Moștenire (derivare)

each constructor manages the area of its class

```
class Base{
        int attribute1;
        int attribute2;
};
class Subclass: Base{
        int new_attribute;
};
```



Moștenire (derivare)

in the base class and the subclass you can define methods with the same header - overloading

```
class Base{
        int Method1(int a){...}
};
class Subclass: Base{
        int attribute_new;
        int Method1(int a){...}
        int Method2(int a){ super.Method1(a);}
};
```

call to the base class

Polimorfism

Overriding

```
class Person{
    public string SaySomething(){...}
}
```

same signature

```
class Student extends Person{
    public string SaySomething(){...}
}
```

Overloading

```
class Person{
    public void Eat(){...}
    public void Eat(Food food){...}
}
```

different parameters list

```
class Student extends Person{
    public void Eat(int calories){...}
    public void Eat(string foodName){...}
    public string Eat(string foodName,
string drinkName){...}
}
```

Static Binding vs. Dynamic Binding

Există două tipuri de binding:

- **statică** (cunoscută și sub denumirea de legare **timpurie**).
- **dinamică** (cunoscută și sub numele de legare **târzie**).

Static Binding vs. Dynamic Binding

- Când tipul obiectului este determinat la compilare (de către compilator), acesta este cunoscut sub numele de legare statică.
- Dacă există vreo metodă privată, finală sau statică într-o clasă, există o legare statică.

```
class Dog{
    private void eat(){System.out.println("dog is eating...");}

    public static void main(String args[]){
    Dog d1=new Dog();
    d1.eat();
 }
}
```

Static Binding vs. Dynamic Binding

- Când tipul obiectului este determinat în timpul rulării, acesta este cunoscut sub numele de legare dinamică.

```
class Animal{
    void eat(){System.out.println("animal is eating...");}
}

class Dog extends Animal{
     void eat(){System.out.println("dog is eating...");}

    public static void main(String args[]){
    Animal a=new Dog();
    a.eat();
 }
}
```

Clase abstracte

Metode virtuale pure (abstracte):

- Funcții virtuale care nu au implementare în clasa părinte;
- Clasa părinte trebuie să fie abstractă;
- Clasa derivată trebuie să le implementeze dacă nu este abstractă.

Clase abstracte

PURE VIRTUAL methods - ABSTRACT:

```
abstract class Superclass{
public abstract int Method1(int a) ;
};

class Subclass extends Superclass{
public int Method1(int a){...}
};
```

Clase abstracte

Clasele abstracte:

- Conțin cel puțin o metodă abstractă (virtuală pură), dar nu este obligatoriu;
- Pot conține atribute și alte metode;
- Sunt o interfață pentru alte clase care trebuie să implementeze un set de metode comune.

Clase abstracte

Abstract classes:

- you can't instantiate an abstract class;
- used for class hierarchies

```
abstract class AbstractClass{
        int attribute1;
        public abstract int Method1(int a);
};
....
AbstractClass ba1;
AbstractClass ba1 = new AbstractClass();
}
```

Interfețe

- Interfețele sunt clase abstracte care conțin doar funcții abstracte;
- Sunt o interfață pentru alte clase care trebuie să implementeze un set de metode comune;
- Sunt definite prin cuvântul cheie *interface*.

Interfețe

Interfaces:

```
interface IOperations {
        void Operation1( );
        void Operation2( );
};
class Base implements IOperations {
        public void Operation1() {…}
        public void Operation2() {…}
}
```

override interface method

Interfețe

Abstract classes

- contain abstract methods + attributes + non-abstract methods
- a class can extend only one base class (abstract or not)
- can be used as *reference type*

vs.

Interfaces

- contain only abstract methods
- a class can implement on or many interfaces
- can be used as *reference type*

Bibliografie

- [1] Jonathan Knudsen, Patrick Niemeyer – *Learning Java, 3rd Edition*, O'Reilly.
- [2] http://www.itcsolutions.eu
- [3] http://www.acs.ase.ro
- [4] http://docs.oracle.com/javase/tutorial/ /index.html

Curs 4 PPOO

Conf. univ. dr. Cristian CIUREA

Departamentul de Informatică și Cibernetică Economică

cristian.ciurea@ie.ase.ro

Java fundamentals

- Colecții în Java (**Java Collections Framework**)

Colecții

Java Collections Framework este o arhitectură unificată pentru reprezentarea și manipularea colecțiilor ce conține:

- **interfețe**: permit colecțiilor să fie folosite independent de implementările lor;
- **implementări;**
- **algoritmi**: metode de prelucrare (căutare, sortare) pe colecții de obiecte oarecare. Algoritmii sunt *polimorfici*, adică pot fi folosiți pe implementări diferite de colecții, deoarece le abordează la nivel de interfață.

Colecții

Colecțiile oferă implementări pentru următoarele tipuri:

- **mulțime** (ordinea elementelor este neimportantă);
- **listă** (ordinea elementelor contează);
- **tabel asociativ** (perechi cheie-valoare).

și pot lucra cu orice tip de obiecte (implementările sunt *generice*).

Colecții

Tipuri de colecții:

- **sortate** (TreeMap, TreeSet, PriorityQueue);
- **nesortate** (ArrayList, HashMap, Hashtable, etc.);
- **ordonate** (LinkedHashMap, LinkedHashSet, LinkedList, etc.);
- **neordonate** (HashMap, Hashtable, HashSet).

Colecții

Class	Map	Set	List	Ordered	Sorted
HashMap	X				
Hashtable	X				
TreeMap	X			sorted	X
LinkedHashMap	X			by insertion	
HashSet		X			
TreeSet		X		sorted	X
LinkedHashSet		X		by insertion	
ArrayList			X	by index	
Vector			X	by index	
LinkedList			X	by index	
PriorityQueue				sorted	X

Colecții

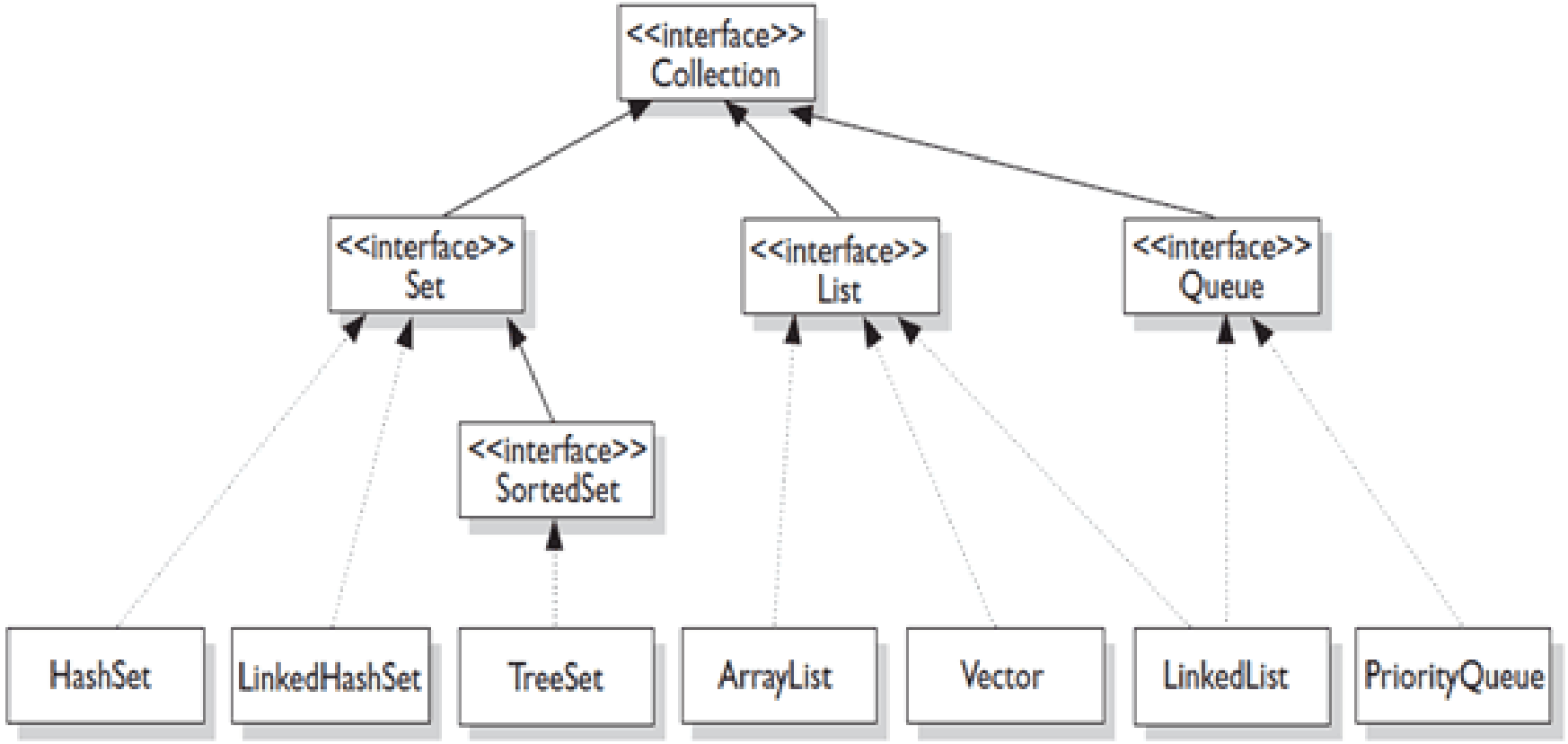
Tipuri de colecții:

- **Lists** – liste de valori (**ArrayList, Vector, LinkedList**);
- **Sets** – liste de valori unice (**HashSet, TreeSet, LinkedHashSet**);
- **Maps** – liste de valori cu un ID unic (**HashTable, HashMap, TreeMap, LinkedHashMap**);
- **Queues** – liste de valori procesate într-o ordine specifică.

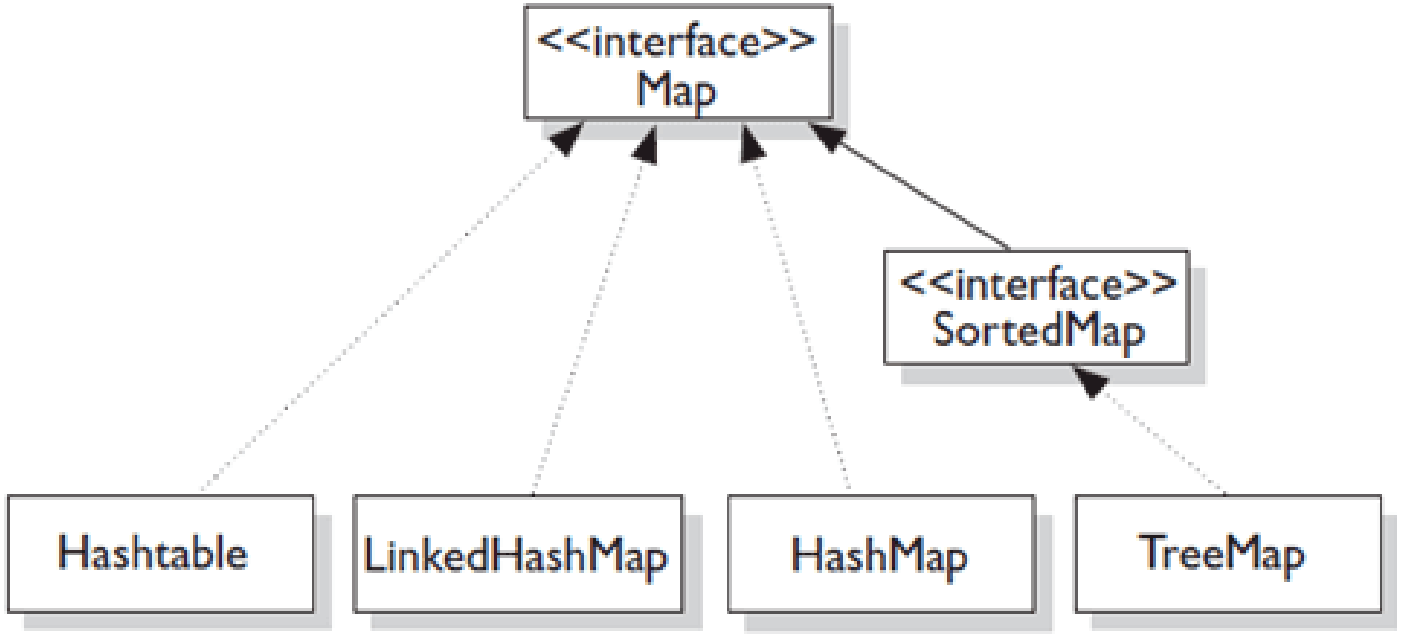
Colecții

- Ierarhia de colecții din Java este formată din două categorii distincte de interfețe.
- Prima categorie asigură funcționalitățile de bază utilizate de toate colecțiile, cum ar fi metode de adăugare și/sau eliminare de elemente.
- Subinterfețele sale - *Set*, *List* și *Queue* - furnizează metode pentru o serie de colecții specializate.
- Cea de-a doua categorie include interfața *Map* care mapează perechi cheie - valoare similar cu o tabelă de dispersie.

Colecții


<<interface>>
Collection
<<interface>>
Set
<<interface>>
List
<<interface>>
Queue
<<interface>>
SortedSet
HashSet
LinkedHashSet
TreeSet
ArrayList
Vector
LinkedList
PriorityQueue

Colecții


<<interface>>
Map
<<interface>>
SortedMap
Hashtable
LinkedHashMap
HashMap
TreeMap

Colecții

- Interfața **List** oferă metode pentru o colecție ordonată, pentru situațiile în care este nevoie de un control precis asupra locului în care se introduce fiecare element. Se pot localiza elemente dintr-o listă **List** după poziția lor exactă.
- O listă este o colecție ordonată. Listele pot conține elemente duplicate. Pe lângă operațiile moștenite de la interfața **Collection**, interfața **List** definește următoarele operații:
  - **T get(int index)** - întoarce elementul de la pozitia index;
  - **T set(int index, T element)** - modifica elementul de la pozitia index;
  - **void add(int index, T element)** - adauga un element la pozitia index:
  - **T remove(int index)** - sterge elementul de la pozitia index.

Colecții

- Interfața **Set** nu permite elemente duplicate și are o subinterfață, **SortedSet**, care oferă metode pentru ordonarea elementelor din structură.
- Exista trei implementări utile pentru **Set**:
  - **HashSet**: memorează elementele într-o tabelă de dispersie; este implementarea cea mai performantă, însă nu există garanții asupra ordinii de parcurgere.
  - **TreeSet**: memorează elementele sub formă de arbore roșu-negru; elementele sunt ordonate pe baza valorilor, iar implementarea este mai lentă decât HashSet.
  - **LinkedHashSet**: este implementat ca o tabelă de dispersie, diferența față de HashSet este că LinkedHashSet menține o listă dublu-înlănțuită peste toate elementele sale; spre deosebire de HashSet, elementele rămân în ordinea în care au fost inserate; o parcurgere a LinkedHashSet va găsi elementele mereu în această ordine.

Colecții

- Un **Map** este un obiect care mapează chei pe valori. Într-o astfel de structură nu există chei duplicate. Fiecare cheie este mapată la exact o valoare. **Map** reprezintă o modelare a conceptului de funcție: primește o entitate ca parametru (cheia) și întoarce o altă entitate (valoarea).
- Subinterfața **SortedMap** menține perechile sale de tipul cheie-valoare în ordine crescătoare sau într-o anumită ordine specificată de un **Comparator**.
- Cele trei implementari pentru **Map** sunt:
  - **HashMap**
  - **TreeMap**
  - **LinkedHashMap**

Colecții

- Interfața **Queue** permite operații suplimentare de inserare, extragere și inspecție. Elementele dintr-o coadă sunt ordonate pe baza principiului FIFO.
- Implementări utilizate frecvent pentru **Queue**:
  - **LinkedList**: pe lângă List, clasa LinkedList implementează și interfața Queue
  - **PriorityQueue**;

Colecții

- Pentru a putea utiliza propriile obiecte în cadrul colecțiilor este nevoie de suprascrierea metodelor (din clasa *Object*):
  - **boolean equals(Object obj)**
  - **int hashCode()**
- Pentru sortarea obiectelor în cadrul colecției este nevoie de implementarea interfețelor:
  - **Comparable** (metoda **int compareTo(Object obj)**) sau
  - **Comparator** (metoda **int compare(Object one, Object two)**).

Colecții

- Spre deosebire de **Comparable** care, implementată de o clasă, marchează faptul ca instanțele sale sunt comparabile, **Comparator** desemnează o entitate externă care realizează o comparație între două obiecte oarecare.
- Intuitiv, faptul că o clasă implementează **Comparator** înseamnă că ea se comportă ca un comparator pentru obiecte de un anumit tip.

Colecții

Colecțiile pot fi parcurse (element cu element) folosind:

- **iteratori**;
- o construcție *for* specială, cunoscută sub denumirea de **for-each** sau **enhanced-for**.

Colecții

Iteratorii:

- Sunt obiecte utilizate pentru a gestiona poziția curentă în cadrul unei colecții;
- Un iterator este un obiect care permite traversarea unei colecții si modificarea acesteia (ex: ștergere de elemente) în mod selectiv;
- Clasele utilizate sunt **Interator** și **ListIterator**;
- Pentru a defini un iterator peste o colecție este nevoie de implementarea interfețelor **Iterable** sau **Iterator**.

Colecții

Algoritmi de prelucrare:

- Sunt algoritmii polimorfici care implementează diferite funcționalități oferite de platforma Java;
- Sunt metode statice în clasa *Collections*;
- În majoritatea cazurilor sunt utilizați cu instanțe de *List*.

Colecții

Algorithms:

Algorithm	Method	Description
Sorting	sort()	Sorts a List by natural order or by a Comparator
Shuffling	shuffle()	Shuffles a Collection
Searching	binarySearch()	Searches a sorted list for a given value
Composition	frequency() disjoint()	The frequency of a given value Number of common elements in 2 collections
Find extreme values	min() max()	

Colecții

Algorithms:

Algorithm	Method	Description
Routine Data Manipulation	reverse()	Reverses the order of the elements in a List
	fill()	Overwrites every element in a List with the specified value
	copy()	Copies a source List into a destination one
	swap()	Swaps the elements at the specified positions in a List
	addAll()	Adds elements to a Collection

Bibliografie

- [1] Jonathan Knudsen, Patrick Niemeyer – *Learning Java, 3rd Edition*, O'Reilly.
- [2] http://www.itcsolutions.eu
- [3] http://www.acs.ase.ro
- [4] http://docs.oracle.com/javase/tutorial/ /index.html
- [5] http://cursuri.cs.pub.ro/~poo/wiki/index.php/Colectii

Curs 5 PPOO

Conf. univ. dr. Cristian CIUREA

Departamentul de Informatică și Cibernetică Economică

cristian.ciurea@ie.ase.ro

Java fundamentals

- Genericitate în Java

Generics

- **Generics** permite unui tip de date sau unei metode să opereze pe obiecte de diferite tipuri, oferind în același timp siguranță la momentul compilării.
- Conceptul de **Generics** a fost încorporat în Java începând cu versiunea de JDK 5.0, împreună cu conceptul de **wildcards**.
- Se aseamănă cu conceptul de **template** din C++.

Generics

- **Generics** forțează siguranța la momentul compilării pe colecții sau alte clase și metode declarate utilizând parametrii de tipuri generice (verificarea tipului de date se face la compilare).
- Se utilizează sintaxa cu **<denumire tip>**.
- Se utilizează în majoritatea situațiilor pe colecții de obiecte.

Generics

```
public class Box {

Object value; //generic reference

public void setValue(Object value)    {this.value = value;}
public Object getValue()    {return value; }
}
```

A generic class – classic approach

VS.

```
public class GenericBox<T> {
T value; //generic type
public void setValue(T value) { this.value = value;}
public T getValue()    {return value;}
}
```

Generics

Convenții de denumire a tipurilor:

- **E - Element** (folosit în special de Java Collections Framework)
- **K - Key**
- **N - Number**
- **T - Type**
- **V - Value**

Generics

- Mecanismul genericității oferă un mijloc de abstractizare a tipurilor de date și este util mai ales în ierarhia de **colecții**.
- ```
  List lista= new LinkedList();
  lista.add(new Integer(0));
  Integer x = (Integer) lista.iterator().next();
  ```
- Se observă necesitatea operației de cast pentru a identifica corect variabila obținută din listă.
- Această situație are mai multe dezavantaje:
  - este îngreunată citirea codului;
  - apare posibilitatea unor erori la execuție, în momentul în care în listă se introduce un obiect care nu este de tipul *Integer*.
  ```

Generics

- Genericitatea intervine tocmai pentru a elimina aceste probleme.
- ```
  List<Integer> lista= new LinkedList<Integer>();
  lista.add(new Integer(0));
  Integer x = lista.iterator().next();
  ```
- În această situație, lista nu mai conține obiecte oarecare, ci poate conține doar obiecte de tipul *Integer*.
- Beneficiile dobândite prin utilizarea genericității constau în:
  - îmbunătățirea lizibilității codului;
  - creșterea gradului de robustețe.
  ```

Generics

- În ceea ce privește relația de moștenire, în cazul a două tipuri de date A și B (de exemplu, ***Number*** și ***Integer***), indiferent dacă A sau B sunt în relație de moștenire sau nu, ***MyClass<A>*** nu este în nici o relație cu ***MyClass<B>***.
- Părintele comun al claselor ***MyClass<A>*** și ***MyClass<B>*** este ***Object***.
- **Integer** extends **Object**
- **Integer** extends **Number**
- ~~**Stack<Integer> extends Stack<Object>**~~
- ~~**Stack<Integer> extends Stack<Number>**~~

Generics

- Java permite utilizarea de metacaractere pentru a servi ca argumente de tip pentru tipurile parametrizate. **Wildcard-urile** sunt argumente de tip de forma "?".
- Wildcard-urile sunt utilizate atunci când dorim să întrebuințăm o structură generică drept parametru într-o funcție și nu dorim să limităm tipul de date din colecția respectivă.

Generics

- Sintaxa unui wildcard permite unei metode generice să accepte subtipuri sau supertipuri ale tipului declarat.
- Wildcard-ul de forma "?" se numește **wildcard nemărginit** și arată că orice tip de instanțiere este posibilă.
- Ca un exemplu de wildcard nemărginit, **List <?>** indică o listă care are un tip de obiect necunoscut. Metodele care primesc o astfel de listă ca parametru vor accepta orice tip de obiect ca argument.

```
List<?> anyList = new ArrayList<Date>( );
anyList = new ArrayList<String>( );
```

Generics

- Sintaxa unui **wildcard mărginit** utilizează cuvintele cheie **extends** sau **super** pentru a limita domeniul de tipuri atribuite.

```
List<? extends Date> dateList = new ArrayList<Date>( );
dateList = new ArrayList<MyDate>( );
```

Generics

Wildcard-urile pot fi:

- delimitate superior: **List<? extends Number> list**
- delimitate inferior: **List<? super Integer> list**
- fără delimitare: **List<?> list**

Generics

- De reținut acronimul ***PECS*** (**Producer Extends, Consumer Super**).
- **Producer Extends:** dacă avem nevoie de o colecție List pentru a produce valori de tip **T** (dorim să citim obiecte de tip T din listă), trebuie să o declarăm **<? extends T>**, dar nu putem adăuga elemente la această listă.
- **Consumer Super:** dacă avem nevoie de o colecție List pentru a consuma valori de tip **T** (dorim să scriem obiecte de tip T în listă), trebuie să o declarăm **<? super T>**, dar nu există garanții referitor la ce tip de elemente putem citi din această listă.

Generics

Clase generice:

- Sunt clase template similar ca în C++, adică sunt șabloane de clase cu parametri;
- Pot fi adaptate pe tipuri reale, concrete (tipurile fundamentale din Java plus cele definite de utilizator);
- Prin instanțiere, JVM generează clase concrete;
- O clasă generică necesită unul sau mai multe tipuri parametrizate.

Generics

```
public class TestGenerics<T> {
        T instanceVariable;
        T[] array
        TestGenerics(T input){
            instanceVariable = input;
        }
        T getInstance(){
            return instanceVariable;
}

TestGenerics<int> ref = new TestGenerics<int>();
```

type variable

Generics

Metode generice:

- Sunt metode care introduc parametri proprii, alții decât cei ai clasei din care fac parte;
- Conțin declararea tipului parametrizat folosind sintaxa <> (operatorul ***diamond***);
- Permit creșterea gradului de generalizare prin definirea de șabloane de funcții;
- Sintaxa cu <> se pune înaintea tipului returnat al metodei;
- Spre deosebire de clasele generice, metodele generice nu trebuie să fie instanțiate cu un tip concret înainte de a fi utilizate.

```
<T> return_type method(parameters)
ex: <T> T doSomething(int val1, T val2){}
```

Bibliografie

- [1] Jonathan Knudsen, Patrick Niemeyer – *Learning Java, 3rd Edition*, O'Reilly.
- [2] http://www.itcsolutions.eu
- [3] http://www.acs.ase.ro
- [4] http://docs.oracle.com/javase/tutorial/ /index.html
- [5] http://cursuri.cs.pub.ro/~poo/wiki/index.php/Genericitatea
- [6] https://profs.info.uaic.ro/~acf/java/slides/ro/colectii_slide.pdf

Curs 6 PPOO

Conf. univ. dr. Cristian CIUREA

Departamentul de Informatică și Cibernetică Economică

cristian.ciurea@ie.ase.ro

Java fundamentals

- Fișiere – operații de I/O în Java
- Serializare

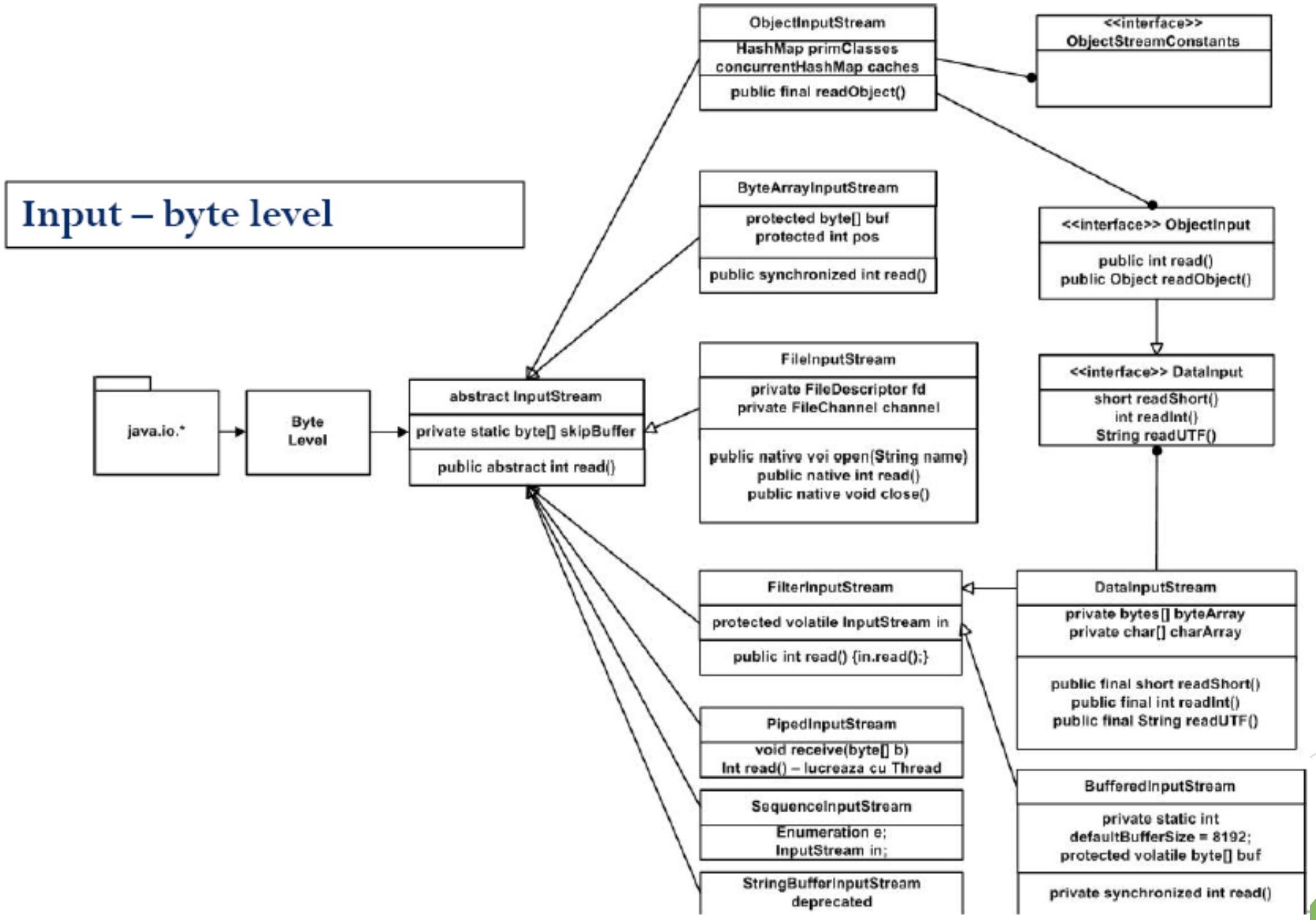
Fișiere

- Operațiile de intrare/ieșire sunt realizate, în general, cu ajutorul claselor din pachetul **java.io**.
- Bibliotecile I/O folosesc conceptul de **stream** (*flux*), ce reprezintă orice **sursă** sau **consumator** de date care este capabil să producă sau să primească **unități** de date, într-o manieră **secvențială**.

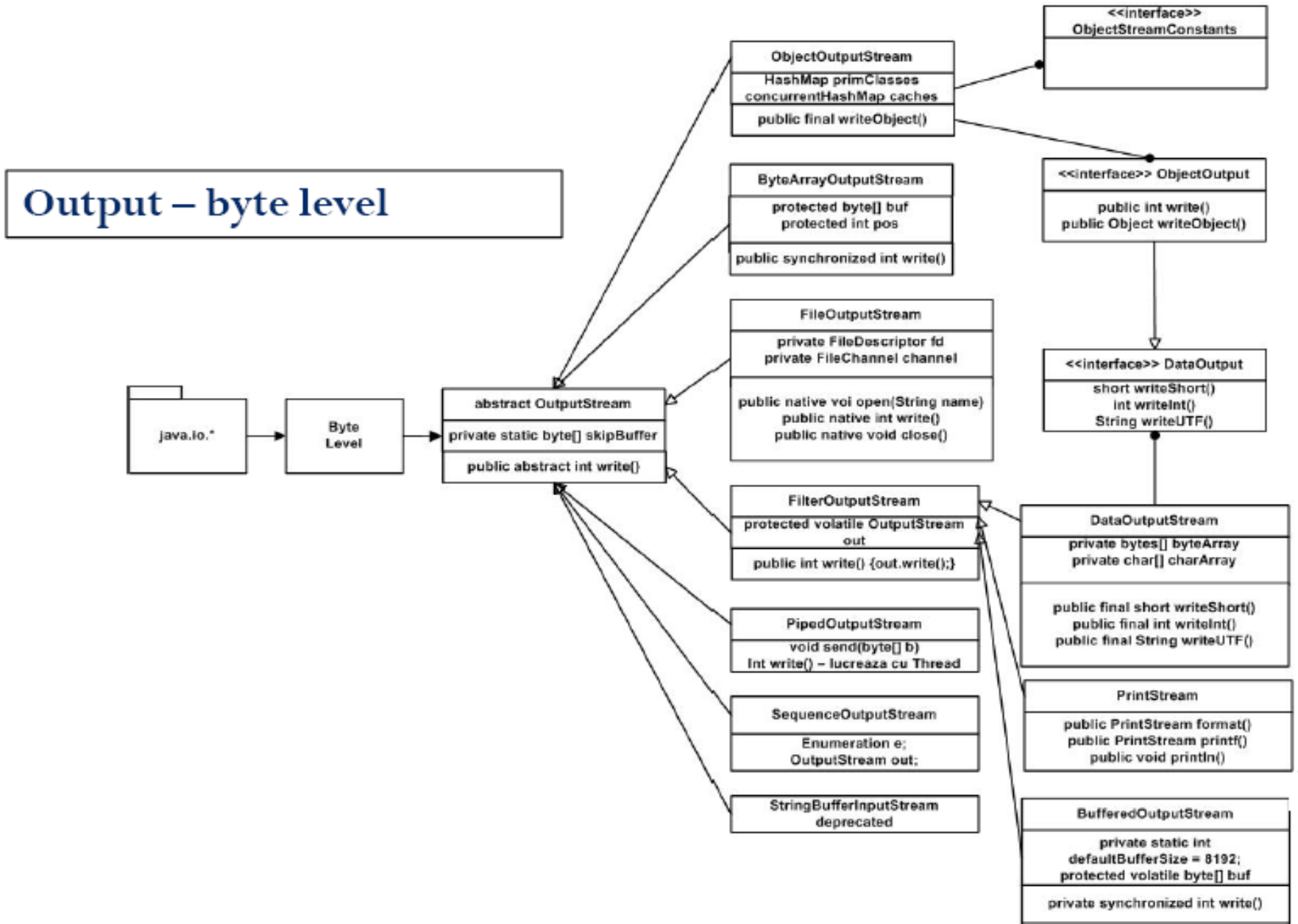
Fișiere

- Operațiile de I/O se bazează pe fluxuri de date:
  - la nivel de bait: **InputStream**, **OutputStream**;
  - la nivel de caracter (în Java un caracter = 2 B): **Reader**, **Writer**.
- Fișierele în Java sunt gestionate prin obiecte din clasa **File**.
- Majoritatea funcțiilor din aceste clase pot arunca excepții de tip **IOException** sau derivate din aceasta, cum este **FileNotFoundException**.

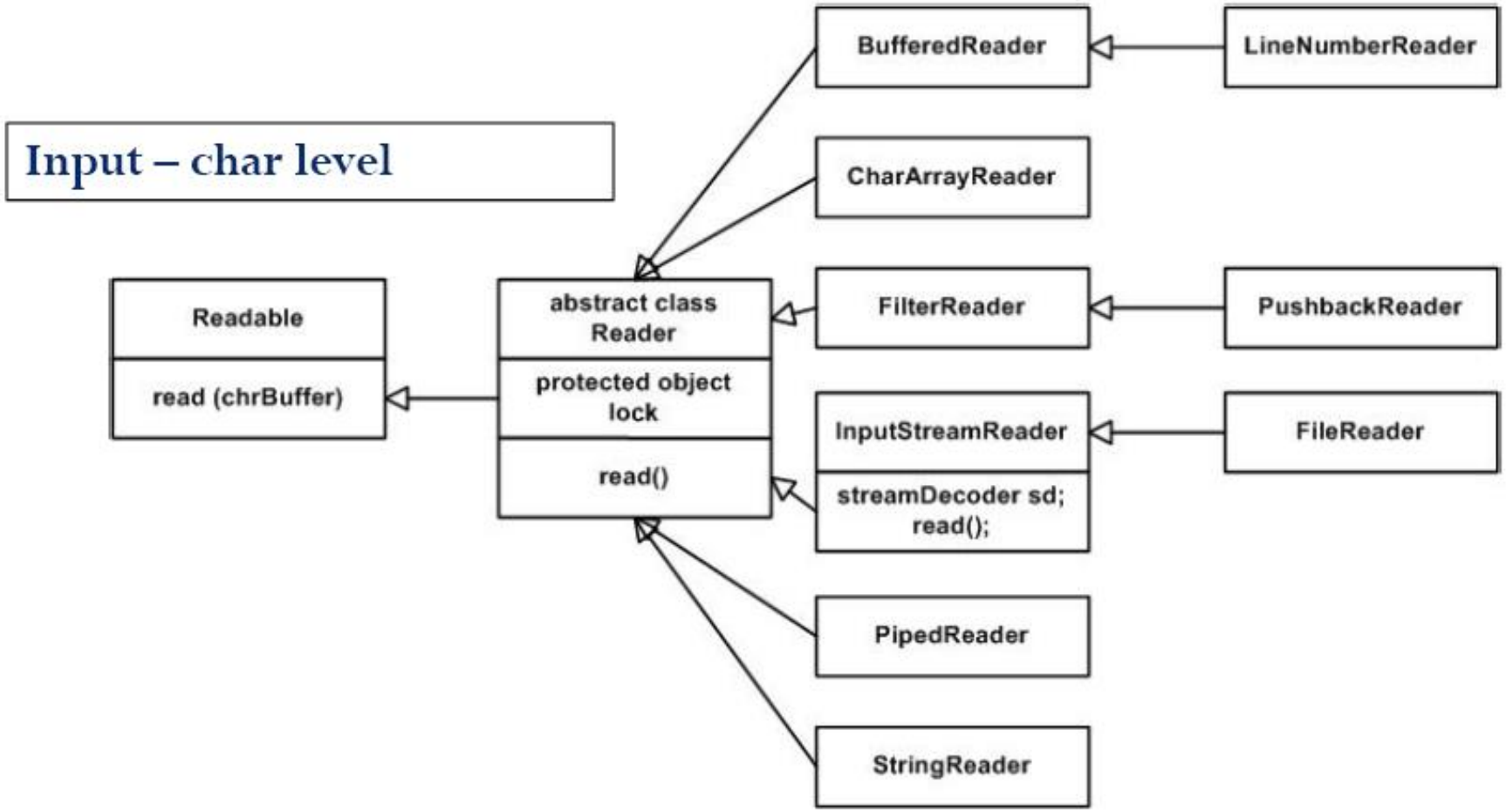
Fișiere


Input – byte level
java.io.*
Byte Level
abstract InputStream
private static byte[] skipBuffer
public abstract int read()
ObjectInputStream
HashMap primClasses
concurrentHashMap caches
public final readObject()
<<interface>> ObjectStreamConstants
ByteArrayInputStream
protected byte[] buf
protected int pos
public synchronized int read()
<<interface>> ObjectInput
public int read()
public Object readObject()
FileInputStream
private FileDescriptor fd
private FileChannel channel
public native void open(String name)
public native int read()
public native void close()
<<interface>> DataInput
short readShort()
int readInt()
String readUTF()
FilterInputStream
protected volatile InputStream in
public int read() {in.read();}
DataInputStream
private bytes[] byteArray
private char[] charArray
public final short readShort()
public final int readInt()
public final String readUTF()
PipedInputStream
void receive(byte[] b)
Int read() – lucreaza cu Thread
SequenceInputStream
Enumeration e;
InputStream in;
StringBufferInputStream
deprecated
BufferedInputStream
private static int
defaultBufferSize = 8192;
protected volatile byte[] buf
private synchronized int read()

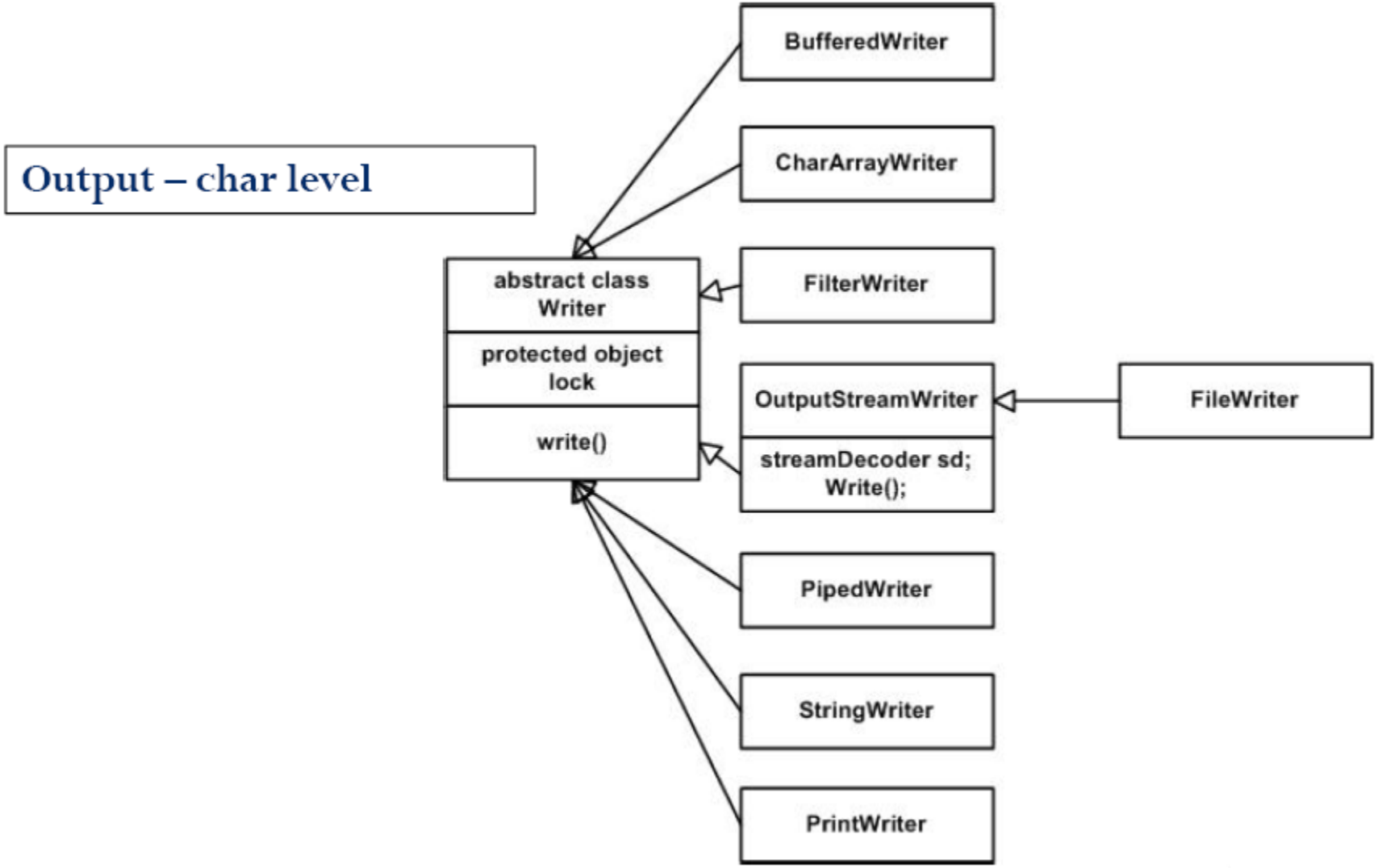
Fișiere


Output – byte level
java.io.*
Byte Level
abstract OutputStream
private static byte[] skipBuffer
public abstract int write()
ObjectOutputStream
HashMap primClasses
concurrentHashMap caches
public final writeObject()
<<interface>> ObjectStreamConstants
ByteArrayOutputStream
protected byte[] buf
protected int pos
public synchronized int write()
<<interface>> ObjectOutput
public int write()
public Object writeObject()
FileOutputStream
private FileDescriptor fd
private FileChannel channel
public native void open(String name)
public native int write()
public native void close()
<<interface>> DataOutput
short writeShort()
int writeInt()
String writeUTF()
FilterOutputStream
protected volatile OutputStream out
public int write() {out.write();}
DataOutputStream
private bytes[] byteArray
private char[] charArray
public final short writeShort()
public final int writeInt()
public final String writeUTF()
PipedOutputStream
void send(byte[] b)
Int write() – lucreaza cu Thread
PrintStream
public PrintStream format()
public PrintStream printf()
public void println()
SequenceOutputStream
Enumeration e;
OutputStream out;
StringBufferInputStream
deprecated
BufferedOutputStream
private static int defaultBufferSize = 8192;
protected volatile byte[] buf
private synchronized int write()

Fișiere


Input – char level
Readable
read (chrBuffer)
abstract class Reader
protected object lock
read()
BufferedReader
LineNumberReader
CharArrayReader
FilterReader
PushbackReader
InputStreamReader
streamDecoder sd; read();
FileReader
PipedReader
StringReader

Fișiere


Output – char level
abstract class Writer
protected object lock
write()
BufferedWriter
CharArrayWriter
FilterWriter
OutputStreamWriter
streamDecoder sd;
Write();
FileWriter
PipedWriter
StringWriter
PrintWriter

Fișiere

- **InputStreamReader**, **OutputStreamWriter** - convert bytes to characters and vice versa
- **DataInputStream**, **DataOutputStream** - read and write simple data types
- **ObjectInputStream**, **ObjectOutputStream** – read and write serialized Java objects
- **BufferedInputStream**,**BufferedOutputStream**, **BufferedReader**, **BufferedWriter** - stream filters with buffering

Fișiere

- **PrintStream**, **PrintWriter** – prints text;
- **FileInputStream**, **FileOutputStream** - implementations of InputStream, OutputStream
- **FileReader**, **FileWriter** - implementations of Reader, and Writer

Fișiere

- Există conversii de la șiruri de octeți la șiruri de caractere. Filtrul ce realizează această conversie este **InputStreamReader**, care obține perspectiva caracter asupra unui stream octet.
- **InputStreamReader** folosește o codificare dată ca parametru în constructor sau folosește codificarea implicită.
- O aplicație foarte importantă a acestui mecanism o reprezintă citirea de la consolă, ținând cont de faptul ca **System.in** are tipul **InputStream**. Deși era mai firesc ca tipul sa fie **Reader** (orientat caracter), acesta a rămas orientat octet din motive de compatibilitate: în versiunea inițială nu existau stream-uri caracter.

Fișiere

- Stream-urile au limitarea de a oferi o perspectivă exclusiv **secvențială** asupra unei surse/destinații de date.
- La un moment dat, nu putem sări direct la un octet/caracter din flux, ci trebuie să consumăm unitatea imediat următoare.
- Anumite stream-uri, cum ar fi pipe-urile sau socket-ii, nu pot fi abordate decât secvențial. Pentru fișiere situația stă altfel: ele reprezintă obiecte care au asociat un **cursor**, ce indică poziția următoarei operații de citire/scriere și care poate fi deplasat (***seek***).

Fișiere

Clasa **File** permite manipularea fișierelor ca entități, oferind o reprezentare abstractă a căilor de fișiere și directoare. Aceasta oferă operații precum:

- **`boolean createNewFile()`** – creează un nou fișier la calea specificată în constructor;
- **`boolean delete()`** – șterge fișierul;
- **`boolean exists()`** – verifică dacă fișierul/directorul dat de cale există;
- **`boolean isDirectory()`** – întoarce true atunci când calea denotă un director;
- **`boolean isFile()`** - întoarce true atunci când calea denotă un fisier;
- **`long length()`** – întoarce dimensiunea fisierului;
- **`String[] list()`** – întoarce intrările din director.

Fișiere

Class	Extends	Constructor	Methods
File	Object	(String) (String,String) (File,String)	createNewFile() delete() exists() isDirectory() isFile() list() mkdir() renameTo()
FileWriter	Writer	(File) (String)	cose() flush() write()
BufferedWriter	Writer	(Writer)	close() flush() newLine() write()

Fișiere

Class	Extends	Constructor	Methods
PrintWriter	Writer	(File) (String) (OutputStream) (Writer)	close() flush() write() print() println() format()
FileReader	Reader	(File) (String)	read()
BufferedReader	Reader	(Reader)	read() readLine()

Fișiere

Stream-urile standard, pe care le posedă orice program, sunt:

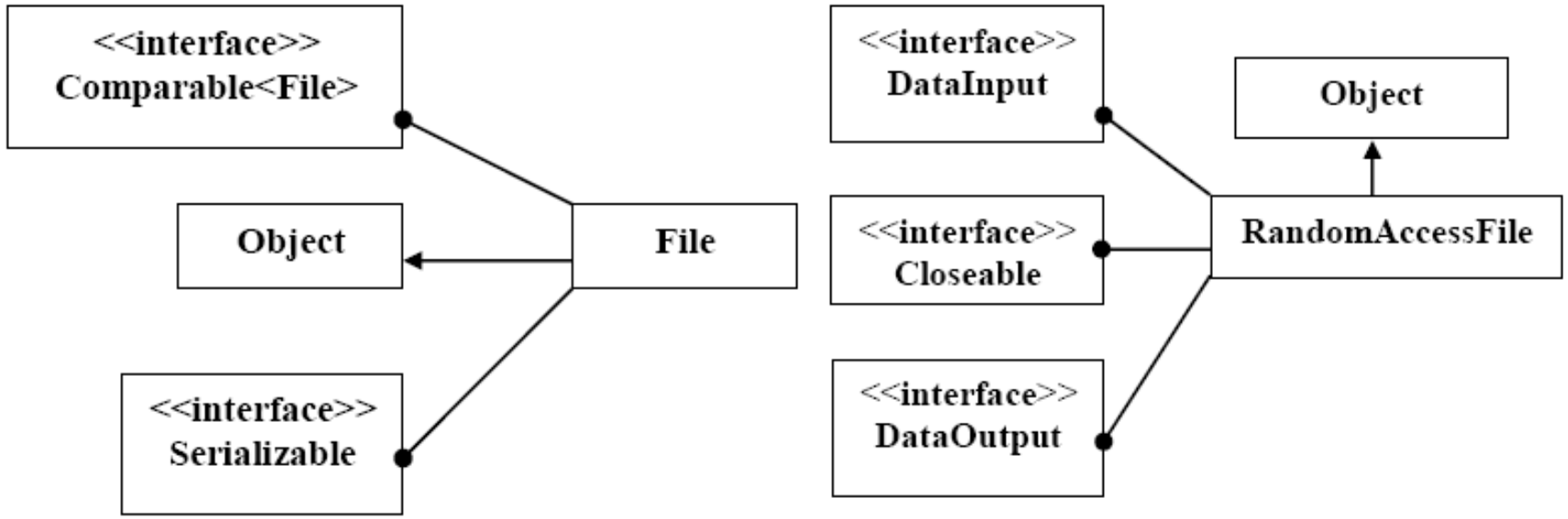
- **System.in**, de tip **InputStream;**
- **System.out**, de tip **PrintStream;**
- **System.err**, de tip **PrintStream**.

Clasa **PrintStream** oferă posibilitatea de formatare ca șir de caractere a valorilor având tipuri primitive. În general, nu este foarte folosită.

Fișiere

- Spre deosebire de clasele menționate anterior, care gestionează **conținutul** fișierelor, clasa **File** permite manipularea fișierelor ca **entități**, oferind o reprezentare abstractă a căilor de fișiere și directoare.
- Clasa **RandomAccessFile** nu aparține niciunei ierarhii de clase menționate anterior. Ea oferă posibilitatea deplasării prin fișierul deschis, folosind **seek** și implementează funcționalitatea **DataInput** și **DataOutput** (cea implementată și în **DataInputStream**, **DataOutputStream**), pentru a putea lucra cu tipuri primitive. Este asemănătoare ca funcționalitate cu folosirea fișierelor în ANSI C (fseek, fread, fwrite, etc).

Fișiere



File & RandomAccessFile

Serializare

- Serializarea presupune salvarea stării unui obiect;
- Deserializarea presupune încărcarea/restaurarea stării unui obiect;
- Operațiile sunt realizate prin apelul metodelor **ObjectInputStream.writeObject()**, respectiv **ObjectOutputStream.readObject()**;
- Clasa ale cărei obiecte se serializează trebuie să implementeze interfața **Serializable**;
- Atributele marcate ca **transient** nu sunt serializate.

Serializare

- you can override the mechanism:

```
private void writeObject(ObjectOutputStream os){
os.defaultWriteObject();
//other data
}
```

Serializare

- În cazul unei clase serializabile derivată dintr-o clasă non-serializabilă, toate variabilele instanță moștenite vor fi resetate la valorile primite în constructorul clasei derivate;
- Variabilele statice nu sunt serializate;
- Serializarea utilizează o valoare hash pe 64 biți, denumită **Serial Version UID (SUID)**, pentru a stoca versiunea structurii clasei:
- **static final long serialVersionUID = 1L;**

Serializare

- Fiecare clasă serializabilă are un număr unic de identificare asociat cu aceasta (**SUID**).
- În cazul în care numărul de identificare nu este specificat în mod explicit prin declararea unui câmp **private static final long**, denumit **serialVersionUID**, sistemul îl generează în mod automat, prin aplicarea unei proceduri deterministe complexe asupra clasei.
- În cazul în care clasa este modificată în vreun fel, câmpul **serialVersionUID** generat automat se modifică.

Serializare

- Cu excepția cazului în care programatorul dorește să definească explicit o schemă particulară de serializare, sistemul poate asigura serializarea automată a întregii componențe a obiectelor, explorând recursiv structura acestuia.
- Graful de referințe obținut poartă denumirea de **web of objects**.
- Serializarea este utilizată pentru implementarea conceptului de **persistență**, văzută drept capacitatea unui obiect de a supraviețui după încheierea execuției programului.

Bibliografie

- [1] Jonathan Knudsen, Patrick Niemeyer – *Learning Java, 3rd Edition*, O'Reilly.
- [2] http://www.itcsolutions.eu
- [3] http://www.acs.ase.ro
- [4] http://docs.oracle.com/javase/tutorial/ /index.html
- [5] http://cursuri.cs.pub.ro/~poo/wiki/index.php/Input/Output
- [6] http://cursuri.cs.pub.ro/~poo/wiki/index.php/Serializare

Curs 7 PPOO

Conf. univ. dr. Cristian CIUREA

Departamentul de Informatică și Cibernetică Economică

cristian.ciurea@ie.ase.ro

Java fundamentals

- Singleton
- Callback

Singleton

- **Singleton** este un model de proiectare (**design pattern**) care restricționează instanțierea unei clase la un singur obiect. Acest lucru este util atunci când este nevoie de exact un obiect pentru a coordona acțiunile în cadrul sistemului.
- Un design pattern este o soluție generală și reutilizabilă a unei probleme comune în proiectarea aplicațiilor software.
- Un design pattern nu este un design în forma finală, ceea ce înseamnă că nu poate fi transformat direct în cod. Acesta este o descriere a soluției sau un template ce poate fi aplicat pentru rezolvarea problemei.

Singleton

- Design pattern-urile au fost inițial grupate în următoarele categorii: *Creational patterns*, *Structural patterns*, și *Behavioral patterns*.
- *Creational Patterns* sunt pattern-uri ce implementează mecanisme de creare a obiectelor. În această categorie se încadreaza pattern-urile **Singleton** si **Factory**.
- *Structural Patterns* sunt pattern-uri ce simplifică design-ul aplicației prin găsirea unei metode de a defini relațiile dintre entități. În această categorie se încadrează pattern-ul **Decorator**.
- *Behavioral Patterns* sunt pattern-uri ce definesc modul în care obiectele comunică între ele. În această categorie se încadrează pattern-urile **Command**, **Visitor** și **Observer**.

Singleton

- **Singleton** asigură faptul că o clasă are o singură instanță și oferă un punct global de acces la aceasta.
- Se dorește **crearea unei singure instanțe** pentru o clasă prin care sa fie gestionată o resursă într-un mod **centralizat**.
- *Scenarii:*
  - Conexiune unică la baza de date
  - Gestiune unică fișiere
  - Gestiune unică conexiune la rețea
  - Gestiune unică preferințe pe platforma Android (SharedPreferences)
  - Gestiune centralizată a accesului la anumite resurse utilizate de soluție
  - Gestiune unică a unor obiecte costisitoare

Singleton

- **Factory** definește o interfață pentru crearea unui obiect, dar lasă subclasele să decidă ce clasă să instanțieze.
- Implementarea unui mecanism centralizat prin care **crearea** obiectelor este **transparentă** pentru client.
- *Scenarii:* generator de documente, generator de caractere într-un joc (caracter negativ, caracter pozitiv).

Singleton

- **Decorator** este un model de proiectare care permite adăugarea unui comportament unui singur obiect, într-o manieră statică sau dinamică, fără a afecta comportamentul celorlalte obiecte din aceeași clasă.
- Decorarea sau extinderea statică sau la run-time a functionalității sau stării unor obiecte.
- *Scenariu:* copil în costum de Batman => devine Batman.

Singleton

- **Command** este un model de proiectare comportamental în care un obiect este utilizat pentru a încapsula toate informațiile necesare pentru a efectua o acțiune sau a declanșa un eveniment ulterior.
- Aceste informații includ denumirea metodei, obiectul care deține metoda și valorile pentru parametrii metodei.

Singleton

- **Visitor** este un model de proiectare care descrie o modalitate de a separa un algoritm de structura unui obiect pe care operează.
- Un rezultat practic al acestei separări este capacitatea de a adăuga noi operații la structurile obiectului existent fără modificarea structurilor.
- În esență, **Visitor** permite adăugarea de noi funcții virtuale unei familii de clase, fără a modifica clasele.

Singleton

- **Observer** este un model de proiectare software în care un obiect denumit subiect păstrează o listă a dependenților săi, denumiți observatori, pe care îi notifică automat despre orice schimbări de stare, de obicei apelând una dintre metodele acestora.

Singleton

- Modelul de Singleton trebuie să fie construit cu atenție în aplicațiile care utilizează mai multe fire de execuție (**multi-threading**).
- Soluția clasică a acestei probleme este de a utiliza excluderea reciprocă (**mutual exclusion**) pe clasa care indică faptul că obiectul este instanțiat.

Singleton

- Scopul **Singleton** este de a controla crearea de obiecte din cadrul unei clase, limitând numărul la unu, dar în același timp să permită flexibilitatea de a crea mai multe obiecte în cazul în care situația se schimbă.
- Deoarece există doar o singură instanță **Singleton**, orice câmpuri ale unei instanțe de **Singleton** vor apărea o singură dată pentru fiecare clasă, similar cu atributele statice. **Singleton** deseori controlează accesul la resurse, cum ar fi conexiunile de baze de date sau socketi.

Singleton

Avantaj utilizare **Singleton**:

- este preferat variabilelor globale, deoarece nu încarcă namespace-ul global cu variabile care nu sunt necesare și permite inițializarea întârziată pentru a nu consuma inutil resursele sistemului.

Dezavantaj utilizare **Singleton**:

- îngreunează testarea aplicației, deoarece introduce stări globale.

Singleton

- La baza design pattern-ului **Singleton** stă o metodă ce permite crearea unei noi instanțe a clasei dacă aceasta nu există deja.
- Dacă instanța există deja, atunci întoarce o referință către acel obiect. Pentru a asigura o singură instanțiere a clasei, constructorul trebuie făcut **private** (un constructor privat împiedică reutilizarea sa sau accesul unei unități de testare).

Singleton

```
public class Singleton {
   private static final Singleton INSTANCE = new Singleton();

   // private constructor prevents instantiation from other classes
   private Singleton() { }

   public static Singleton getInstance() {
      return INSTANCE;
   }
}
```

Singleton

- Prin declararea unui constructor privat, utilizatorul clasei nu va putea să genereze o instanță nouă din această clasă, el fiind limitat la utilizarea singurei instanțe disponibile.
- Constructorul nu poate fi apelat decât din interiorul clasei, iar tentativa de a crea un obiect de tipul **Singleton** va genera eroare de compilator, deoarece se încearcă accesarea unui membru privat al clasei din afara ei.
- Metoda ce furnizează unica instanță a clasei se declară statică, deoarece dacă nu ar fi statică, ar fi necesară existența unei instanțe a clasei pentru ca metoda să poată fi apelată.

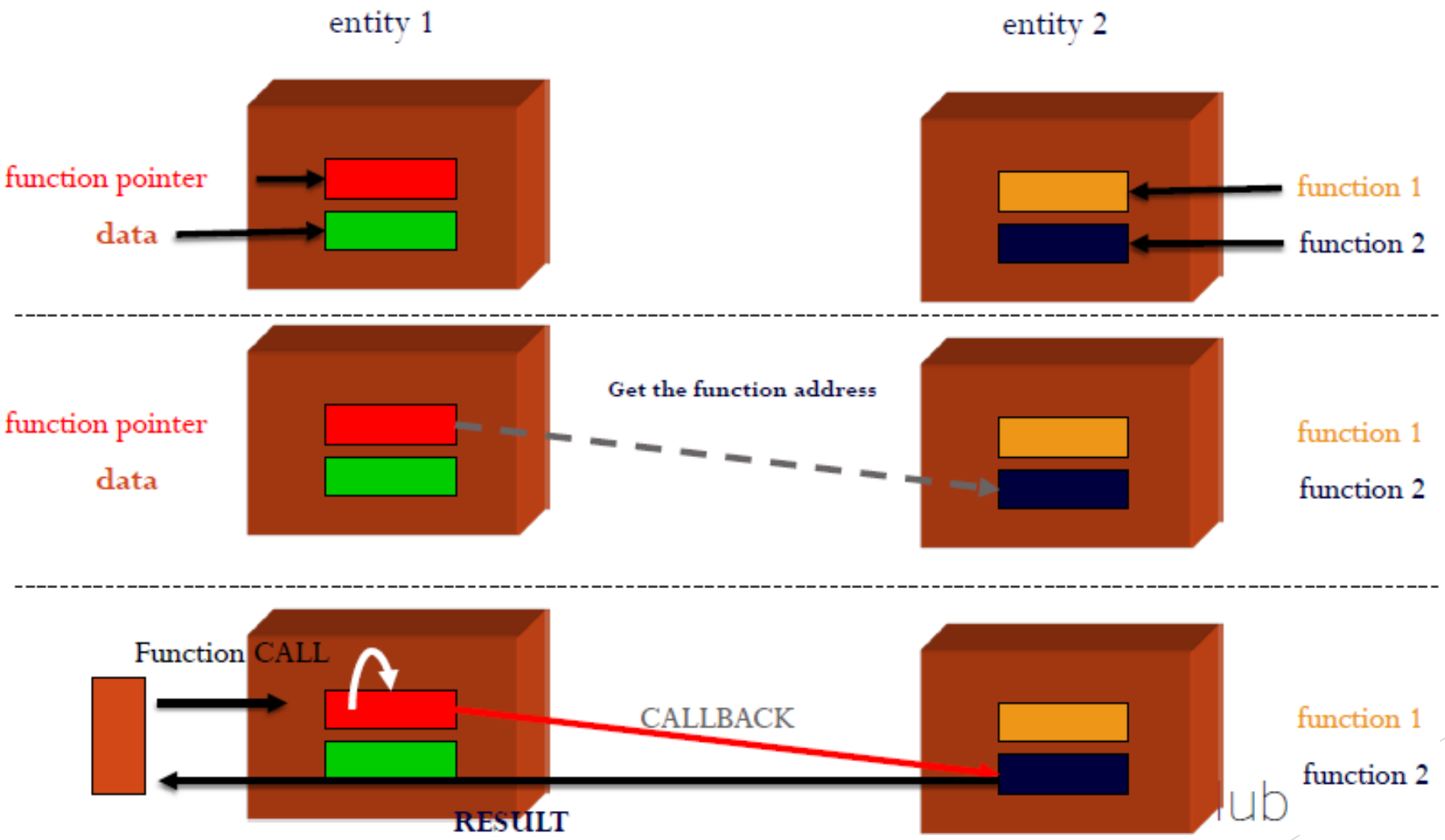
Callback

- **Callback** reprezintă o bucată de cod executabil care este transmis ca argument unui alt cod, care se așteaptă să-l apeleze/execute înapoi la un moment dat.
- Invocarea poate fi imediată, ceea ce înseamnă un apel invers **sincron**, sau s-ar putea întâmpla mai târziu, adică un apel invers **asincron**.

Callback

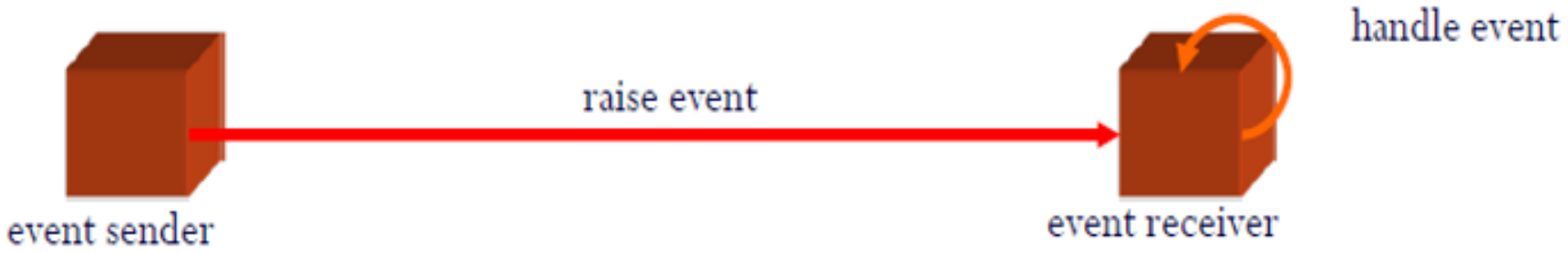
- Un callback (apel invers) este o secvență de cod care va fi transmisă unei anumite metode, astfel încât să poată fi apelată la un moment ulterior.
- Callback-ul este frecvent utilizat în programarea asincronă. În Java, funcțiile cu apel invers (callback) sunt utilizate în principal pentru a implementa un "Observer Pattern", care este strâns legat de programarea asincronă.
- În Java acest lucru poate fi în mod obișnuit realizat prin implementarea unei interfețe și transmiterea unui obiect care o implementează.

Callback


entity 1
entity 2
function pointer
data
function 1
function 2
function pointer
data
Get the function address
function 1
function 2
Function CALL
CALLBACK
function 1
function 2
RESULT

Callback

- Un **eveniment** este un mesaj transmis de un obiect pentru a anunța ceva (la nivelul interfeței grafice a unei aplicații – click de mouse, click de buton sau la nivel de logică a programului – rutine de aplicații).
- În modelul de gestionare a evenimentelor, obiectul care gestionează evenimentul nu cunoaște ce metodă va primi și va gestiona; de aceea este necesar ceva care să conecteze sursa și destinația.



Callback

- În Java nu există pointeri la funcții (ca în C/C++);
- În Java nu există tipul delegate sau evenimente ca în .NET;
- Avem la dispoziție doar INTERFEȚE.

Callback

- În C/C++ este posibil să declarăm pointeri la funcții ca argumente la alte funcții, însă în Java nu există pointeri.
- În Java, doar obiecte și tipuri de date primitive pot fi transmise ca parametri în cadrul metodelor unei clase.
- Suportul Java privind interfețele furnizează un mecanism prin care putem obține echivalentul callback-ului. Este necesar să se declare o interfață care declară la rândul ei funcția care se dorește a fi transmisă.

Callback

- Un exemplu de implementare a mecanismului de apel invers în Java (callback) este dat de funcția **Collections.sort(List list, Comparator c)**
- În acest caz, **c** este o instanță a unei clase care implementează metoda **compare(e1, e2)** în interfața **Comparator**.
- Se sortează lista specificată în funcție de ordinea indusă de comparatorul specificat. Toate elementele din listă trebuie să fie reciproc comparabile folosind un comparator specificat.

Callback

```
class CodedString implements Comparable<CodedString> {
    private int code;
    private String text;
    ...
    @Override
        public boolean compareTo(CodedString cs) {
            // Compare using "code" first, then
            // "text" if both codes are equal.
        }
}

...

public void sortCodedStringsByText(List<CodedString> codedStrings) {
    Comparator<CodedString> comparatorByText = new Comparator<CodedString>() {
        @Override
        public int compare(CodedString cs1, CodedString cs2) {
            // Compare cs1 and cs2 using just the "text" field
        }
    }
Collections.sort(codedStrings, comparatorByText);
}
```

Callback

- **Collections.sort(...)** va apela acest apel invers (**comparatorByText**) ori de câte ori trebuie să compare două elemente din lista care este sortată.
- Ca rezultat, vom obține lista sortată doar după câmpul "text". Dacă nu transmitem un apel invers, **Collections.sort()** va utiliza comparația implicită pentru clasă (mai întâi după câmpul "cod", apoi după câmpul "text").

Callback

Comparable vs. **Comparator**

- Interfața **Comparable** poate fi utilizată pentru a oferi un singur mod de sortare, în timp ce interfața **Comparator** este utilizată pentru a oferi diferite moduri de sortare.
- Pentru a utiliza **Comparable**, clasa trebuie să o implementeze, în timp ce pentru a utiliza **Comparator** nu este necesar să facem nicio schimbare în clasă.
- Interfața **Comparable** se afla în pachetul java.lang, în timp ce interfața **Comparator** este prezentă în pachetul java.util.
- Nu este necesar să facem modificări in cod pentru a utiliza **Comparable**, deoarece **Arrays.sort()** sau **Collection.sort()** utilizează automat metoda **compareTo()** a clasei. Pentru **Comparator**, este nevoie să se furnizeze clasa **Comparator** pentru a o utiliza în metoda **compare()**.

Bibliografie

- [1] Jonathan Knudsen, Patrick Niemeyer – *Learning Java, 3rd Edition*, O'Reilly.
- [2] http://www.itcsolutions.eu
- [3] http://www.acs.ase.ro
- [4] http://en.wikipedia.org/wiki/Singleton_pattern
- [5] http://en.wikipedia.org/wiki/Callback_(computer_programming)
- [6] http://cursuri.cs.pub.ro/~poo/wiki/index.php/Design_Patterns_Basics

Curs 8 PPOO

Conf. univ. dr. Cristian CIUREA

Departamentul de Informatică și Cibernetică Economică

cristian.ciurea@ie.ase.ro

Java fundamentals

- Fire de execuție
- Procese

Fire de execuție

- Simultaneitatea execuției programelor în Java este abilitatea de a rula mai multe programe sau mai multe părți ale unui program în paralel.
- În cazul în care o activitate consumatoare de timp poate fi realizată în mod asincron sau în paralel, aceasta va îmbunătăți obținerea rezultatelor și interactivitatea programului.
- Un calculator modern dispune de mai multe nuclee la nivelul procesorului, ceea ce înseamnă că abilitatea de a mobiliza aceste multi-core-uri poate fi cheia pentru execuția cu succes a unei aplicații cu volum mare de date.

Fire de execuție

- “**Multithreading**” înseamnă capacitatea unui program de a executa mai multe secvențe de cod în același timp.
- O astfel de secvență de cod se numește fir de execuție sau **thread**.
- Limbajul Java suportă multithreading prin clase disponibile în pachetul **java.lang**.

Fire de execuție

- Un **fir de execuție (thread)** este un flux de control în cadrul unui program.
- Un **proces** poate avea mai multe fire de execuție.

Solution 1 – extending Thread	Solution 2 – implementing Runnable

class **NewThread** extends **Thread** {
public void run() {…}
}
	class **NewThread** implements **Runnable** {
	public void run() {…}
	}

Fire de execuție

Instantiating threads

Solution 1 – extending Thread	Solution 2 – implementing Runnable
NewThread f = new **NewThread** ();	**NewThread** obf = new **NewThread** (); Thread f = new Thread (obf);

Executing threads

Solution 1 – extending Thread	Solution 2 – implementing Runnable
f.start();	f.start();

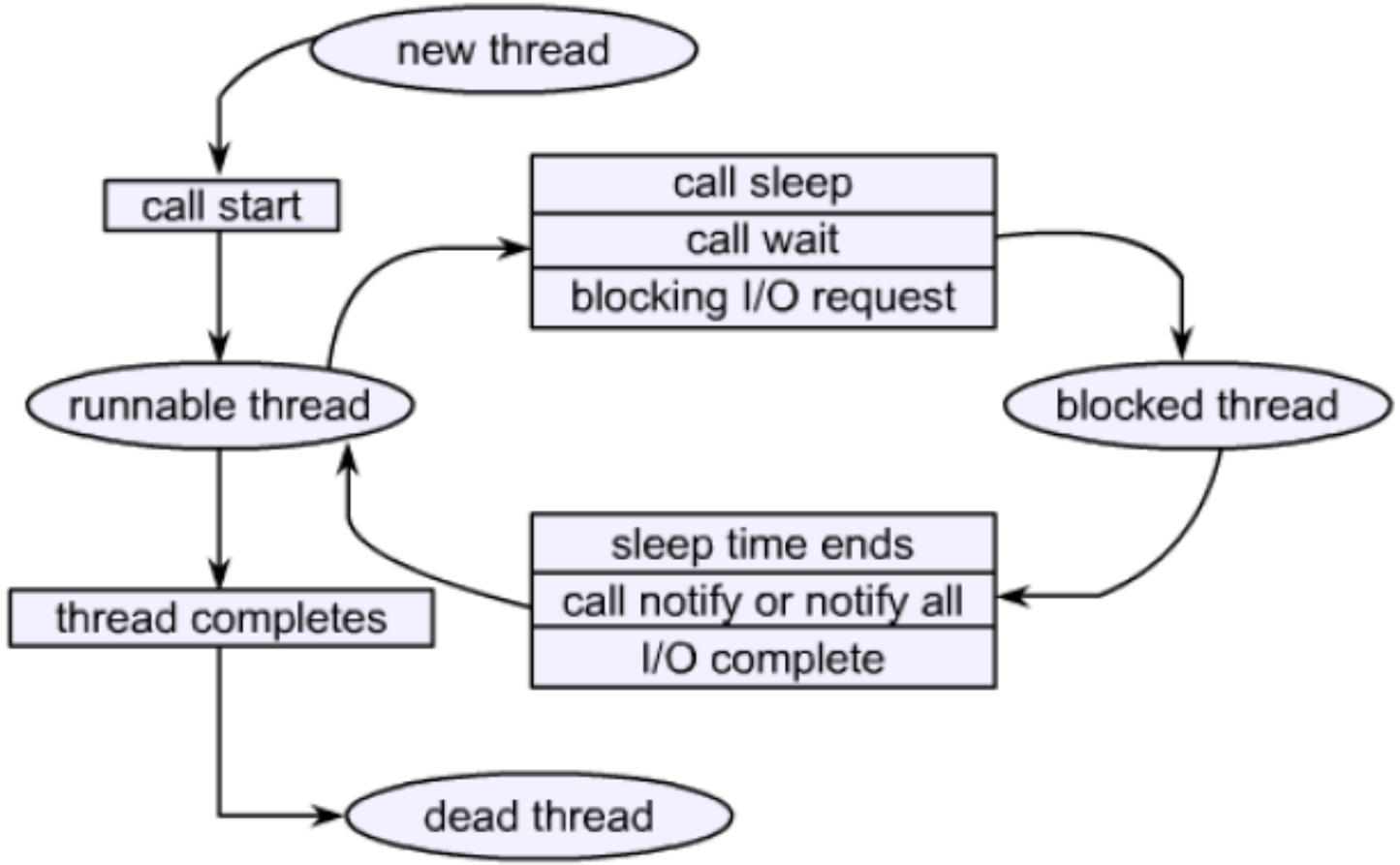
Fire de execuție

Controlling threads

Solution 1 – extending Thread	Solution 2 – implementing Runnable

public void run(){
this.sleep();
}
	public void run(){
	Thread t = Thread.currentThread();
	…
	t.sleep();
	}

Fire de execuție



Fire de execuție

- **new thread**: atunci când se instanțiază cu *new* un nou fir de execuție;
- **runnable thread**: după apelul metodei **start()**;
- **blocked thread**: dacă se apelează **sleep()** sau **wait()**;
- **dead thread**: după ce metoda **run()** s-a terminat.

Fire de execuție

- Există două tipuri de fire de executie într-o aplicație - **user threads** și **daemon threads**. Când pornim o aplicație, main() este primul fir de executie de tip utilizator creat și putem crea mai multe fire de tip utilizator, precum și fire de tip daemon. Când toate firele de executie de tip utilizator sunt executate, JVM încheie programul.

Fire de execuție

- Putem seta priorități diferite pentru diferite fire de executie, dar nu se garantează faptul că thread-ul cu prioritate mai mare se va executa mai întâi față de altul cu prioritate inferioară.
- Planificatorul de fire de execuție este o parte a sistemului de operare și atunci când este pornit un fir, execuția este controlată de **Thread Scheduler**, iar JVM nu are niciun control asupra execuției acestuia.

Fire de execuție

- Un fir de execuție de tip **daemon** este un fir care nu împiedică ieșirea din program când acesta se termină, ci firul de execuție continuă să ruleze. Un exemplu de fir de execuție de tip daemon este **garbage collector**.
- Se poate apela metoda **setDaemon(boolean)** pentru a modifica proprietățile unui fir de execuție înainte ca firul să pornească.

Fire de execuție

- Metoda **main()** are propriul fir de execuție. Prin apelul **start()** se cere JVM crearea și pornirea unui nou fir de execuție. Din funcția **start()** se va ieși imediat. Firul de execuție corespunzător metodei **main()** își va continua execuția independent de noul fir de execuție creat.
- Implementările Java depind de platformă/sistem de operare. Un program Java care folosește fire de execuție poate avea comportamente diferite la execuții diferite pentru aceleași date de intrare.

Fire de execuție

Sincronizarea firelor de execuție:

- **concurență**: utilizarea aceloraşi resurse;
- **cooperare**: interschimb de date.

Controlul firelor de execuție:

- metodele **wait()**, **notify()** și **notifyAll()** moștenite din clasa *Object*;
- metoda **join()** în clasa *Thread*;
- metoda **setPriority()** în clasa *Thread*;
- metodele statice **yield()** și **sleep()** în clasa *Thread*.

Fire de execuție

Controlul firelor de execuție:

- metoda **setPriority()** permite setarea unei priorități de la 1 la 5 sau de la 1 la 10 pentru firul curent de execuție;
- metoda **join()** permite ca firul curent de execuție să aștepte alt fir de execuție să se termine (firul curent de execuție se unește cu celălalt);
- metoda **yield()** întrerupe firul curent de execuție și oferă controlul unui alt fir de execuție având aceeași prioritate;
- metoda **sleep()** întrerupe firul curent de execuție pentru o perioadă de timp.

Fire de execuție

- Java implementează mecanismul de excludere mutuală (**mutual exclusion**) prin specificarea metodelor care partajează variabile ca fiind **synchronized**.
- Orice fir de execuție care încearcă să acceseze o metodă **synchronized** a unui obiect, atât timp cât metoda este utilizată de un alt fir de execuție, este blocat până când primul fir de execuție părăsește metoda.

Fire de execuție

- Tot o problemă de excludere mutuală este exemplul producător-consumator. Producătorul furnizează date pe care consumatorul le utilizează mai departe. Dacă vitezele de producere și consum diferă se utilizează metodele **wait(), notify()** și **notifyAll()** ale clasei **Object**.
- Apelul lui **wait()** va trece obiectul apelat în starea Blocked. El rămâne blocat până la apelul unei metode **notify()** sau **notifyAll()** pentru același obiect. Condiția necesară pentru a se apela una dintre aceste metode este ca apelul lor să se facă în interiorul metodelor **synchronized**.
- Pentru metoda **wait()** se poate specifica o durată maximă de așteptare. În acest caz, firul de execuție rămâne blocat până când timpul expiră sau un alt fir de execuție apelează **notify()**.

Fire de execuție

Concurența / simultaneitatea:

- permite ca metodele **sincronizate** să acceseze resurse comune;
- fiecare obiect are o singură blocare (gestionată de JVM);
- variabilele comune sunt declarate ca **volatile** (modificările sunt șterse din memorie);
- utilizează metodele **wait()**, **notify()** și **notifyAll()** moștenite din clasa Object.

Fire de execuție

Concurența / simultaneitatea:

- Urmărește firele de execuție blocate;
- metodele **wait()**, **notify()** și **notifyAll()** trebuie să fie apelate dintr-un context sincronizat;
- implementarea specifică a metodei **wait()**:

```
while (!condition) {
    this.wait();
}
```

Procese

- Un proces rulează independent și izolat de alte procese. El nu poate avea acces direct la datele partajate din alte procese. Resursele procesului, de exemplu, memorie și timp CPU, sunt alocate acestuia prin intermediul sistemului de operare.
- Un fir de execuție este un proces denumit “ușor” (**lightweight**). El are propria sa stivă de apel, dar poate avea acces la date partajate ale altor fire de execuție în cadrul aceluiași proces. Fiecare fir are propria sa memorie cache.

Procese

- Un flux de ieșire **PipedOutputStream** poate fi conectat la un flux de intrare **PipedInputStream** pentru a crea o conductă/țeavă de comunicație. În mod obișnuit, datele sunt scrise într-un obiect **PipedOutputStream** prin intermediul unui fir de execuție și sunt citite dintr-un **PipedInputStream** conectat printr-un alt fir de execuție. Încercarea de a utiliza ambele obiecte dintr-un singur fir de execuție nu este recomandată, deoarece ar putea bloca firul. Se consideră că țeava este ruptă dacă un fir de execuție care citea octeți de date din fluxul de intrare conectat nu mai este în viață.

Procese

- În cazul în care un fir de execuție citește date partajate, el stochează aceste date în propria sa memorie cache. Un fir de execuție poate reciti datele partajate.
- O aplicație Java rulează în mod implicit într-un singur proces. Într-o aplicație Java se lucrează cu mai multe fire de execuție pentru a realiza **procesarea paralelă** sau **comportament asincron**.

Bibliografie

- [1] Jonathan Knudsen, Patrick Niemeyer – *Learning Java, 3rd Edition*, O'Reilly.
- [2] http://www.itcsolutions.eu
- [3] http://www.acs.ase.ro
- [4] http://inf.ucv.ro/documents/tudori/laborator8_53.pdf

Curs 9 PPOO

Conf. univ. dr. Cristian CIUREA

Departamentul de Informatică și Cibernetică Economică

cristian.ciurea@ie.ase.ro

Java fundamentals

- Lucrul cu baze de date în Java
- Arhitectura JDBC
- Operații DDL/DML

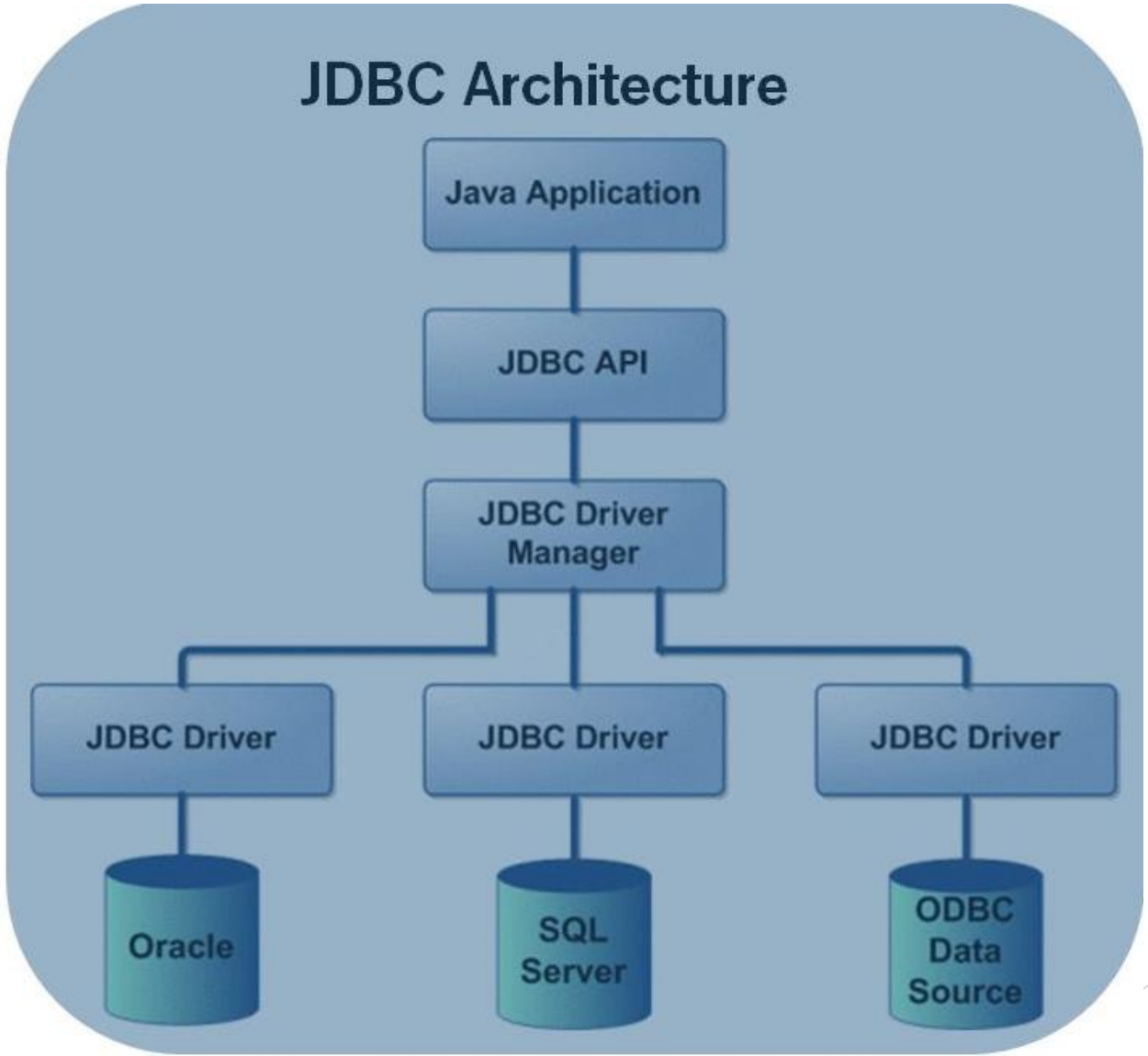
Lucrul cu baze de date în Java

- Instalarea ultimei versiuni de Java SE SDK
- Instalarea sistemului de gestiune a bazelor de date (DBMS) în cazul în care este necesar:
  - **Java DB**
    - http://www.java2s.com/Code/Jar/d/Downloadderbyclient10910jar.htm
  - **MySQL**
    - http://www.java2s.com/Code/Jar/m/Downloadmysqlconnectorjar.htm
  - **SQLite**
    - http://www.java2s.com/Code/Jar/s/Downloadsqlitejdbc372jar.htm

Lucrul cu baze de date în Java

- Instalarea și configurarea unui driver JDBC de la furnizorii bazelor de date utilizate:
  - org.apache.derby.jdbc.ClientDriver
  - com.mysql.jdbc.Driver
  - org.sqlite.JDBC
  - oracle.jdbc.driver.OracleDriver

Arhitectura JDBC



Lucrul cu baze de date în Java

- O aplicație JDBC utilizează unul sau mai multe drivere din pachetul **java.sql.*** care sunt utilizate de către clasa **DriverManager**.
- Driverele sunt specifice bazelor de date, deci pentru fiecare tip de bază de date se utilizează un driver special.
- În aceeași aplicație putem lucra cu baze de date diferite, deci implicit și cu mai multe drivere.

Lucrul cu baze de date în Java

Procesul de conectare la o bază de date implică două operații:

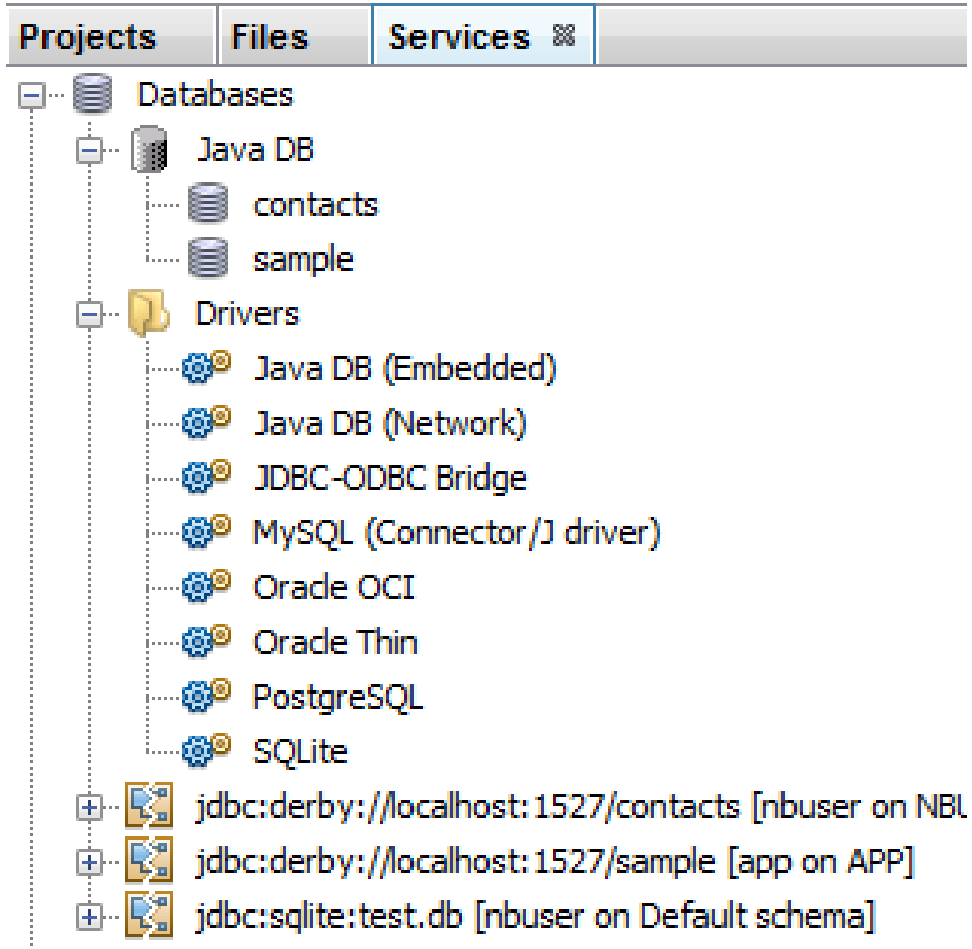
- încărcarea în memorie a unui driver corespunzător (**DriverManager**);
- realizarea unei conexiuni propriu-zise:
  - **DriverManager.getConnection(url);**
  - **DriverManager.getConnection(url, username, password);**
  - **DriverManager.getConnection(url, dbproperies);**

Lucrul cu baze de date în Java

- Baza de date **Java DB** este o distribuție de *Apache Derby* suportată de compania Sun.
- **Java DB** este un server complet tranzacțional, sigur, bazat pe standarde de baze de date, scris în întregime în Java, și care suportă în totalitate SQL, JDBC API și tehnologia Java EE.
- Baza de date **Java DB** este asamblată împreună cu serverul de aplicații *GlassFish* și este inclusă, de asemenea, în JDK 6.
- Dacă există serverul *GlassFish* înregistrat în instalarea NetBeans IDE, atunci **Java DB** va putea fi deja utilizată.

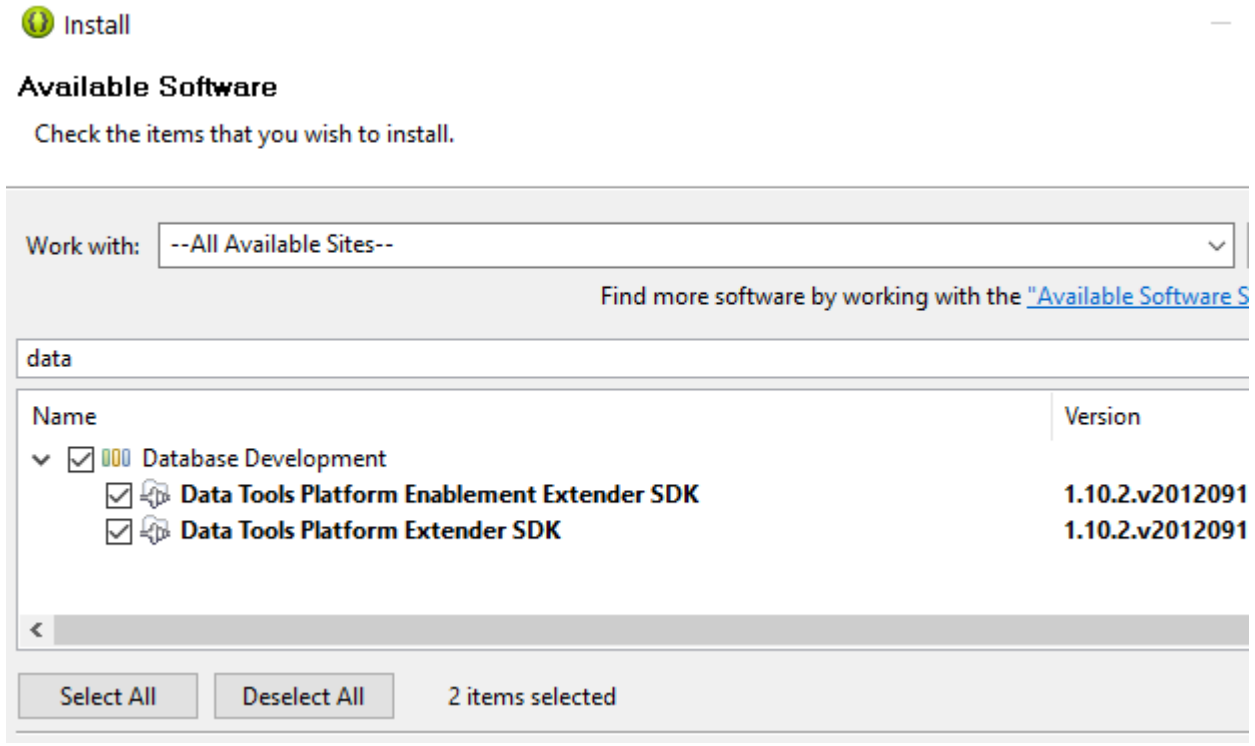
Lucrul cu baze de date în Java

- **NetBeans - Services - Databases**
- Crearea bazelor de date, tabelelor și popularea acestora



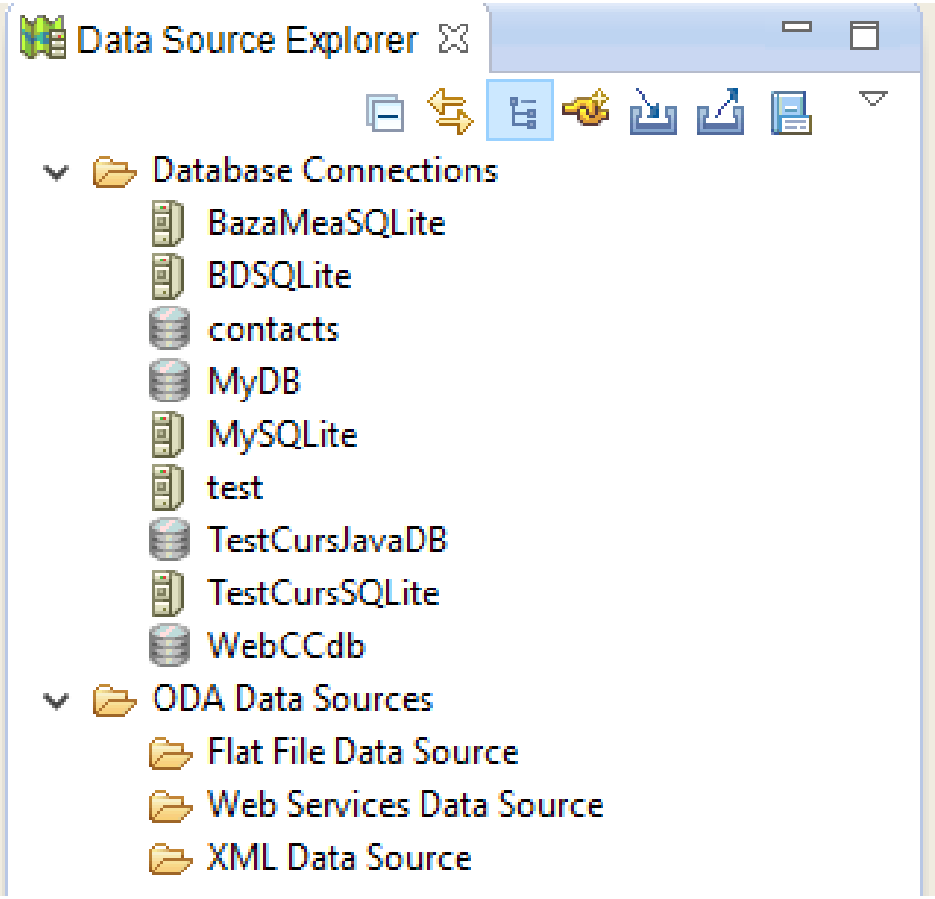
Lucrul cu baze de date în Java

- **Eclipse - Install New Software - Database Development**
- Crearea bazelor de date, tabelelor și popularea acestora



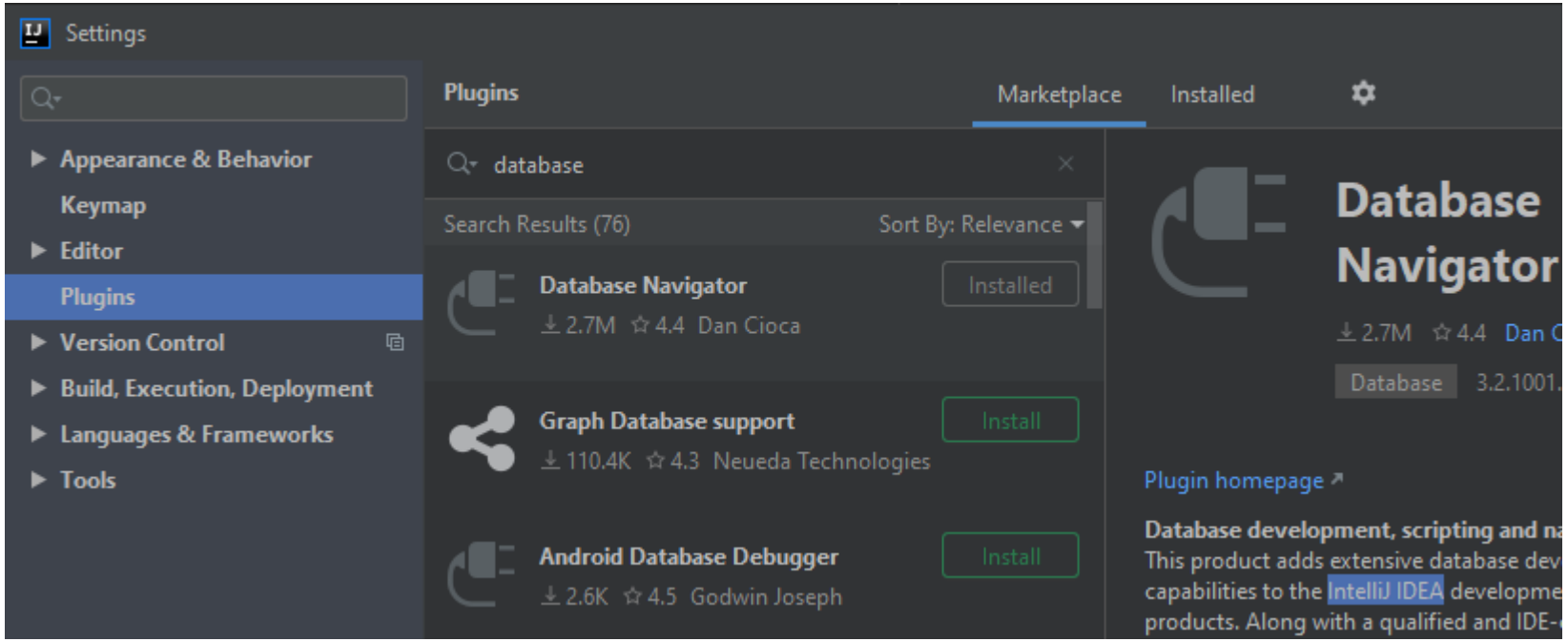
Lucrul cu baze de date în Java

- **Eclipse - Install New Software - Database Development**
- Crearea bazelor de date, tabelelor și popularea acestora



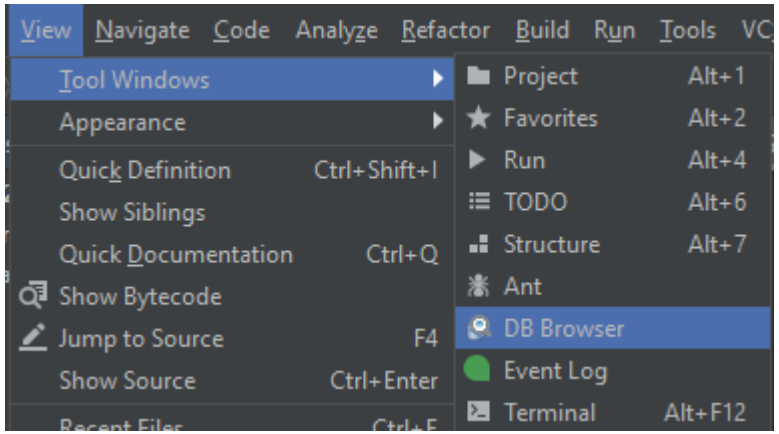
Lucrul cu baze de date în Java

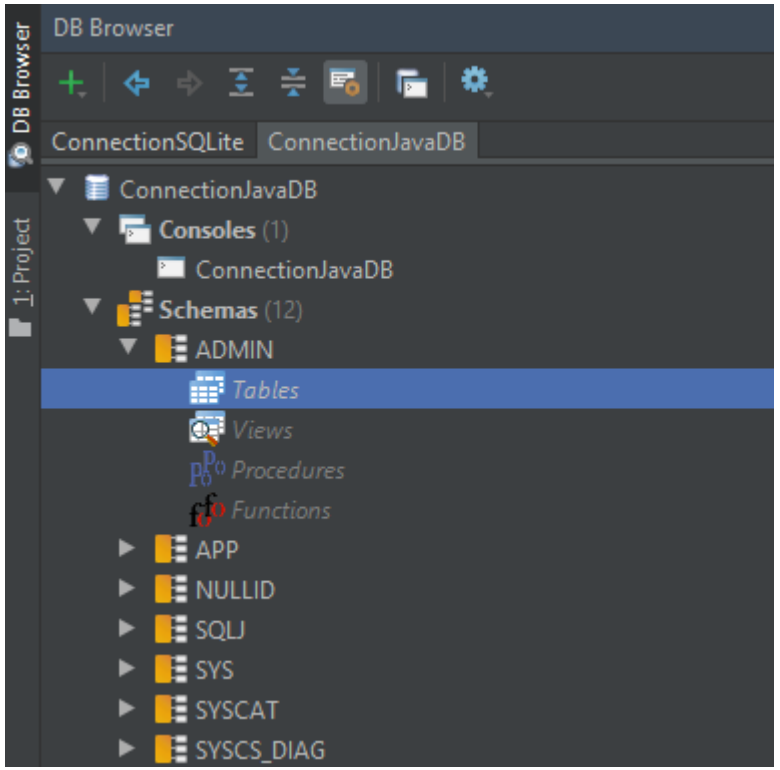
- **IntelliJ IDEA – Settings – Plugins**



Lucrul cu baze de date în Java

- **IntelliJ IDEA – DB Browser**





Lucrul cu baze de date în Java

În fereastra de *Servicii* a IDE-ului **NetBeans** sau din perspectiva *Database Development* a IDE-ului **Eclipse** se pot realiza următoarele operații asupra structurii bazei de date:

- crearea, ștergerea, modificarea tabelelor;
- popularea tabelelor cu date;
- vizualizarea datelor în format tabelar;
- execuția de instrucțiuni SQL și interogări.

Lucrul cu baze de date în Java

Interfețele și clasele pentru JDBC se găsesc în pachetul **java.sql.***.

Procesarea instrucțiunilor SQL cu JDBC presupune:

- stabilirea unei conexiuni;
- crearea unei instrucțiuni;
- execuția unei interogări;
- prelucrarea obiectului **ResultSet**;
- închiderea conexiunii.

Lucrul cu baze de date în Java

- Se apelează clasa **DriverManager** cerând un driver specific pentru baza de date;
- Driverul specific creează legătura cu baza de date și returnează un obiect de tip **Connection**;
- Cu ajutorul obiectului de tip **Connection** se creează un obiect **Statement** care conține și o cerere SQL către baza de date;
- Obiectul **Statement** returnează rezultatele într-un obiect **ResultSet**.

Lucrul cu baze de date în Java

- Crearea unei conexiuni la o bază de date:
- `Connection conn = DriverManager.getConnection (host, username, password);`

- Execuția unei instrucțiuni SQL asupra unei tabele necesită crearea unui obiect **Statement:**
- `import java.sql.Statement;`
- `Statement stmt = con.createStatement( );`

Lucrul cu baze de date în Java

Metodele cele mai importante ale obiectului *Statement*:

- *executeQuery(String)* – execută comanda SQL, returnează un obiect de tip *ResultSet* și se utilizează pentru execuția comenzilor SELECT;
- *executeUpdate(String)* – execută comanda SQL primită ca parametru și returnează numărul rândurilor tabelei modificate. Se utilizează pentru comenzile SQL de manipulare a datelor (INSERT, UPDATE, DELETE) și pentru comenzi de definire a datelor (CREATE/DROP TABLE);
- *execute(String)* – poate fi privită ca fiind generalizarea celorlaltor două metode. Se utilizează dacă comanda SQL poate returna deodată mai multe rezultate sau nu se cunoaște rezultatul execuției.

Lucrul cu baze de date în Java

Instrucțiunile JDBC se reprezintă utilizând următoarele clase:

- **Statement** - instrucțiunea este trimisă la serverul de bază de date de fiecare dată;
- **PreparedStatement** - instrucțiunea este stocată în memoria cache și apoi calea de execuție este predeterminată pe serverul bazei de date, permițându-i să fie executată de mai multe ori într-o manieră eficientă;
- **CallableStatement** - folosit pentru executarea procedurilor stocate în baza de date.

Operații DDL/DML

- Crearea unei tabele (CREATE TABLE);
- Operația de inserare date (INSERT);
- Operația de interogare (SELECT);
- Operația de actualizare (UPDATE);
- Operația de ștergere (DELETE).

Operații DDL/DML

- private **connect()**
- public boolean **update()**
- public **ResultSet query(String strSql)** {

```
try {
  Statement tmpStatement =
  connection.createStatement();
  ResultSet resultSet =
  tmpStatement.executeQuery(strSql);
  return resultSet;
  }
  catch (SQLException ex) {
    //handle exception here
    return null; }
}
```

Operații DDL/DML

- Un **ResultSet** este o modalitate de a stoca și manipula înregistrările returnate dintr-o interogare SQL.
- Odată ce avem la dispoziție toate înregistrările într-un **ResultSet**, există metode pe care le putem utiliza pentru a manipula înregistrările:
  - **`next()`** – poziționare pe linia următoare;
  - **`previous()`** – poziționare pe linia precedentă;
  - **`first()`** – poziționare pe prima linie;
  - **`last()`** – poziționare pe ultima linie;
  - **`absolute()`** – poziționare pe poziția indicată.

Operații DDL/DML

- **ResultSetMetaData** este un obiect care poate fi folosit pentru a obține informații despre tipurile și proprietățile coloanelor dintr-un obiect **ResultSet**:
  - **getColumnCount();**
  - **getColumnName(i);**
  - **getColumnTypeName(i);**

Lucrul cu baze de date în Java

- În Java nu există niciun cadru de interogare care să ofere o integrare directă la nivel de limbaj așa cum face LINQ pentru C# (similar cu Entity Framework).
- Alternative Java la Entity Framework :
  - **JPA (Java Persistence API)**: un ORM standard Java pentru stocarea, accesul și gestiunea obiectelor Java într-o bază de date relațională. Este parte integrantă a Java Enterprise Edition;
  - **Hibernate**: instrument de mapare obiect-relațional pentru limbajul de programare Java. Acesta oferă un cadru pentru maparea unui model orientat pe obiecte la o bază de date relațională.

Bibliografie

- [1] Jonathan Knudsen, Patrick Niemeyer – *Learning Java, 3rd Edition*, O'Reilly.
- [2] http://www.itcsolutions.eu
- [3] http://www.acs.ase.ro
- [4] http://www.tutorialspoint.com/sqlite/sqlite_java.htm
- [5] https://netbeans.org/kb/docs/ide/java-db.html
- [6] https://www.ms.sapientia.ro/~manyi/teaching/oop/oop_romanian/curs10/curs10.html
- [7] http://www.vogella.com/tutorials/EclipseDataToolsPlatform/article.html

Curs 10 PPOO

Conf. univ. dr. Cristian CIUREA

Departamentul de Informatică și Cibernetică Economică

cristian.ciurea@ie.ase.ro

Agenda

- Spring Framework
- Swing versus AWT
- JavaFX

Spring Framework

- **Spring** Framework este o platformă open source pentru simplificarea scrierii aplicațiilor în limbajul Java.
- Deși este folosit în principal pentru platforma Java EE, **Spring** poate fi utilizat pe orice aplicație Java.
- **Spring** este considerat o alternativă la modelul Enterpise JavaBeans (EJB).
- **Spring** a fost lansat în anul 2002 de către Rod Johnson împreună cu cartea sa, *Expert One-on-One: J2EE Design and Development*. În iunie 2003 Spring a intrat sub licența Apache 2.0.

Spring Framework

Pași pentru instalare și utilizare **Spring** Framework:

- Instalare Java Development Kit (JDK)
- Instalare Apache Common Logging API
- Configurare Eclipse
- Configurare biblioteci Spring Framework

Spring Framework

Beneficii utilizare **Spring** Framework:

- Spring permite dezvoltatorilor să dezvolte aplicații enterprise;
- Spring este organizat într-o manieră modulară;
- Spring utilizează unele dintre tehnologiile existente, cum ar fi ORM, JEE, Quartz și JDK, precum și alte tehnologii de vizualizare;
- Testarea unei aplicatii scrise cu Spring este simplă, deoarece codul dependent de mediu este transmis și integrat în cadrul framework-ului;
- Spring este un framework web MVC bine conceput, care oferă o alternativă foarte bună pentru alte framework-uri, cum ar fi Struts sau altele mai puțin populare.

Swing versus AWT

- **Swing** este un set de instrumente GUI pentru Java. Este parte din Oracle Java Foundation Classes (JFC) - un API pentru a furniza o interfață grafică cu utilizatorul (GUI) pentru aplicațiile Java.
- **Swing** a fost dezvoltat pentru a oferi un set mai sofisticat de componente GUI decât cele anterioare din **Abstract Window Toolkit (AWT)**.
- În plus față de componentele clasice, cum ar fi butoanele, checkbox-urile și etichetele, **Swing** oferă mai multe componente avansate, cum ar fi panoul de tab-uri (tabbed panel), scroll panes, tabele și liste.
- Spre deosebire de componentele **AWT**, componentele **Swing** nu sunt implementate prin cod specific platformei. Acestea sunt scrise în întregime în Java și sunt independente de platformă.

Swing versus AWT

- Începând cu versiunea Java 1.1 au apărut Java Foundation Classes (JFC) care includ un set numeros de componente denumite Swing.
- În vederea scrierii aplicatiilor care folosesc aceste componente trebuie importat package-ul **javax.swing.***
- În general, ferestrele care stau la baza aplicatiilor Swing sunt derivate din clasa **JFrame** care contine margini, titlu, butoane pentru închiderea, respectiv minimizarea fereastrei.

Swing versus AWT

- Clasele din pachetul **javax.swing.*** sunt bazate pe arhitectura MVC. Modelul este reprezentarea logică, view-ul este reprezentarea vizuală și controller-ul specifică comportamentul.
- Putem avea mai multe view-uri asociate aceluiași model. De exemplu, putem vizualiza aceleași date sub formă de tabel sau sub formă de grafic.
- În **AWT**, fiecare componentă grafică are o clasă nativă (clasa peer) care comunică cu componenta nativă a sistemului de operare. Componentele din pachetul Swing nu mai au aceste clase peer și pot arăta în mod diferit pe aceeași platformă din cauză că respectă arhitectura MVC.

Swing versus AWT

Pachetul Swing conține două tipuri de componente:

- containere (**JFrame, JApplet, JWindow, JDialog**);
- componente "lightweight" (**JButton, JPanel, Jlist**, etc.).

Containerele reprezintă cadrul în care aceste componente pot exista.

Swing versus AWT

Diferente:

- **Swing**:
  - suportă modelul MVC
  - componentele necesită pachetul **java.swing.***
  - componentele sunt independente de platformă
  - rulează mai rapid decât AWT
  - pluggable look and feel
- **AWT**:
  - nu suportă modelul MVC
  - componentele necesită pachetul **java.awt.***
  - componentele sunt dependente de platformă
  - rulează mai lent
  - native look and feel

JavaFX

- Există mai multe biblioteci de dezvoltare a interfețelor grafice în Java, cea mai cunoscută fiind **Java Swing** și cea mai nouă fiind **JavaFX**.
- **JavaFX** este cea mai nouă implementare a platformei de dezvoltare pentru interfețe GUI de client. Cu toate că este încă într-un stadiu incipient, are avantaje față de vechiul **Swing**.

JavaFX

Avantaje **JavaFX**:

- integrare completă cu Java SE și JDK, începând cu versiunea 7, ceea ce implică faptul că aplicațiile JavaFX vor putea fi dezvoltate și rulate de către orice client cu această versiune de Java;
- inițial JavaFX a fost un limbaj de scripting, dar acum Oracle pune la dispoziție un API pentru dezvoltarea aplicațiilor direct în Java, pentru un mai bun management și reutilizare a codului;
- un nou motor (engine) grafic, denumit Prism, care face uz de accelerarea hardware oferită de GPU-urile moderne, precum și un nou manager de ferestre (Window Toolkit) numit Glass;
- un nou limbaj bazat pe XML, denumit FXML, folosit pentru descrierea interfețelor grafice, astfel încât să nu fie nevoie de recompilarea codului la fiecare modificare;

JavaFX

Avantaje **JavaFX**:

- o componentă care poate afișa conținut web și care poate fi integrată în orice interfață grafică JavaFX;
- o serie de componente noi de interfață, precum grafice, tabele, meniuri și panouri;
- un sistem de a împacheta aplicațiile astfel încât acestea să fie livrate cu toate bibliotecile necesare execuției;
- portabilitate pe Linux, Windows și Mac OS X;
- spre deosebire de Swing, aceeași aplicație poate fi rulată de sine stătător, ca applet, sau ca aplicație de tip Web Start.

JavaFX

- Ca și Java Swing, **JavaFX** se bazează pe o ierarhie de clase care implementează diferite componente și containere ce reprezintă elementele grafice. Analog clasei **JFrame**, în JavaFX, clasa care descrie fereastra principală a unei aplicații este **javafx.stage.Stage**.
- Un obiect de tip **Stage** conține, la un moment dat, o singură scenă (**javafx.scene.Scene**). Această scenă este inițializată dându-i-se dimensiunile scenei și container-ul care conține toate celălalte elemente din fereastră. Acest container este, de regulă, un panou, care pe lângă rolul de container, specifică și modul în care sunt afișate componentele, analog **LayoutManager**-ului din Swing.

JavaFX

- **JavaFX** este o bibliotecă Java pentru proiectarea, crearea, testarea și implementarea aplicațiilor GUI multi-platforma. Este destinat să înlocuiască Swing ca bibliotecă GUI standard pentru Java.
- Biblioteca **JavaFX** este scrisă în Java și este disponibilă pentru limbajele care pot fi executate pe platforma JVM, respectiv **Java**, **Groovy** și **JRuby**. Aplicațiile JavaFX sunt, de asemenea, independente de platformă.

JavaFX

- Există mai multe biblioteci de dezvoltare a interfețelor grafice în Java, cea mai cunoscută fiind **Java Swing** și cea mai nouă fiind **JavaFX**.
- **JavaFX** este cea mai nouă implementare a platformei de dezvoltare pentru interfețe GUI de client. Cu toate că este încă într-un stadiu incipient, are avantaje față de vechiul **Swing**.

JavaFX

Avantaje **JavaFX**:

- integrare completă cu Java SE și JDK, începând cu versiunea 7, ceea ce implică faptul că aplicațiile JavaFX vor putea fi dezvoltate și rulate de către orice client cu această versiune de Java;
- inițial JavaFX a fost un limbaj de scripting, dar acum Oracle pune la dispoziție un API pentru dezvoltarea aplicațiilor direct în Java, pentru un mai bun management și reutilizare a codului;
- un nou motor (engine) grafic, denumit Prism, care face uz de accelerarea hardware oferită de GPU-urile moderne, precum și un nou manager de ferestre (Window Toolkit) numit Glass;
- un nou limbaj bazat pe XML, denumit FXML, folosit pentru descrierea interfețelor grafice, astfel încât să nu fie nevoie de recompilarea codului la fiecare modificare;

JavaFX

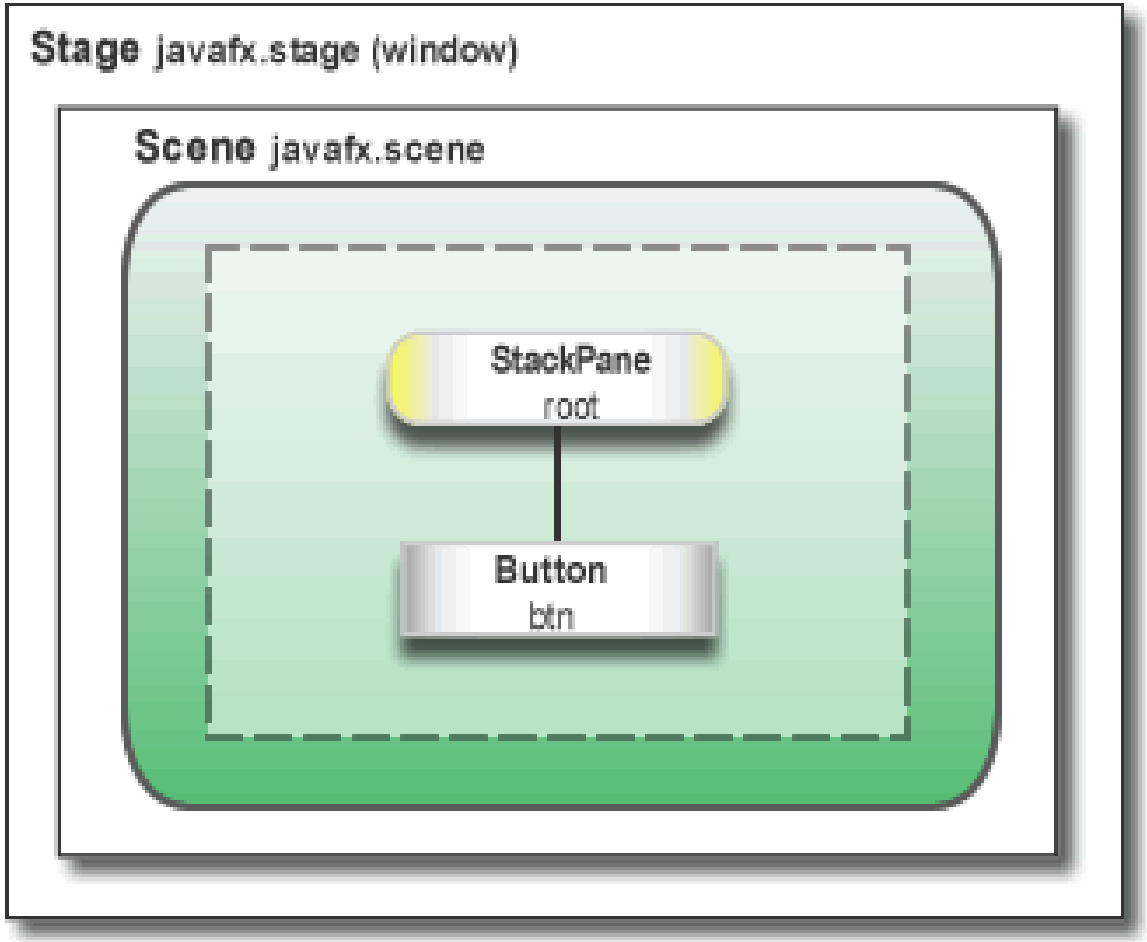
Avantaje **JavaFX**:

- o componentă care poate afișa conținut web și care poate fi integrată în orice interfață grafică JavaFX;
- o serie de componente noi de interfață, precum grafice, tabele, meniuri și panouri;
- un sistem de a împacheta aplicațiile astfel încât acestea să fie livrate cu toate bibliotecile necesare execuției;
- portabilitate pe Linux, Windows și Mac OS X;
- spre deosebire de Swing, aceeași aplicație poate fi rulată de sine stătător, ca applet, sau ca aplicație de tip Web Start.

JavaFX

- Ca și Java Swing, **JavaFX** se bazează pe o ierarhie de clase care implementează diferite componente și containere ce reprezintă elementele grafice. Analog clasei **JFrame**, în JavaFX, clasa care descrie fereastra principală a unei aplicații este **javafx.stage.Stage**.
- Un obiect de tip **Stage** conține, la un moment dat, o singură scenă (**javafx.scene.Scene**). Această scenă este inițializată dându-i-se dimensiunile scenei și container-ul care conține toate celălalte elemente din fereastră. Acest container este, de regulă, un panou, care pe lângă rolul de container, specifică și modul în care sunt afișate componentele, analog **LayoutManager**-ului din Swing.

JavaFX


Stage javafx.stage (window)
Scene javafx.scene
StackPane
root
Button
btn

JavaFX

- Pentru a ajuta dezvoltatorii să proiecteze aplicațiile, JavaFX oferă un instrument de design denumit **JavaFX Scene Builder**. Se pot draga componente UI într-un panou de conținut **JavaFX Content pane**, iar instrumentul generează codul **FXML** care poate fi utilizat într-un IDE precum NetBeans sau Eclipse.
- FXML este un limbaj de marcatori bazat pe XML care permite dezvoltatorilor să creeze o interfață cu utilizatorul (UI) într-o aplicație JavaFX, separat de implementarea logicii aplicației.

JavaFX

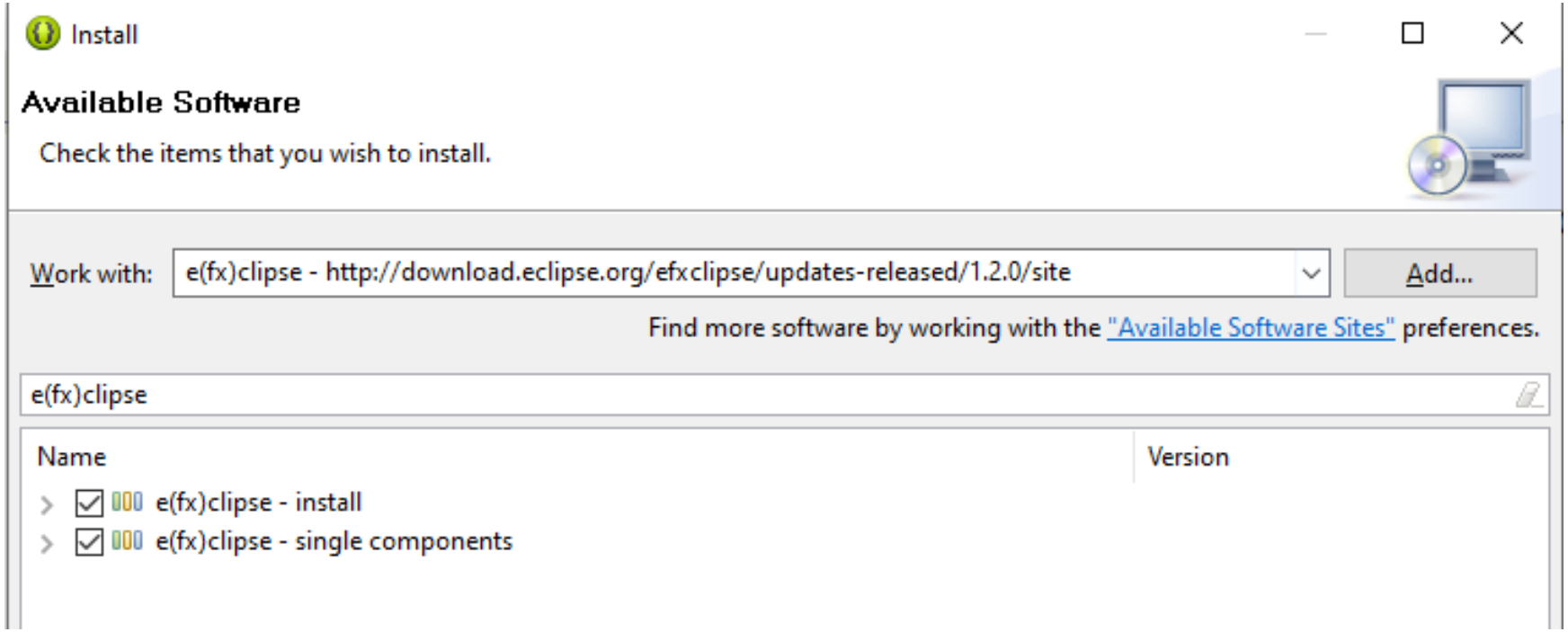
- Într-o aplicație JavaFX, se poate încorpora conținut Swing folosind clasa Swing Node. În mod similar, se pot actualiza aplicațiile Swing existente cu caracteristici JavaFX, cum ar fi conținut web încorporat și suport grafic bogat (**Swing Interoperability**).
- JavaFX oferă suport canvas in pachetul **javafx.scene.canvas**, care deține un set de clase pe care le putem folosi pentru a desena direct într-o zonă a scenei JavaFX. JavaFX oferă, de asemenea, clase pentru suport tipărire în pachetul **javafx.print**.

JavaFX

Ciclul de viață al unei aplicații JavaFX:

- punctul de intrare pentru aplicațiile JavaFX este clasa **Application**; de fiecare dată când este lansată o aplicație JavaFX se derulează următoarele etape, în ordinea aceasta:
- 1. se creează o instanță a clasei Application specificate.
- 2. se apelează metoda **init()** a clasei Application.
- 3. se apeleaza metoda **start()**.
- 4. in acest moment, aplicația este vizibilă pe ecran; iesirea din aplicatie se face atunci când aplicația apelează **Platform.exit()** sau
- când ultima fereastră a aplicației este închisă.
- 5. inainte de a ieși, se apeleaza metoda **stop()** din clasa Application; metoda stop() se poate supradefini pentru a distruge orice resurse utilizate de aplicație.

JavaFX


Install
Available Software
Check the items that you wish to install.
Work with:
e(fx)clipse - http://download.eclipse.org/efxclipse/updates-released/1.2.0/site
Add...
Find more software by working with the "Available Software Sites" preferences.
e(fx)clipse
Name
Version
e(fx)clipse - install
e(fx)clipse - single components

Bibliografie

- [1] Jonathan Knudsen, Patrick Niemeyer – *Learning Java, 3rd Edition*, O'Reilly.
- [2] http://www.itcsolutions.eu
- [3] http://www.acs.ase.ro
- [4] https://en.wikipedia.org/wiki/Swing_(Java)
- [5] http://wiki.dcae.pub.ro/index.php/Graphical_User_Interface_(GUI)_-_Java_Swing_%C8%99i_JavaFX
- [6] https://www.javaguides.net/2019/07/javafx-hello-world-example-tutorial.html?m=1
- [7] https://ro.wikipedia.org/wiki/Spring_Framework

Curs 11 PPOO

Conf. univ. dr. Cristian CIUREA

Departamentul de Informatică și Cibernetică Economică

cristian.ciurea@ie.ase.ro

Agenda

- JavaFX
- JUnit
- FindBugs

Resurse utile JavaFX

- https://www.slideshare.net/alexandercasall/javafx-10-things-i-love-about-you
- https://www.slideshare.net/alexandercasall/javafx-pitfalls
- https://o7planning.org/en/11009/javafx
- https://github.com/mhrimaz/AwesomeJavaFX

Swing sau JavaFX?

Swing

- Oferă maturitate și stabilitate
- Include o mulțime de biblioteci de componente și framework-uri
- Oferă o multitudine de resurse

JavaFX

- Modern, suport MVC, CSS, FXML
- Suport 3D și Animații
- Este posibil să nu fie încă destul de „solid” în producție
- Nu oferă atât de multe resurse

JUnit

- **Unit Testing**: testarea celor mai mici componente ale codului sursă (clase sau metode).
- **Unit Testing** este o metodă de testare software prin care unitățile individuale de cod sursă sunt testate pentru a determina dacă acestea sunt bune pentru a fi utilizate.
- **Test-Driven Development (TDD)** este un proces de dezvoltare software care se bazează pe repetarea unui ciclu de dezvoltare foarte scurt.
- **Test-Driven Development (TDD)** înseamnă:
  - scrierea unui test care eșuează;
  - modificarea codului sursă, astfel încât să meargă;
  - eliminarea redundanțelor.

JUnit

De ce să efectuăm **Unit Testing**?

- testele sunt ușor de scris;
- teste pot fi scrise ad-hoc, atunci când avem nevoie de ele;
- pe baza acestora se pot defini colecții de teste - denumite TestSuites;
- acestea pot fi rulate automat ori de câte ori este necesar (se scriu o dată și se utilizează de mai multe ori);
- există mai multe framework-uri și instrumente care simplifică procesul de scriere și rulare a testelor;
- reduce timpul petrecut la depanare și pentru găsirea de bug-uri;
- reduce numărul de bug-uri în codul livrat sau integrat;
- mărește rata de identificare a erorilor în timpul fazei de scriere a codului.

JUnit

- **JUnit** este un instrument pentru dezvoltare Test-Driven (TDD).
- **JUnit** este un cadru de clase care permite scrierea și executarea de teste pentru diferite metode / clase din codul sursă.
- Arhitectura JUnit include componentele:
  - **TestRunner,**
  - **TestResults,**
  - **TestCase,**
  - **Assert**

FindBugs

- **FindBugs** caută bug-uri în programele Java. Acesta se bazează pe conceptul de șabloane de bug-uri. Un șablon de bug este un cuvânt sau secvență de cod sursă, care generează de multe ori o eroare.
- FindBugs utilizează analiza statică pentru a inspecta bytecode-ul Java pentru a determina aparițiile șabloanelor de bug-uri.
- FindBugs poate identifica bug-uri pur și simplu prin inspectarea codului sursă al unui program, fără ca execuția programului să fie necesară.

FindBugs

Descrierea tipurilor de bug-uri identificate:

- **Bad practice** (încălcări ale practicilor recomandate și esențiale de scriere a codului sursă);
- **Correctness** (o eroare aparentă de codificare care rezultă în cod și care nu a fost, probabil, ceea ce a dorit dezvoltatorul);
- **Malicious code vulnerability**;
- **Multithreaded correctness**;
- **Performance**;
- **Dodgy** (cod care este confuz sau scris într-un mod care conduce la erori).

FindBugs

Instalare FindBugs Eclipse plugin:

- In **Eclipse**, select **Help->Install new software**
- Click **New Remote Site**
- Name: **FindBugs update site**
- URL: http://findbugs.cs.umd.edu/eclipse
- You should see **FindBugs Feature**
- Select the **I accept option** to accept the license and click **Next**

FindBugs

Configurare FindBugs **Eclipse** plugin:

- In **Eclipse**, select **Windows->Preferences**, under **Java** select **FindBugs**

Configurare plugin **QAPlug-FindBugs** in **IntelliJ**:

- In **IntelliJ IDEA**, select **File>Settings>Plugins**, select **QAPlug-FindBugs**

Bibliografie

- [1] Jonathan Knudsen, Patrick Niemeyer – *Learning Java, 3rd Edition*, O'Reilly.
- [2] http://www.itcsolutions.eu
- [3] http://www.acs.ase.ro
- [4] https://en.wikipedia.org/wiki/Swing_(Java)
- [5] http://wiki.dcae.pub.ro/index.php/Graphical_User_Interface_(GUI)_-_Java_Swing_%C8%99i_JavaFX
- [6] https://www.javaguides.net/2019/07/javafx-hello-world-example-tutorial.html?m=1
- [7] https://ro.wikipedia.org/wiki/Spring_Framework