

Introducere in Java

Razvan Veina

July 13, 2018

Contents

1	Introducere	4
1.1	Java	4
1.1.1	Caracteristicile limbajului Java	5
1.1.2	Istoricul versiunilor	6
1.1.3	Editii	6
1.1.4	Unde gasim Java	6
1.1.5	JDK vs JRE	7
1.1.6	Instalarea JDK	7
1.1.7	Setarea variabilei PATH	7
1.1.8	Drumul de la fisierul Java la programul functional	8
1.2	Primul program Java	8
1.3	Variabila classpath	9
1.4	Eclipse IDE	9
1.4.1	Instalare Eclipse	10
1.4.2	Primul proiect in Eclipse	11
1.4.3	Rularea proiectului din afara Eclipse	13
1.4.4	Setari initiale in Eclipse	14
2	Concepte OOP	17
2.1	Ce este un obiect?	17
2.2	Ce este o clasa?	17
2.3	Ce este mostenirea?	18
2.4	Interfete	19
2.5	O privire mai atenta la HelloWorld	20
3	Elemente de baza	21
3.1	Comentarii	21
3.2	Variabile	22
3.3	Tipuri primitive	25
3.4	Literali	25
3.5	Array-uri	27
3.6	Operatorii	29
3.7	Expresii	33
3.8	Statement-uri	33
3.9	Blocuri	34
3.10	Instructiuni de control al flow-ului	34
3.10.1	Instructiunea if	34

3.10.2	Instructiunea if-else	34
3.10.3	Instructiunea switch	35
3.10.4	Instructiunea while si do-while	37
3.10.5	Instructiunea for	38
3.10.6	Instructiuni de ramificare	39
4	Clase si obiecte	41
4.1	Clase	41
4.1.1	Declararea claselor	42
4.1.2	Declararea variabilelor membru	42
4.1.3	Modificatori de acces	43
4.1.4	Declararea metodelor	43
4.1.5	Supraincarcarea metodelor	43
4.1.6	Constructori	44
4.1.7	Pasarea parametrilor	44
4.2	Obiecte	46
4.2.1	Crearea obiectelor	46
4.2.2	Declararea variabilelor de tip referinta	46
4.2.3	Initializarea obiectelor	47
4.2.4	Folosirea obiectelor	49
4.2.5	Garbage Collector-ul	50
4.2.6	Returnarea unei valori dintr-o metoda	50
4.2.7	Returnarea unei referinte dintr-o metoda	51
4.2.8	Cuvantul cheie this	51
4.3	Controlul accesului la membrii unei clase	52
4.4	Membrii statici	53
4.4.1	Variabile statice	53
4.4.2	Metode statice	53
4.5	Constante	53
4.6	Initializarea campurilor statice	54
4.7	Initializarea campurilor ne-statice	54
4.8	Clase nested	54
4.8.1	Scopul claselor nested	55
4.8.2	Clase nested statice	55
4.8.3	Inner classes	55
4.8.4	Conflicte de nume	56
4.8.5	Clase anonime	56
4.8.6	Lambda expressions	57
4.9	Enum-uri	58
4.10	Adnotari	59
4.10.1	Formatul adnotarilor	59
4.10.2	Adnotarile predefinite	60
5	Package-uri	60
5.1	Crearea unui package	60
5.2	Denumirea package-urilor	60
5.3	Folosirea membrilor unui package	61
5.3.1	Ambiguitati de nume	61
5.3.2	Import static	62
5.4	Managementul surselor si claselor	62

6	Interfete	62
6.1	Definirea unei interfete	63
6.2	Implementarea unei interfete	64
6.3	Folosirea unei interfete ca si un tip	64
6.4	Metode default. Metode statice	64
7	Mostenirea	65
7.0.1	Operatorul cast	65
7.1	Supradefinirea si ascunderea metodelor	66
7.2	Polimorfismul	66
7.3	Ascunderea campurilor	66
7.4	Cuvantul cheie super	66
7.5	Clasa Object	66
7.6	Cuvantul cheie final	67
7.7	Cuvantul cheie abstract	67
7.8	Mostenire vs agregare	67
8	Clase pentru manipulat numere si caractere	68
8.1	Clasele Number	68
8.2	Formatarea numerelor	69
8.3	Clase pentru operatii aritmetice complexe	69
8.4	Problemele cu aritmetica reala de precizie	69
8.5	Caractere	70
9	Clase pentru manipulat siruri de caractere	71
9.1	Conversia intre numere si siruri	71
9.2	Manipularea caracterelor dintr-un String	71
9.3	Cautarea in String	72
9.4	Inlocuirea caracterelor intr-un String	72
9.5	Compararea String-urilor si a portiunilor de String-uri	72
9.6	Clasa StringBuilder	72
10	Exceptii	73
10.1	Ce este o exceptie?	73
10.2	Cerintele de tratare a exceptiilor	73
10.3	Prinderea si tratarea exceptiilor	74
10.3.1	Blocul try	74
10.4	Blocul catch	74
10.5	Blocul finally	75
10.6	Blocul try-with-resources	75
10.7	Specificarea exceptiilor aruncate de o metoda	75
10.8	Aruncarea exceptiilor	75
10.9	Crearea claselor de exceptie	75
10.10	Avantajele exceptiilor	75
11	Colectii	76
11.1	Ce sunt colectiile?	76
11.2	Interfete de baza	76
11.2.1	Interfata Collection	76
11.2.2	Interfata Set	78

11.2.3	Interfata List	78
11.2.4	Interfata Queue	79
11.2.5	Interfata Deque	79
11.2.6	Interfata Map	79
11.2.7	Ordonarea obiectelor	79
11.2.8	Interfata SortedSet	79
11.2.9	Interfata SortedMap	80
11.3	Operatii agregate	80
11.3.1	Pipeline-uri si stream-uri	81
11.4	Implementari	81
11.4.1	Implementarile interfetei Set	81
11.4.2	Implementarile interfetei List	82
11.4.3	Implementarile interfetei Map	82
11.4.4	Implementarile interfetei Queue	82
11.4.5	Implementarile interfetei Deque	82
11.4.6	Implementari wrapper	83
11.5	Algoritmi polimorfici	83
12	Operatii I/O	83
12.1	Stream-uri I/O	83
12.1.1	Stream-uri de bytes	83
12.1.2	Stream-uri de caractere	83
12.1.3	Stream-uri buffered	84
12.1.4	Formatare si scanare	84
12.1.5	I/O din linia de comanda	84
12.1.6	Stream-uri de date	84
12.1.7	Stream-uri de obiecte	84
12.2	I/O la nivel de fisier	84
12.2.1	Clasa Path	84
12.2.2	Operatii cu fisierele	86
13	Anexe	87
13.1	Eclipse tips	87
13.2	Total commander tips	88

1 Introducere

1.1 Java

- un limbaj de programare orientat obiect initiat de Sun Microsystems. Prima versiune dateaza din 1995
- programele Java au avantajul ca pot rula pe orice sistem de operare atata timp cat exista un interpretor Java disponibil
- codul Java care ruleaza pe o platforma nu trebuie recompilat pentru a rula pe alta platforma (write once, run anywhere)
- exista o "masina virtuala" (Java Virtual Machine sau JVM) care executa codul Java pe masina reala

1.1.1 Caracteristicile limbajului Java

- Orientat Obiect – în Java totul este privit ca un obiect.
- Independent de platforma – o aplicație Java odată scrisă poate fi rulată pe orice platformă (Windows /UNIX/Linux...)
- Simplu – unul din obiectivele inițiale ale creatorilor limbajului Java a fost să fie ușor de învățat
- Independent de arhitectura – rezultatul compilării unui fișier Java este un fișier independent de arhitectura sistemului pe care rulează
- Interpretat – Java este un limbaj interpretat. În timpul rularii programului conținutul fișierului compilat este tradus în instrucțiuni specifice procesorului pe care rulează.

1.1.2 Istoricul versiunilor

- JDK 1.0 (1996)
- JDK 1.1 (1997)
- J2SE 1.2 (1998)
- J2SE 1.3 (2000)
- J2SE 1.4 (2002)
- J2SE 5.0 (2004)
- Java SE 6 (2006)
- Java SE 7 (2011)
- Java SE 8 (2014)

1.1.3 Editii

- Java SE – Java Standard Edition permite realizarea de aplicatii desktop (standalone), server, applet-uri(pentru rulare in browser)
- Java EE – Java Enterprise Edition permite dezvoltarea aplicatiilor de tip Enterprise, orientate pe servicii
- Java ME – Java Micro Edition – permite dezvoltarea aplicatiilor pe dispozitive mobile

1.1.4 Unde gasim Java

Java se poate descarca de la www.oracle.com.



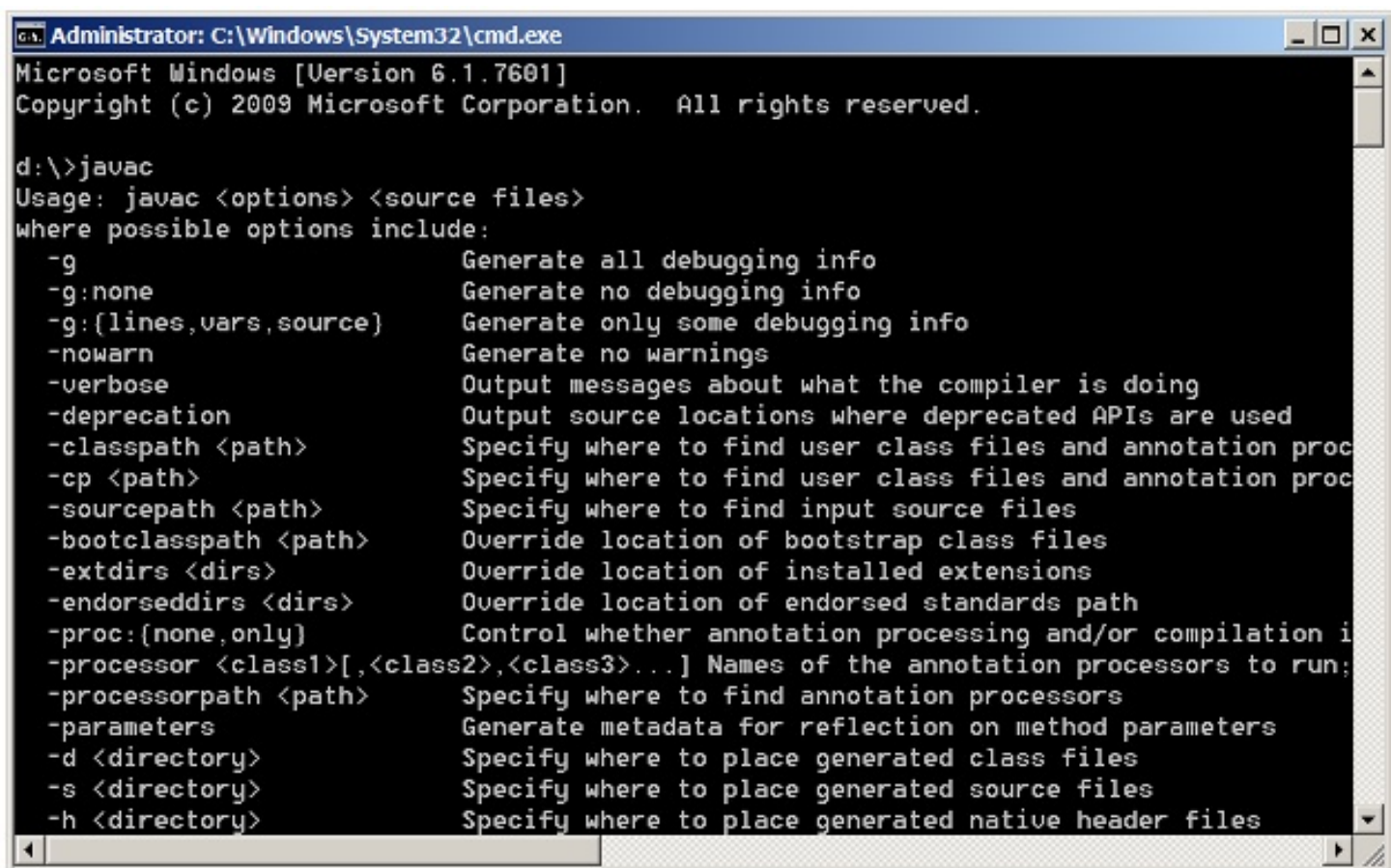
1.1.5 JDK vs JRE

- JRE – Java Runtime Environment – versiunea pentru utilizatorii de aplicatii Java. Contine tot ce e necesar pentru a putea rula aplicatii java
- JDK – Java SE Development Kit – versiunea pentru dezvoltatori. Include JRE si in plus tool-uri pentru dezvoltare, debug si monitorizare a aplicatiilor Java

1.1.6 Instalarea JDK

Dupa rularea kit-ului de instalare trebuie configurata variabila sistem PATH pentru a permite rularea tool-urilor java din orice loc de pe disc.

Pentru a verifica daca Java e corect configurat se deschide un cmd (Win-R si cmd). Aici se tasteaza comanda javac pentru a incerca rularea utilitarului javac.exe din JDK. Outputul in cazul unei setari corecte ar trebui sa arate ca in imagine:



```
Administrator: C:\Windows\System32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

d:\>javac
Usage: javac <options> <source files>
where possible options include:
  -g                  Generate all debugging info
  -g:none             Generate no debugging info
  -g:{lines,vars,source}  Generate only some debugging info
  -nowarn             Generate no warnings
  -verbose            Output messages about what the compiler is doing
  -deprecation        Output source locations where deprecated APIs are used
  -classpath <path>   Specify where to find user class files and annotation proc
  -cp <path>          Specify where to find user class files and annotation proc
  -sourcepath <path>  Specify where to find input source files
  -bootclasspath <path> Override location of bootstrap class files
  -extdirs <dirs>     Override location of installed extensions
  -endorseddirs <dirs> Override location of endorsed standards path
  -proc:{none,only}   Control whether annotation processing and/or compilation i
  -processor <class1>[,<class2>,<class3>...] Names of the annotation processors to run;
  -processorpath <path> Specify where to find annotation processors
  -parameters         Generate metadata for reflection on method parameters
  -d <directory>      Specify where to place generated class files
  -s <directory>      Specify where to place generated source files
  -h <directory>      Specify where to place generated native header files
```

1.1.7 Setarea variabilei PATH

- Se deschide fereastra Edit Environment Variables for your account
- la valoarea curenta a variabilei PATH se adauga (ideal la inceput) calea catre folder-ul bin din instalarea de JDK
- Dupa OK se redeschide un cmd si se retesteaza javac.exe

1.1.8 Drumul de la fisierul Java la programul functional



1.2 Primul program Java

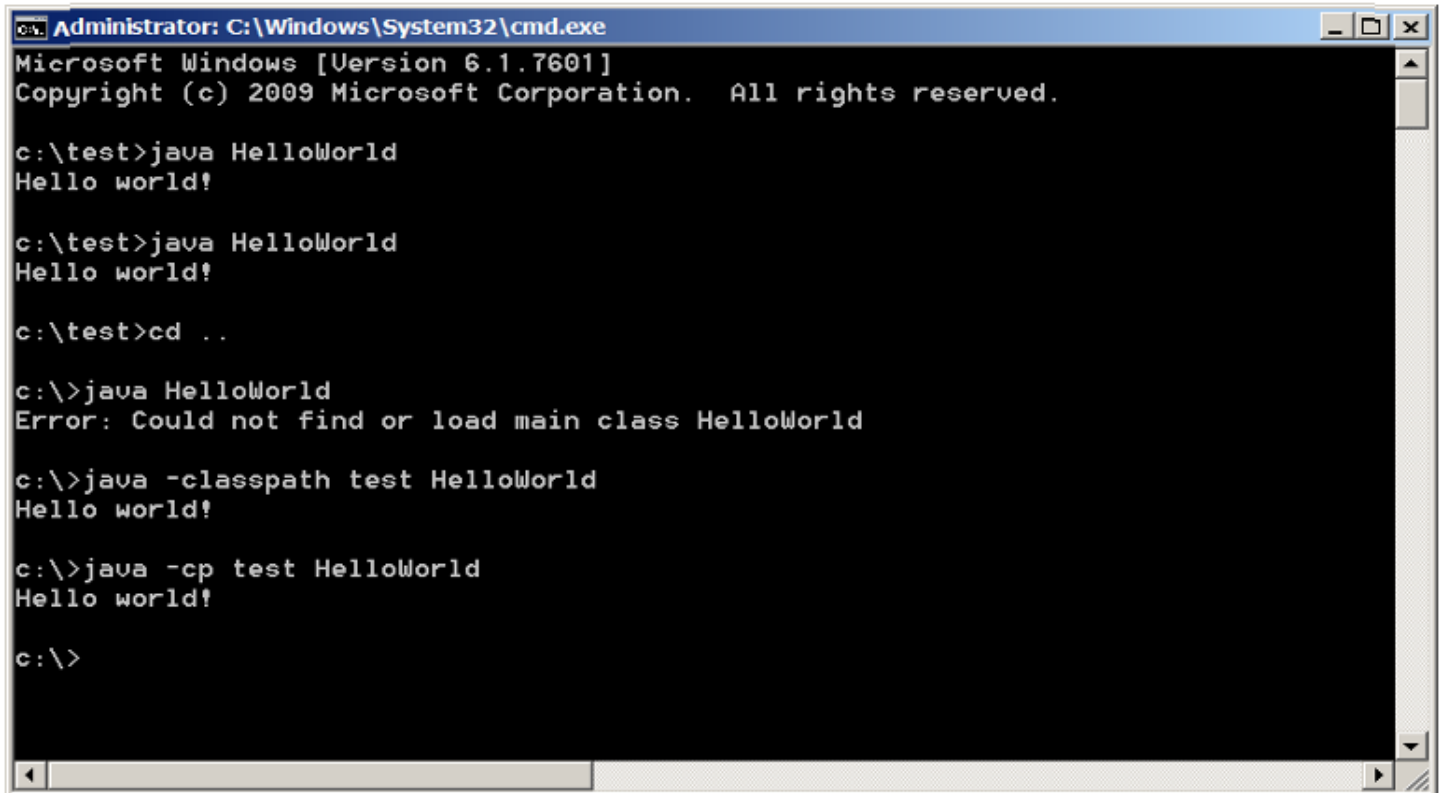
- Se creeaza un folder numit **test** in radacina si in el se creeaza un fisier numit **HelloWorld.java** cu continutul de mai jos:

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello world!");  
    }  
}
```

- Se deschide un cmd in folder-ul **test** si se ruleaza comanda **javac HelloWorld.java**
- In urma compilarii se observa ca a aparut in acelasi folder un fisier numit **HelloWorld.class**
- Pentru a executa acest fisier se foloseste comanda **java HelloWorld**

1.3 Variabila classpath

Daca schimbam folder-ul curent observam ca apelul java HelloWorld nu mai functioneaza, deoarece nu mai gaseste clasa HelloWorld. Acest lucru se intampla deoarece implicit masina virtuala cauta clasa in folder-ul curent. Pentru a comunica masinii virtuale unde sa gaseasca clasa specificata trebuie folosit un parametru aditional in linia de comanda:

A screenshot of a Windows command prompt window titled "Administrator: C:\Windows\System32\cmd.exe". The window shows the following commands and their outputs:

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

c:\test>java HelloWorld
Hello world!

c:\test>java HelloWorld
Hello world!

c:\test>cd ..

c:\>java HelloWorld
Error: Could not find or load main class HelloWorld

c:\>java -classpath test HelloWorld
Hello world!

c:\>java -cp test HelloWorld
Hello world!

c:\>
```

```
java -classpath test HelloWorld
```

sau

```
java -cp test HelloWorld
```

”test” reprezinta numele folder-ului unde poate fi gasita clasa cautata. Calea catre acesta poate fi absoluta sau relativa (C:\test sau test).

1.4 Eclipse IDE

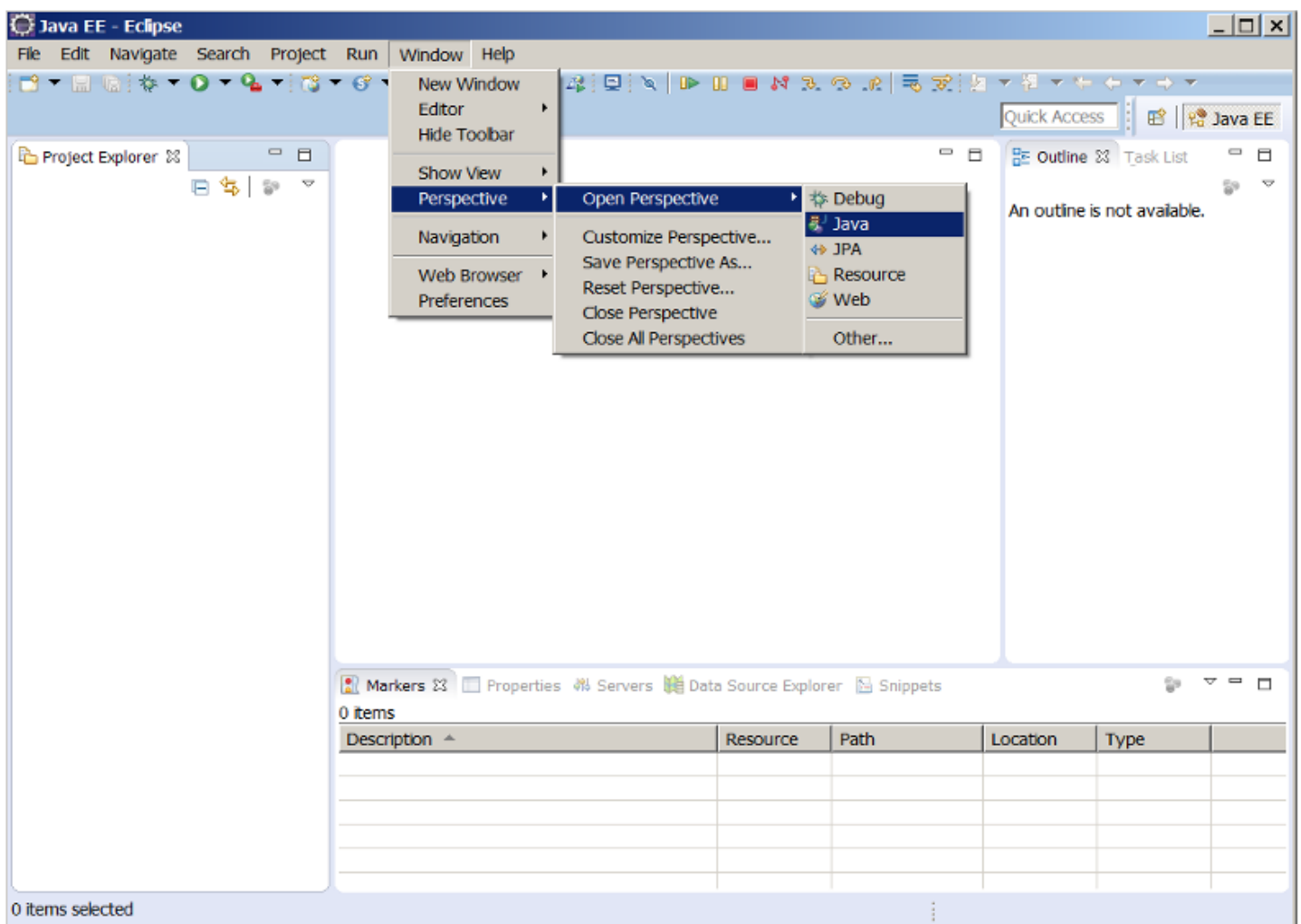
Pentru o mai usoara dezvoltare a aplicatiilor Java au aparut o serie de IDE-uri (Integrated Development Environment).

- Eclipse
- Netbeans
- IntelliJIdea
- BlueJ
- Jdeveloper

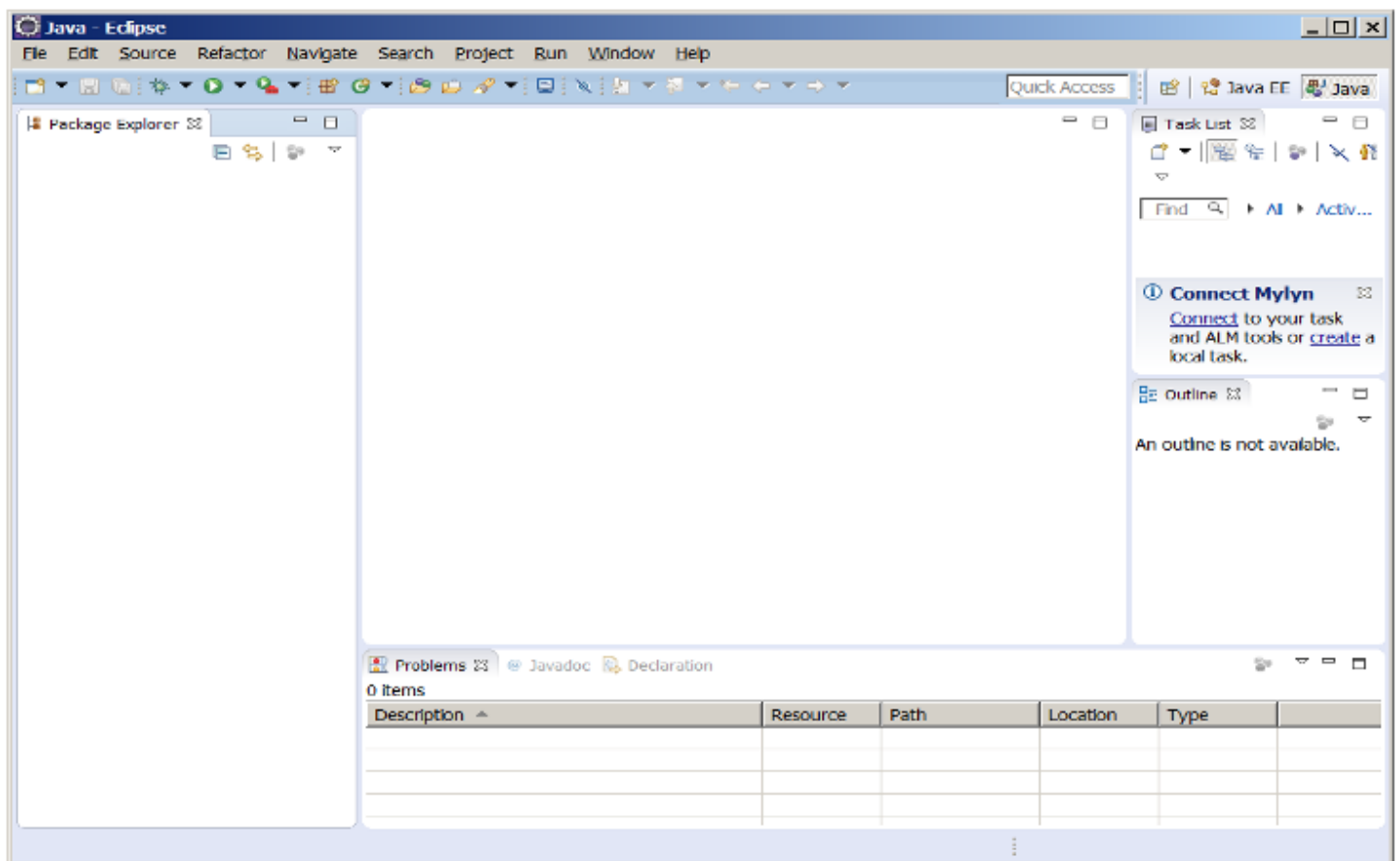
1.4.1 Instalare Eclipse

- Acesta se poate descarca de la www.eclipse.org.
- Download link: <https://www.eclipse.org/downloads/eclipse-packages/>
- Aici se alege Eclipse IDE for Java Developers
- Kit-ul de instalare e un fisier .zip care se dezarchiveaza intr-un folder numit eclipse.

Primul pas dupa pornirea Eclipse este deschiderea perspectivei de Java. O perspectiva Eclipse reprezinta o colectie si un aranjament de view-uri si editoare care faciliteaza anumite operatii. Exista de exemplu si o perspectiva de Debug. Un view reprezinta o fereastră care prezinta anumite informatii intr-un format special in functie de utilitatea sa.



Perspectiva de Java arata astfel:



Ea se poate customiza in functie de preferinte. O sugestie buna ar fi ca view-ul Problems care arata problemele curente din cod sa fie tot timpul vizibil pentru a identifica problemele aparute in cod cat mai rapid. De asemenea si view-ul Console care arata output-ul programelor.

1.4.2 Primul proiect in Eclipse

File->New->Java Project

Optiunile standard sunt de obicei suficiente. Locatia poate varia eventual, ea nu trebuie sa aibe neaparat legatura cu workspace-ul.

Folderul src reprezinta radacina structurii de clase. O practica profesionista este de a nu crea clase direct in folder-ul src ci de a crea o structura de package-uri pentru o organizare mai clara a claselor. Package-urile permit atat structurarea clara a claselor cat si rezolvarea conflictelor de nume, respectiv daca avem doua clase cu acelasi nume diferenta dintre ele se rezolva prin intermediul numelui package-ului.

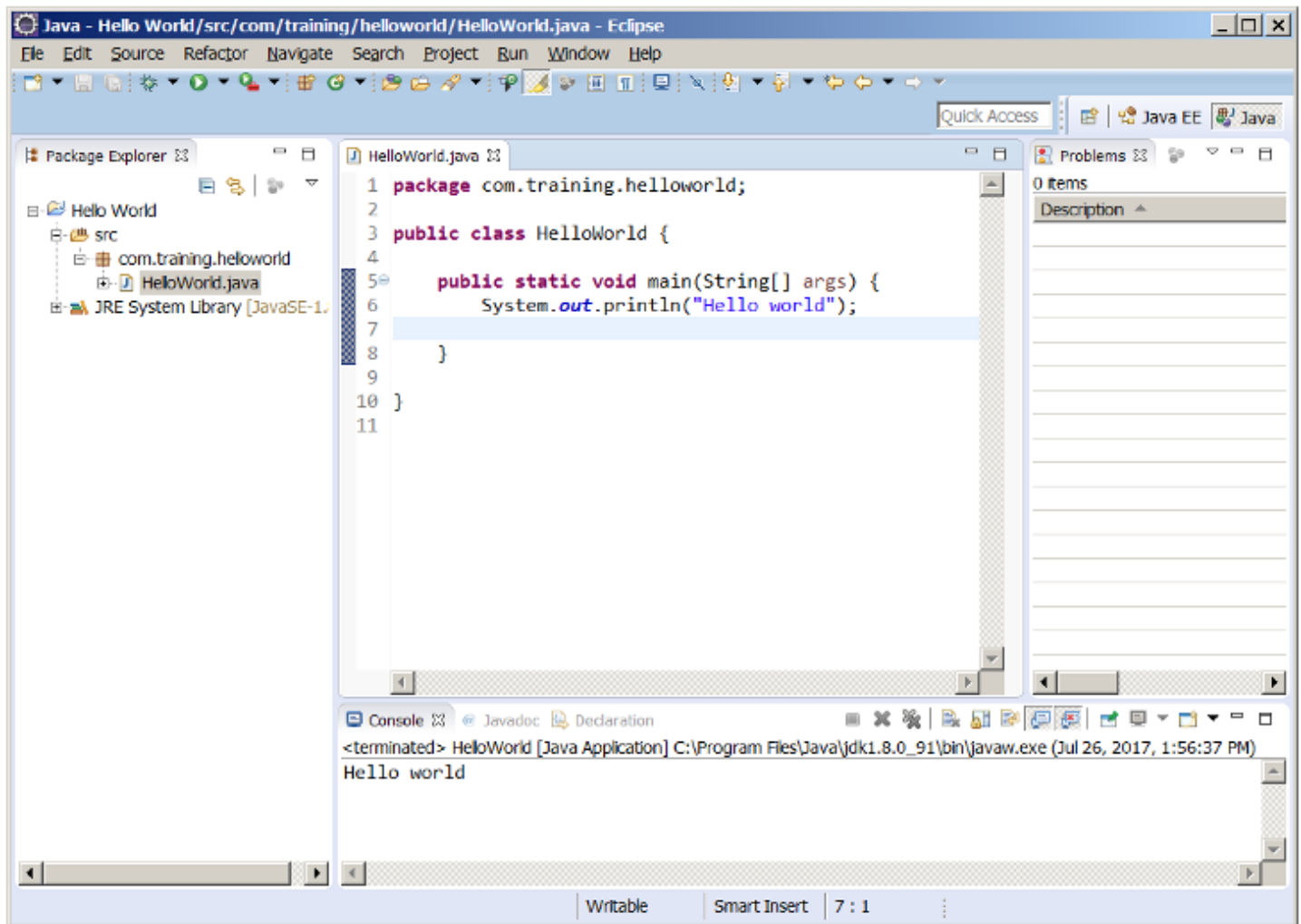
Pe disc structura de package-uri se pastreaza sub forma unei structuri de foldere.

Exista un code convention pentru numele package-ului, respectiv el trebuie sa contina doar litere mici. Un exemplu de nume de package pentru acest caz ar fi: `com.training.helloworld`.

In acest package vom crea o clasa numita `HelloWorld`. Vom checkui in partea de jos a dialogului de creare `public static void main(String[] args)` pentru ca Eclipse sa genereze automat o metoda `main` in clasa.

Pentru a scrie "Hello world" se tasteaza in interiorul metodei `main` `sysout` urmat de `Ctrl-Space`:

Eclipse va scrie automat `System.out.println()`; iar ce ramane de facut e de scris sirul care se doreste a fi afisat. Programul se poate rula cu `Ctrl-F11` sau `Alt-Shift-X, J`. Outputul ar trebui sa arate astfel:



Dar sa vedem ce anume creeaza Eclipse pe disc in urma acestor operatii. Pentru asta se selecteaza proiectul in Package Explorer si se apasa `Alt-Enter` ceea ce duce la afisarea ferestrei de Project properties. Aici se pot configura proprietatile proiectului. Momentan ne intereseaza locatia unde a fost creat proiectul, locatie care de altfel se poate specifica la creare.

Explorand acest folder cu Windows Explorer observam ca este compus din mai multe foldere si fisiere.

- `.settings` pastreaza setarile specifice proiectului
- `.classpath` pastreaza informatii despre classpath-ul proiectului
- `.project` pastreaza informatii despre proiect. `.classpath` si `.project` sunt esentiale pentru a putea deschide un proiect in eclipse
- `src` este radacina surselor java. Intrand in el se observa ca va contine o structura de foldere corespunzatoare structurii de package-uri. Pe ultimul nivel se va gasi clasa `HelloWorld.java`
- `bin` este radacina class-urilor care rezulta din compilarea surselor. Eclipse tine implicit in foldere separate fisierele `.java` si fisierele `.class`, si recomandarea e de a se pastra aceasta structura. Fisierele `.class` se compileaza automat oricand este salvat un fisier din proiect. Aceasta setare se poate schimba din meniul `Project->Build Automatically`, dar nu se recomanda acest lucru. In concluzie pasul pe care l-am facut din linie de comanda cand am executat comanda `javac` nu mai este necesar.

1.4.3 Rularea proiectului din afara Eclipse

În mod normal aplicațiile sunt realizate pentru clienți iar aceștia evident că nu vor dori să ruleze programul din Eclipse ci probabil cu ajutorul unui shortcut de Windows.

Pentru a rula programul din linie de comandă în primul rând trebuie să ne poziționăm în folder-ul rădăcină al proiectului.

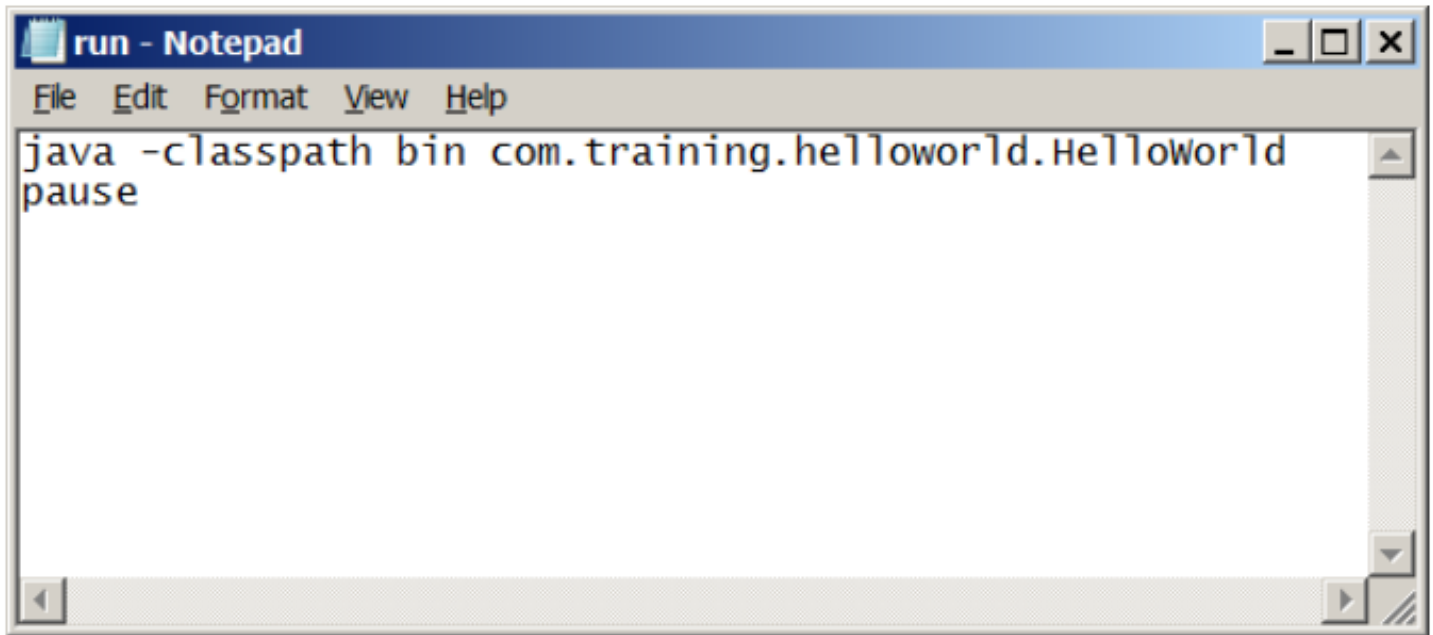
Prima tentativă e să rulăm comanda `java HelloWorld`. Aceasta nu va funcționa în primul rând deoarece în momentul când rulăm o clasă java trebuie să-i specificăm numele complet (**fully qualified name**). Acest nume conține și numele package-ului din care face parte clasa înainte. În concluzie o variantă mai bună ar fi:

```
java com.training.helloworld.HelloWorld
```

Nici aceasta nu va funcționa însă deoarece mașina virtuală nu găsește clasa respectivă deoarece implicit o caută în folder-ul curent. Pentru a o găsi trebuie să specificăm în variabila `classpath` rădăcina ierarhiei de clase, respectiv folderul `bin`:

```
java -classpath bin com.training.helloworld.HelloWorld
```

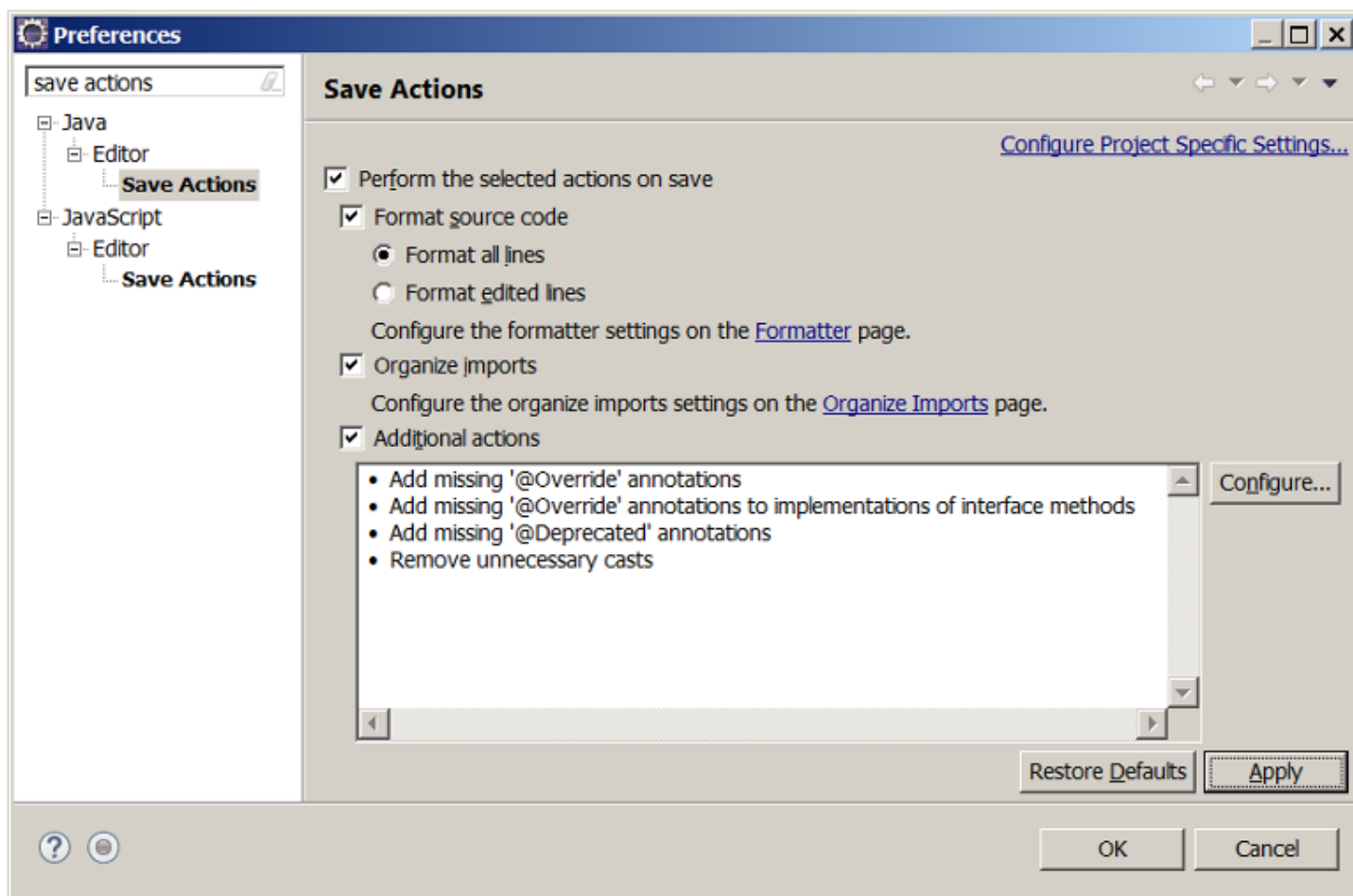
Dacă se dorește rularea acestui program cu ajutorul unui shortcut de Windows se poate crea un fișier de comenzi numit `run.bat` cu următorul conținut:



Creând un shortcut de Windows către acest fișier este posibilă rularea programului nostru direct din Windows, fără a mai fi necesară macar existența Eclipse-ului.

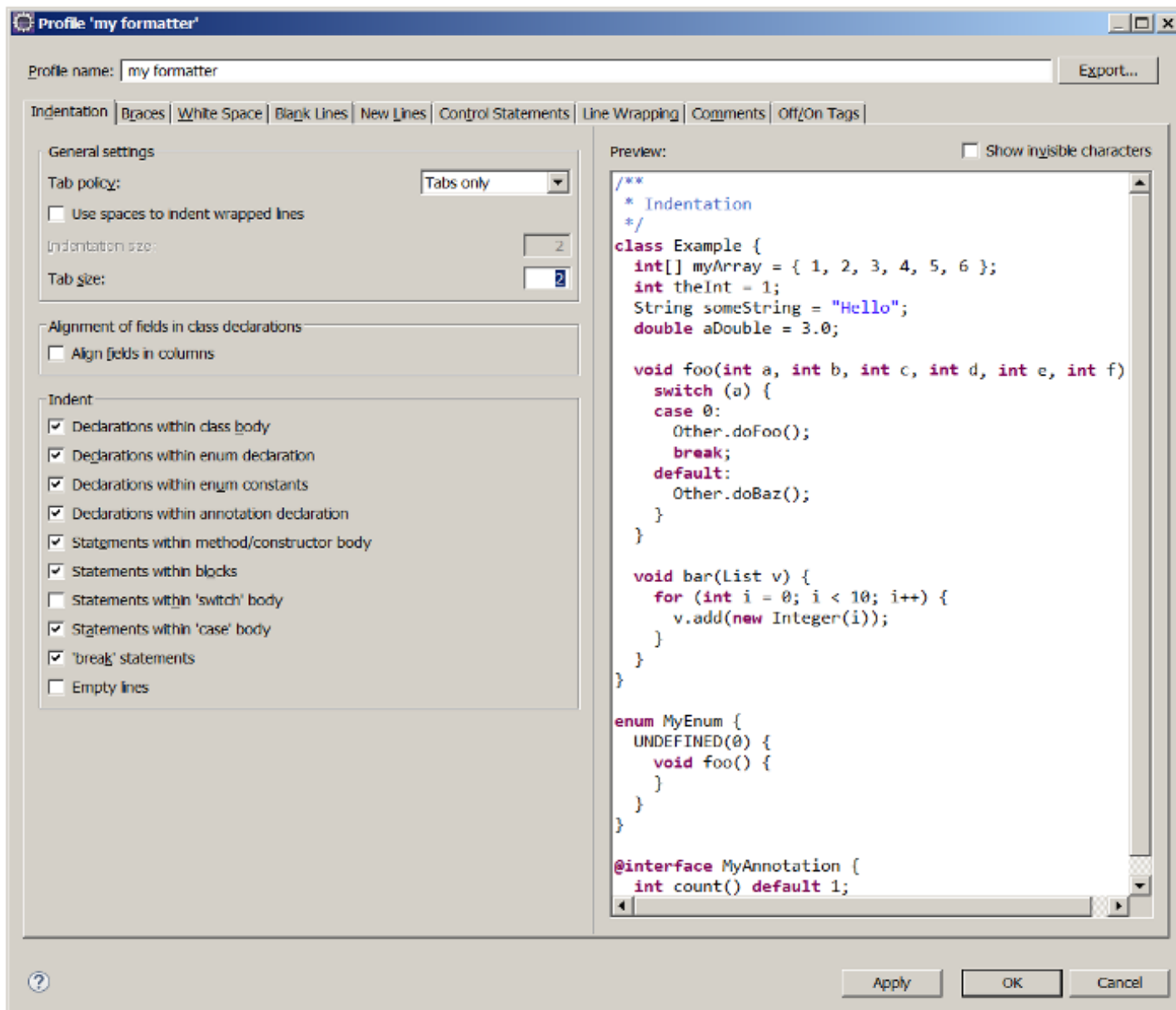
1.4.4 Setari initiale in Eclipse

Se recomanda formatarea automata a codului de catre Eclipse atat pentru a face codul sa arate mai bine ci si pentru avantaje in lucrul in echipa.

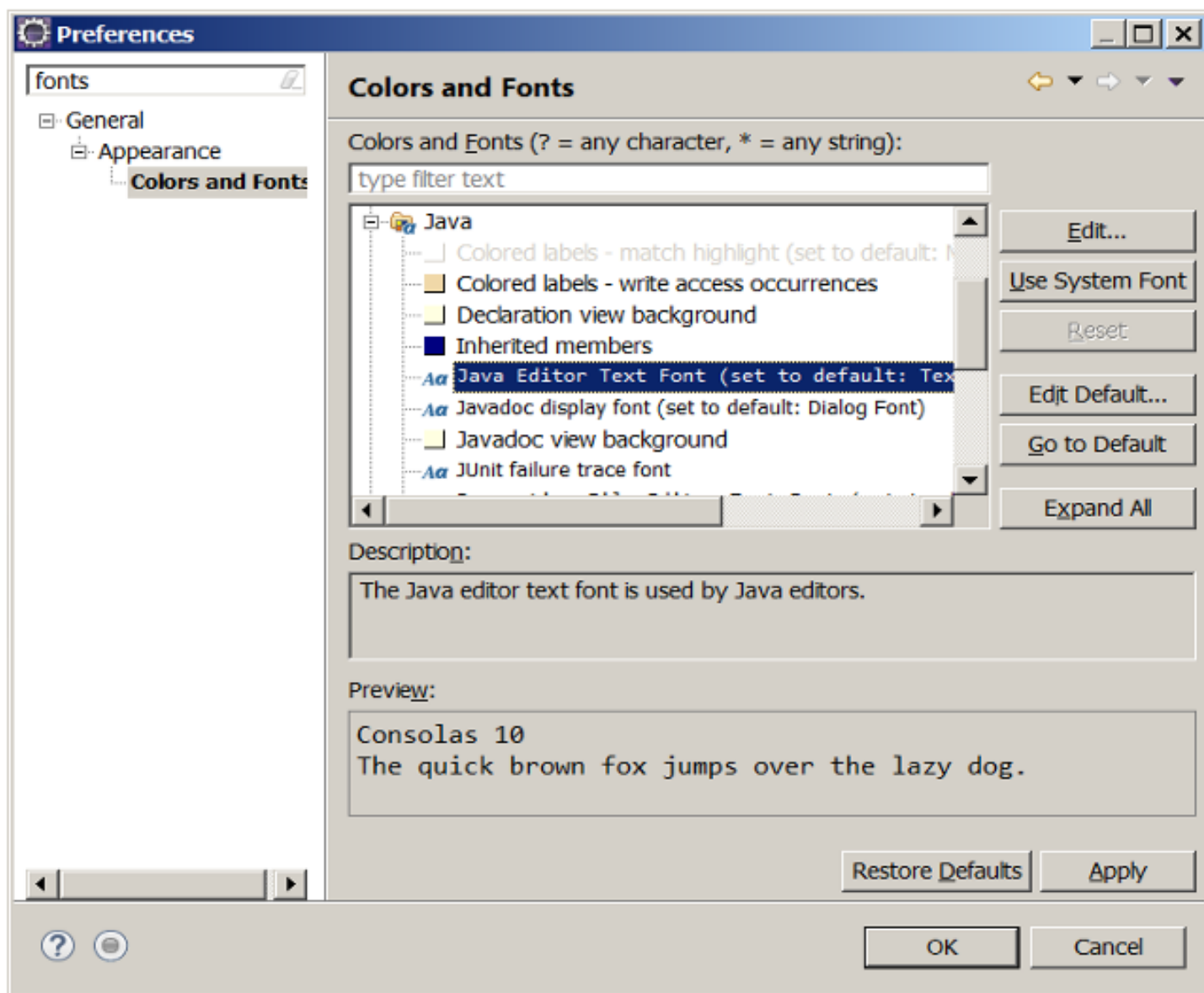


In urma acestor setari la fiecare salvare de fisier codul se va formata automat.

Totodata pentru a nu permite codului sa creasca prea mult in latime se recomanda setarea caracterului Tab la 2 caractere in loc de 4.



In caz ca se doreste se poate modifica si dimensiunea fontului. Se recomanda un font cat mai mic pentru a-l cuprinde cat mai bine cu privirea, evident la o dimensiune suficienta cat sa nu oboseasca ochii.



2 Concepte OOP

2.1 Ce este un obiect?

Daca ne uitam in jur observam ca suntem inconjurati de obiecte: Un caine, un birou, un televizor, o bicicleta.

Toate obiectele au doua caracteristici comune: toate au o stare si un comportament. De ex: cainii au o stare (nume, culoare, rasa, nivelul de foame, ...) si comportament (latra, alearga, dau din coada, ...). Si bicicletele au stare (viteza curenta, treapta de viteza curenta, cadenta pedalelor) si comportament (schimbarea vitezei, schimbarea ritmului, franarea). Este foarte important sa se identifice corect starea si comportamentul obiectelor.

Unele obiecte sunt compuse din mai multe alte obiecte. De exemplu o masina are in compozitie un motor, un rezervor, 4 roti, etc...

Obiectele din programare sunt conceptual asemanatoare cu obiectele din lumea reala. Ele au o stare care se pastreaza in variabilele membru (sau campuri) si ofera un comportament prin metodele lor. Metodele opereaza asupra campurilor care descriu starea obiectelor si permit comunicarea intre obiecte prin apelul lor.

Ascunderea starii interne a unui obiect si impunerea ca orice interactiune intre obiect si exterior sa se realizeze doar prin metodele obiectului poarta numele de incapsulare.

Impartirea codului in obiecte prezinta urmatoarele avantaje:

- Modularitatea – codul unui obiect poate fi scris si intretinut independent de alte obiecte
- Ascunderea informatiei – interactionand doar cu ajutorul metodelor unui obiect detaliile implementarii sale raman ascunse de exterior
- Reutilizarea codului – daca un obiect exista deja se poate folosi intr-un alt program
- Usurinta debugului – daca un anumit obiect prezinta probleme este suficienta repararea lui pentru ca programul sa functioneze

2.2 Ce este o clasa?

In lumea reala de multe ori exista obiecte care sunt toate de acelasi fel. De exemplu pot exista mii de biciclete toate de acelasi model. Fiecare a fost construita la fel dupa aceleasi schite si in concluzie au aceleasi componente. In OOP spunem ca bicicleta este o instanta a clasei de obiecte bicicleta. O clasa este schita dupa care se creeaza obiecte individuale.

Exemplu de clasa care modeleaza o bicicleta:

```
class Bicycle {  
  
    int cadence = 0;  
    int speed = 0;  
    int gear = 1;  
  
    void changeCadence(int newValue) {  
        cadence = newValue;  
    }  
  
    void changeGear(int newValue) {  
        gear = newValue;  
    }  
}
```

```

void speedUp(int increment) {
    speed = speed + increment;
}

void applyBrakes(int decrement) {
    speed = speed - decrement;
}

void printStates() {
    System.out.println("cadence:" +
        cadence + " speed:" +
        speed + " gear:" + gear);
}
}

```

Exemplu de clasa care creaza doua obiecte de tip Bicycle si le apeleaza metodele:

```

class BicycleDemo {
    public static void main(String[] args) {

        // Create two different
        // Bicycle objects
        Bicycle bike1 = new Bicycle();
        Bicycle bike2 = new Bicycle();

        // Invoke methods on
        // those objects
        bike1.changeCadence(50);
        bike1.speedUp(10);
        bike1.changeGear(2);
        bike1.printStates();

        bike2.changeCadence(50);
        bike2.speedUp(10);
        bike2.changeGear(2);
        bike2.changeCadence(40);
        bike2.speedUp(10);
        bike2.changeGear(3);
        bike2.printStates();
    }
}

```

2.3 Ce este mostenirea?

Exista multe obiecte care au lucruri in comun. De exemplu mountain bike-urile, bicicleta de sosea si bicicleta dubla sunt toate biciclete, dar mountain bike-urile au probabil suspensii, bicicleta de sosea are roti mari si cauciucuri subtiri, bicicleta dubla are 2 locuri. In OOP este posibil sa mostenim caracteristicile comune de la o asa numita superclasa. In cazul nostru Bicycle ar fi superclasa, iar MountainBike sau RoadBike ar fi subclasa. In java fiecare clasa poate sa aibe o singura superclasa. Cand o clasa mosteneste

(sau extinde) o alta clasa, subclasa va mosteni toate campurile si metodele superclasei, permitand ca in subclasa doar caracteristicile speciale, diferite, sa fie implementate.

Exemplu de clasa care mosteneste o alta clasa:

```
class MountainBike extends Bicycle {
    public int suspensionType;

    public void setupSuspension(){
        //...
    }
}
```

2.4 Interfete

Metodele pe care un obiect le expune in exterior constituie interfata acestuia cu exteriorul.

In cea mai simpla forma o interfata e compusa din metode fara implementare. Un obiect care implementeaza aceasta interfata se obliga sa furnizeze o implementare pentru fiecare din aceste metode iar din exterior se considera ca obiectul respectiv stie sa faca operatiunile descrise in interfata respectiva.

Exemplu de interfata:

```
interface Bicycle {
    void changeCadence(int newValue);

    void changeGear(int newValue);

    void speedUp(int increment);

    void applyBrakes(int decrement);
}
```

Exemplu de implementare a interfetei:

```
class MyBicycle implements Bicycle {

    int cadence = 0;
    int speed = 0;
    int gear = 1;

    void changeCadence(int newValue) {
        cadence = newValue;
    }

    void changeGear(int newValue) {
        gear = newValue;
    }

    void speedUp(int increment) {
        speed = speed + increment;
    }
}
```

```

void applyBrakes(int decrement) {
    speed = speed - decrement;
}

void printStates() {
    System.out.println("cadence:" +
        cadence + " speed:" +
        speed + " gear:" + gear);
}
}

```

2.5 O privire mai atenta la HelloWorld

Prima linie declara faptul ca aceasta clasa face parte din package-ul `com.training.helloworld`.

Plasarea clasei in alt loc decat in folder-ul `src/com/training/helloworld` ar genera o eroare de compilare.

```
public class HelloWorld { ... }
```

Aceasta linie declara o clasa numita `HelloWorld`.

Un code convention foarte important este ca un nume de clasa trebuie sa inceapa intotdeauna cu litera mare si apoi fiecare cuvant din nume trebuie sa inceapa cu litera mare (asa numitul CamelCase).

Intr-un fisier java se pot afla mai multe clase, dar una singura trebuie sa aibe cuvantul cheie `public` in fata, iar numele ei trebuie sa dea numele fisierului la care se adauga extensia `.java`.

Intre acolade se va scrie codul clasei respective (declaratii de variabile si metode).

Cuvantul cheie `public` semnifica faptul ca aceasta clasa va fi accesibila din alte clase din exterior.

Singura metoda din clasa `HelloWorld` este

```

public static void main(String[] args) {
    System.out.println("Hello world");
}

```

Eclipse tips!

- Tastand in cod in interiorul unei metode `sys` urmat de `Ctrl-Space` se va scrie automat codul `System.out.println()`;
- Tastand in cod in interiorul unei clase `main` urmat de `Ctrl-Space` se va scrie automat o metoda `main` default.
- `Alt-Shift-Up`, `Alt-Shift-Down` muta codul selectat sus/jos.
- `Ctrl-S` salveaza fisierul curent.
- `Ctrl-Space` in orice loc in cod prezinta ce optiuni am avea sa folosim in acel loc (ex:variabile, metode)

`public static void main(String[] args)` este o semnatura de metoda standard pentru punctul de pornire al unui program.

Cuvantul cheie `public` semnifica faptul ca aceasta metoda este vizibila din exterior.

Cuvantul cheie `static` semnifica faptul ca metoda apartine clasei `HelloWorld` si nu unei anume instante a ei. Detalii despre static mai tarziu.

Cuvantul cheie `void` semnifica faptul ca metoda nu returneaza nici o valoare.

`main` nu e cuvant cheie ci reprezinta numele metodei.

Pentru ca metoda sa poata fi apelata de JVM la pornirea unui program ea trebuie sa se numeasca exact asa si nu altfel. De asemenea trebuie sa aibe un singur parametru de tip `array`.

(`String[] args`) reprezinta un `array` de parametri de tip `String` (sir de caractere) pe care ii poate primi aceasta metoda. Parametrii pot fi specificati din linie de comanda.

`System.out.println("Hello world");` este modul standard in care se tipareste un text la consola in Java. La `String`-ul dintre ghilimele se va adauga automat o linie noua la sfarsit. Versiunea care nu adauga o linie noua este `System.out.print();`

Reguli referitoare la fisierele sursa:

- Intr-un fisier sursa poate sa existe o singura clasa `public`
- Comentariile pot aparea oriunde intr-un fisier sursa
- Daca exista o clasa publica intr-un fisier java atunci numele fisierului trebuie sa se potriveasca cu numele clasei. Ex: clasa `public class Car` trebuie sa apara in fisierul `Car.java`
- Daca clasa face parte dintr-un package atunci `statement-ul package` trebuie sa fie prima linie din clasa, inainte de `statement-urile de import`
- Daca exista `import-uri` ele trebuie sa fie intre package si declaratia clasei, sau daca nu exista package atunci `import-urile` trebuiesc sa fie primele.
- `Statement-urile package` si `import` se refera la toate clasele dintr-un fisier sursa
- Un fisier sursa poate contine mai multe clase ne-publice
- Fisierele sursa care nu contin nici o clasa publica pot sa aibe un nume care nu se potriveste cu nici o clasa din ele.

3 Elemente de baza

3.1 Comentarii

Exista 3 tipuri de comentarii:

- Comentarii pe o singura linie. Au forma:

```
int length; // lungimea sirului
```

- Comentarii pe mai multe linii:

```
/* Aici incepe un comentariu  
pe mai multe linii  
si aici se termina */
```

- Comentarii de tip Javadoc. Sunt niste comentarii speciale care permit documentarea automata a codului cu ajutorul unei sintaxe mai speciale:

```

/**
 * Acest comentariu poate descrie functionalitatea unei metode
 */
public void metoda(){

```

Un cod bine scris nu ar trebui sa necesite prea multe comentarii deoarece ar trebui sa se intelaga cat mai clar din citirea codului care este functionalitatea acestuia.

3.2 Variabile

Variabilele sunt folosite pentru a stoca in ele valori in diferite contexte. In functie de context variabilele pot fi de urmatoarele tipuri:

- variabile membru (instance variables) sau campuri (fields) sau campuri ne-statice (non-static fields).
 - apartin intotdeauna unui obiect
 - descriu starea obiectului
 - nu au cuvantul cheie static in declaratie
 - fiecare obiect in parte are propriile campuri cu propriile lor valori. Ex: currentSpeed pentru o bicicleta e diferit de currentSpeed pentru alta bicicleta
- variabile statice (static fields) sau variabile de clasa (class variables)
 - sunt declarate cu ajutorul cuvantului cheie static
 - exista o singura copie a acestei variabile pentru toate obiectele
 - se poate considera ca apartine unei clase si nu unei instante anume a clasei
 - astfel de variabile se declara pentru a fi folosite in comun de toate instantele clasei
 - constantele de obicei se declara ca statice. In plus se mai adauga cuvantul cheie final care specifica faptul ca valoarea variabilei nu poate fi modificata ulterior.
- variabile locale (local variables)
 - sunt folosite pentru a stoca o situatie temporara in interiorul unei metode sau a unui bloc de cod
 - nu e necesar un cuvant cheie special care sa spuna ca o variabila este locala, acest lucru rezulta din locul in care apare declaratia variabilei. O variabila declarata intr-un bloc de cod interior unei metode va avea durata de viata doar pana la incheierea respectivului bloc de cod. Prin bloc de cod se intelege o sectiune cuprinsa intre si
- parametrii
 - parametrii sunt tot un fel de variabile locale in corpul unei metode, dar declaratia lor apare practic in semnatura metodei. Se declara separati prin virgula.

Numele variabilelor este case-sensitive

Numele variabilelor poate sa contina orice litera, cifra si caracterul _ sau \$, dar nu poate sa inceapa cu o cifra.

Numele unei variabile nu poate sa contina spatii

Nu exista limite la lungimea unei variabile

Nu pot fi folosite cuvinte cheie pe post de variabile.

Lista cu cuvinte cheie:

- abstract
- boolean
- break
- byte
- case
- catch
- char
- class
- const
- continue
- default
- do
- double
- else
- extends
- final
- finally
- float
- for
- goto
- if
- implements
- import
- instanceof
- int
- interface
- long
- native
- new
- package

- private
- protected
- public
- return
- short
- static
- strictfp
- super
- switch
- synchronized
- this
- throw
- throws
- transient
- try
- void
- volatile
- while
- assert
- enum

Exista un code convention unanim acceptat de programatorii java ca numele variabilelor sa inceapa cu litera mica, iar fiecare cuvnt urmator din nume sa inceapa cu litera mare. Ex: classVariable, localVariableInitialized

Exceptie fac constantele, care se scriu cu litere mari si cu '_' intre cuvinte.

Ex: CONSTANT_WITH_VALUE_0

Este foarte important atat pentru claritatea codului dar si pentru a arata profesionalismul programatorului ca numele variabilelor sa fie simple si clare si sa exprime intotdeauna scopul si utilitatea variabilei respective.

Variabilele membru nu trebuiesc initializate explicit, daca nu se initializeaza primesc niste valori implicite, despre care vom vorbi mai jos.

Variabilele locale trebuiesc initializate inainte de prima utilizare, altfel programul nu va fi compilabil

O variabila final trebuie si ea initializata la declarare pentru ca este singurul loc unde se poate face acest lucru.

3.3 Tipuri primitive

In Java toate variabilele trebuiesc declarate inainte de a fi folosite prima oara.

Declaratia consta in definirea tipului si a numelui variabilei si optional initializarea acesteia cu o valoare.

Ex:

```
int gear = 1;
```

Declaratia de mai sus spune ca o variabila cu numele gear exista, are tipul int si are valoarea initiala 1.

Tipul variabilei determina ce valori poate contine respectiva variabila si ce operatii se pot efectua asupra ei. Tipurile primitive sunt predefinite de limbaj iar numele lor sunt cuvinte cheie in Java.

Nume	Biti	Min	Max	Valoare	Val. initiala
byte	8	-128	127	Nr intreg	0
short	16	-32,768	32,767	Nr intreg	0
char	16	\u0000	\uFFFF	Unicode	\u0000
int	32	-2,147,483,648	2,147,483,647	Nr intreg	0
long	64	-9,223,372,036,854,775,808	9,223,372,036,854,775,807	Nr intreg	0
float	32	3.402,823,5 E+38	1.4 E-45	Nr real	0
double	64	1.797,693,134,862,315,7 E+308	4.9 E-324	Nr real	0
boolean	1	false	true	boolean	false

De notat ca operatiile aritmetice cu double sau float nu sunt intotdeauna precise si nu se recomanda folosirea lor in cazuri in care operatiile necesita precizie totala. Exemple si solutii mai tarziu.

Pe langa aceste tipuri primitive Java trateaza special pentru sirurile de caractere cu ajutorul clasei `java.util.String`. Un sir de caractere intre ghilimele va fi considerat de java automat un String, iar o astfel de constructie este posibila:

```
String s = "this is a string";
```

`String` nu e un tip primitiv, doar este tratat mai special de limbaj.

Desi se pot declara mai multe variabile pe o linie aceasta practica nu este recomandata, facand codul sa arate neprofesionist.

Ex:

In loc de

```
int x, y;
```

preferabil:

```
int x;
```

```
int y;
```

In cadrul unei clase variabilele membru se definesc grupat sus, inaintea metodelor.

In schimb la variabilele locale este o practica profesionista sa se declare cat mai aproape de locul unde sunt folosite, nu la inceputul metodei, ca in C.

3.4 Literali

Literalii sunt reprezentarea in codul sursa a unei valori fixe.

Literalii intregi sunt toti de tip int, in afara de cazul cand se termina cu un L, caz in care sunt long.

Se recomanda folosirea unui L mare desi se poate folosi si l mic, dar acesta este greu de distins de 1.

Literalii intregi se pot exprima in:

- decimal: baza 10, cifre de la 0 la 9

- hexa: baza 16 – cifre de la 0 la 9 si litere de la A la F
- binar: baza 2 – cifre de la 0 la 1 (incepand din java 1.7)

Un literal hexa se prefixeaza cu 0x Un literal binar se prefixeaza cu 0b

Ex:

```
// The number 26, in decimal
int decVal = 26;
// The number 26, in hexadecimal
int hexVal = 0x1a;
// The number 26, in binary
int binVal = 0b11010;
```

Literalii reali sunt de tip float sau double. Float sunt daca se termina cu F sau f, double sunt daca se termina cu D sau d sau implicit daca nu se termina cu o litera.

Se pot exprima si folosind e sau E (notatia stiintifica)

Ex:

```
double d1 = 123.4;
// same value as d1, but in scientific notation
double d2 = 1.234e2;
float f1 = 123.4f;
```

Literalii char si String pot contine orice caracter Unicode.

Char poate contine un singur caracter, String un sir de caractere.

Caracterele speciale Unicode se obtin cu prefixul \ urmat de codul hexa al caracterului.

Ex:

```
\u0108
```

Literalii char se inconjoara cu ", iar String cu "".

Java permite introducerea intr-un sir de caractere si a unor caractere mai speciale:

- \b – backspace
- \t – tab
- \n – line feed
- \f – form feed
- \r – carriage return
- \" - double quote
- \' - single quote
- \\ - backslash

Ex:

```
char ch = 'A';
String s = "abc";
String s1 = null;
String s2 = "\n\r\"\\u0108";
```

Alti literali speciali:

- **null** – poate fi folosit ca valoare pentru o referinta (ex: String) poate fi asignat oricarei variabile, cu exceptia celor de tip primitiv este folosit in general ca un marker pentru prezenta sau absenta unui obiect
- **class** – adaugand .class la un nume de clasa (ex: String.class) se obtine un obiect de tip Class care se refera la clasa respectiva.

Ex:

```
String s = null;
Class sClass = String.class;
```

Incepand din Java 1.7 se poate folosi caracterul underscore (_) in interiorul unui literal numeric cu scopul de a imbunatati claritatea codului. Valoarea literalului este cea ramasa ignorand underscore-urile.

Ex:

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi = 3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

3.5 Array-uri

Un array (sir) este un obiect de tip container care poate pastra un numar fix de valori de un anume tip.

Lungimea array-ului este stabilita la creare si nu poate fi modificata ulterior.

Array-ul este format din elemente.

Elementele pot fi accesate prin intermediul unui index. Numerotarea indecsilor incepe de la 0, deci elementul al n-lea are indexul n-1

Declararea unui array are forma:

```
int[] array;
```

Tipul este urmat de [] ceea ce specifica faptul ca e vorba de un array.

Dimensiunea array-ului nu are legatura cu tipul sau ci se decide la initializare.

Numele array-ului respecta aceleasi contitii ca numele de variabile.

Este posibila si o constructie de forma `int array[]` dar nu este recomandata.

Declaratia specifica faptul ca exista o variabila numita array care poate referi un array de int, dar momentan aceasta variabila nu refera nimic, valoarea ei este **null**.

Crearea unui array:

```
array = new int[10];
```

In acest moment se alocă memorie pentru un array cu 10 valori de tip int iar variabila **array** va referi respectivul array.

Lipsa initializarii unei astfel de variabile locale duce la o eroare de compilare de forma "Variable may not have been initialized".

Asignarea unei valori unui element din array are forma:

```
array[2] = 200;
```

Pentru a accesa un element se foloseste aceeaasi sintaxa:

```
System.out.println("Element 2 at index 1: " + anArray[1]);
```

Un mod mai simplu de initializare a unui array in momentul in care i se cunosc din start valorile:

```
int[] array = {1, 2, 3, 4, 5};  
char[] chars = {'a', 's', 'm'};
```

In acest caz lungimea array-ului este determinata de numarul de elemente dintre acolade.

Pentru a determina lungimea unui array se poate folosi proprietatea `length` pe care o are fiecare array.

```
System.out.println(array.length);
```

Array-uri multidimensionale

Java nu trateaza in mod special tablourile multidimensionale. Practic o matrice este un array de array-uri.

Din aceasta cauza este posibil sa avem "matrici" cu "randuri" de dimensiuni diferite.

Accesarea unui element al unui array al carui index nu este valid (mai mic ca 0 sau mai mare sau egal cu dimensiunea array-ului) va genera o eroare la rularea programului de forma:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2  
at com.training.basics.arrays.MatrixUneven.main(MatrixUneven.java:13)
```

Eroarea arata clar valoarea indexului cu probleme si linia de cod unde a aparut problema, deci in general citirea ei cu atentie permite repararea rapida a problemei.

Afisarea usoara a unui array

```
System.out.println(Arrays.toString(new int[] { 5, 3, 6, 5 }));
```

va produce un output de forma:

```
[5, 3, 6, 5]
```

Copierea array-urilor

Clasa `System` are o metoda

```
arraycopy(Object src, int srcPos  
          Object dest, int destPos, int length)
```

care permite copierea unei portiuni dintr-un sir in alt sir.

Exercitiu:

Implementati un mic program care copiaza o parte dintr-un `char[]` sursa intr-un `char[]` destinatie si afisati rezultatul.

Manipularea array-urilor

Array-urile sunt un concept foarte important si util in programare.

Java ofera metode utilitare pentru a putea face usor operatiile de baza cu array-uri:

- copiere

- cautare
- comparare a doua array-uri
- umplerea unui array cu o anumita valoare
- sortarea unui array (din versiunea 1.8 exista si posibilitatea unei sortari paralele care foloseste in caz ca sunt disponibile capacitatile multiprocesor ale sistemului)
- afisarea usoara a unui array

Clasa `java.util.Arrays` ofera aceste metode.

Exercitiu: Sa se descopere metodele care implementeaza functionalitatile de mai sus si sa se utilizeze intr-un program care le demonstreaza utilizarea.

Hint: Scriind in Eclipse numele clasei/obiectului urmat de punct si apasand Ctrl-Spatiu se pot vedea toate metodele si variabilele care se pot folosi in contextul respectiv.

3.6 Operatorii

Sunt simboluri speciale care permit efectuarea de operatii asupra a unu, doi sau trei operanzi si returneaza un rezultat. Precedenta operatorilor se refera la ordinea in care se evalueaza operatorii.

Operatorii cu aceeasi precedenta se evalueaza de la stanga la dreapta cu exceptia operatorilor de assignare care se evalueaza de la dreapta la stanga.

Tabelul de mai jos prezinta operatorii ordonati dupa precedenta, iar cei de pe aceeasi linie au aceeasi precedenta.

Operator Type	Category	Precedence
Unary	postfix	<i>expr++ expr--</i>
	prefix	<i>++expr --expr +expr -expr ~ !</i>
Arithmetic	multiplicative	<i>* / %</i>
	additive	<i>+ -</i>
Shift	shift	<i><< >> >>></i>
Relational	comparison	<i>< > <= >= instanceof</i>
	equality	<i>== !=</i>
Bitwise	bitwise AND	<i>&</i>
	bitwise exclusive OR	<i>^</i>
	bitwise inclusive OR	<i> </i>
Logical	logical AND	<i>&&</i>
	logical OR	<i> </i>
Ternary	ternary	<i>? :</i>
Assignment	assignment	<i>= += -= *= /= %= &= ^= = <<= >>= >>>=</i>

Operatorul de assignment simplu

Cel mai folosit operator: "="

Asigneaza valoarea expresiei din dreapta operandului din stanga.

```
int gear = 4;
```

Operatorii aritmetici

+, -, *, /, %

```
int x = 1+2;
int y = x + 4;
```

Operatorii aritmetici pot fi compusi cu cei de asignment. Cele doua constructii de mai jos sunt echivalente si valabile pentru orice operator aritmetic.

```
x += 1
x = x + 1
```

Operatorul + se poate apela si la Stringuri:

```
String sir1 = "abc";
String sir2 = "cde";
String sir3 = sir1+sir2;
// sir3 va avea valoarea "abccde"
```

De notat ca operatorii aritmetici au sens si pentru tipul char.

Operatorii unari

+, -, ++, --, !

Operatorii de incrementare/decrementare se pot aplica inainte sau dupa un operand. Diferenta e ca versiunea (++result) va fi evaluata la valoarea incrementata, iar expresia (result++) va fi evaluata la valoarea originala. Diferenta e importanta cand operatorul e folosit in expresii complexe. Dar in general nu se recomanda folosirea acestor operatori atunci cand interpretarea corecta a codului nu e foarte clara.

Exercitiu:

Ce va afisa urmatorul program?

```
public class PrePostDemo {
    public static void main(String[] args) {
        int i = 3;
        i++;
        System.out.println(i);
        ++i;
        System.out.println(i);
        System.out.println(++i);
        System.out.println(i++);
        System.out.println(i);
    }
}
```

Operatorii de egalitate si relationali

==, !=, <, >, <=, >=

Determina daca un operand este mai mare, mai mic, egal sau diferit cu celalalt operand.

Atentie, egalitatea se testeaza cu ==, nu cu =

Rezultatul verificarii este un boolean (true/false)

Operatorii conditionali

&&, ||

Realizeaza operatia AND, respectiv OR logic.

Sunt operatori cu "scurtcircuitare" in sensul ca a doua expresie se evalueaza doar daca e nevoie.

Alt operator conditional: operatorul ternar ?: are rolul unei prescurtari de if-then-else.

```
public class ConditionalDemo2 {
    public static void main(String[] args) {
        int value1 = 1;
        int value2 = 2;
        int result;
```

```

boolean someCondition = true;
result = someCondition ? value1 : value2;

System.out.println(result);
}
}

```

Comparatorul de tip instanceof

Compara daca un obiect este de un anumit tip.

null este considerat ca nu e instanceof de nimic (null instanceof orice returneaza false)

```

public class Instanceof {
    public static void main(String[] args) {
        String s = "";
        System.out.println(s instanceof String);
        Object s2 = null;
        System.out.println(s2 instanceof String);
        s2 = new Object();
        System.out.println(s2 instanceof String);
    }
}

```

Operatorii pe biti

```

& - AND
| - OR
^ - XOR
<< - shift left
>> - shift right
>>> - shift right with zero insertion

```

```

public class BitDemo {
    public static void main(String[] args) {
        int bitmask = 0x000F;
        int val = 0x0022;
        System.out.println(val & bitmask);
        System.out.println(val | bitmask);
        System.out.println(val ^ bitmask);
        val = -10;
        System.out.println(Integer.toBinaryString(val));
        System.out.println(val >> 2);
        System.out.println(Integer.toBinaryString(val >> 2));
        System.out.println(val >>> 2);
        System.out.println(Integer.toBinaryString(val >>> 2));
        System.out.println(val << 2);
        System.out.println(Integer.toBinaryString(val << 2));
    }
}

```


3.7 Expresii

O expresie este formata din variabile, operatori si apeluri de metode care returneaza o valoare.

Tipul de date returnat de expresie depinde de elementele folosite in expresie.

O expresie de tipul :

```
int x=0;
```

Returneaza un int cu valoarea 0 pentru ca operatorul de assignment.

Returneaza o valoare de acelasi tip cu tipul operandului din stanga.

Expresiile pot fi compuse din mai multe expresii mai mici atata timp cat tipul operanzilor permite efectuarea operatiilor dintre ei.

Ordinea operatiilor este data de precedenta operatorilor.

```
x + y / 100
```

Pentru a afecta aceasta ordine se folosesc parantezele rotunde

```
(x + y) / 100
```

Pentru claritatea codului se recomanda folosirea parantezelor chiar si in situatii cand ordinea operatiilor va dicta acelasi rezultat.

```
x + (y / 100)
```

Desi limbajul permite se recomanda ca expresiile sa nu fie extrem de complicate astfel incat citirea codului sa fie cat mai usoara. Ex:

```
temp = y / 100
```

```
z = x + temp
```

3.8 Statement-uri

Un statement e echivalentul propozitiilor din limbajul natural.

Un statement formeaza o unitate de executie.

Statement-urile se separa cu ;

Statement-uri de tip expresie

Ex:

expressii de assignment

++, -

apeluri de metode

expresii de creare de obiecte

```
// assignment statement
```

```
aValue = 8933.234;
```

```
// increment statement
```

```
aValue++;
```

```
// method invocation statement
```

```
System.out.println("Hello World!");
```

```
// object creation statement
```

```
Bicycle myBike = new Bicycle();
```

Statement-uri de declaratie

```
// declaration statement
```

```
double aValue = 8933.234;
```

Statement-uri de control al flow-ului – ceva mai tarziu.

3.9 Blocuri

Un grup de zero sau mai multe statementuri grupate intre acolade.

Un bloc poate fi folosit oriunde este permis un statement.

```
public class BlockDemo {
    public static void main(String[] args) {
        boolean condition = true;
        if (condition) { // begin block 1
            System.out.println("Condition is true.");
        } // end block one
        else { // begin block 2
            System.out.println("Condition is false.");
        } // end block 2
    }
}
```

3.10 Instructiuni de control al flow-ului

Instructiunile sunt executate in mod normal de sus in jos in ordinea din cod. Instructiunile de control al flow-ului permit intreruperea flow-ului normal si ramificare lui in functie de anumite conditii.

3.10.1 Instructiunea if

Permite programului sa execute o anumita sectiune de cod doar daca o anumita conditie este adevarata.

Expresia din paranteze trebuie sa se evalueze la un boolean. Nu e ca in C unde se poate scrie ceva de tipul if (1) sau if (0).

Ex:

daca am vrea sa afisam un String doar daca este diferit de null:

Permite programului sa execute o anumita sectiune de cod doar daca o anumita conditie este adevarata. Expresia din paranteze trebuie sa se evalueze la un boolean. Nu e ca in C unde se poate scrie ceva de tipul if (1) sau if (0). Ex: daca am vrea sa afisam un String doar daca este diferit de null:

```
if (s != null) {
    System.out.println(s);
}
```

Chiar daca acest statement s-ar putea scrie si asa:

```
if (s != null)
    System.out.println(s);
```

se recomanda intotdeauna folosirea acoladelor chiar daca exista un singur statement in if. Motivul e simplu, si anume ca adaugarea unei noi linii in if poate crea confuzie.

3.10.2 Instructiunea if-else

Permite in plus fata de un if simplu definirea unei sectiuni de cod care sa se execute in caz ca expresia testata e falsa.

Ex:

la exemplul de mai sus daca vrem sa afisam "—" daca sirul e nul:

```

if (s != null) {
    System.out.println(s);
} else {
    System.out.println("--");
}

```

Pe ramura de else se pot adauga si noi if-uri pentru a testa conditii multiple:

```

public class ElseIf {
    public static void main(String[] args) {
        int nota = 9;
        if (nota >= 9) {
            System.out.println("Foarte bine");
        } else if (nota >= 8) {
            System.out.println("Bine");
        } else if (nota >= 6) {
            System.out.println("Acceptabil");
        } else if (nota >= 4) {
            System.out.println("Varza");
        } else {
            System.out.println("Praful");
        }
    }
}

```

3.10.3 Instructiunea switch

Spre deosebire de if-then-else care permite ramificarea programului in 2 ramuri, switch permite ramificarea in oricate ramuri in functie de valoarea unei variabile.

Tipul variabilei testate in switch poate fi byte, short, char, int, enumerare (mai tarziu), String.

De observat faptul ca tipul variabilei testate nu poate fi long, float sau double!

```

public class Switch {
    public static void main(String[] args) {
        int day = 3;
        switch (day) {
            case 1:
                System.out.println("Luni");
                break;
            case 2:
                System.out.println("Marti");
                break;
            case 3:
                System.out.println("Miercuri");
                break;
            case 4:
                System.out.println("Joi");
                break;
            case 5:
                System.out.println("Vineri");
                break;
        }
    }
}

```

```

case 6:
    System.out.println("Sambata");
    break;
default:
    System.out.println("Duminica");
}
}
}

```

Codul din interiorul unui switch se numeste bloc de switch. Un statement din blocul de switch poate fi marcat printr-un label case sau default.

Expresia din switch se evalueaza si apoi se sare la primul label case cu valoarea egala cu valoarea expresiei iar executia codului continua de acolo in jos.

Un bloc switch se poate scrie si cu ajutorul blocului if-then-else.

Alegerea e la latitudinea programatorului, dar switch se preteaza cand e vorba de a ramifica programul in functie de valoarea unei expresii, iar if-then-else cand e vorba de a testa un interval de valori cum a fost exemplul de mai sus.

Deoarece toate instructiunile de dupa case-ul corect se vor executa in mod normal e nevoie de o modalitate de a intrerupe blocul switch dupa case-ul potrivit. Pentru asta se foloseste instructiunea break

Ex:

de observat ce se intampla la programul anterior daca scoatem break-urile.

In sectiunea default se intra daca nu s-a intrat in nici un case.

Se pot folosi mai multe label-uri unul dupa altul.

Exercitiu:

Sa se realizeze un program care calculeaza folosind instructiunea switch numarul de zile dintr-o luna dandu-se numarul lunii si anul

```

public class SwitchDemo2 {
    public static void main(String[] args) {

        int month = 2;
        int year = 2000;
        int numDays = 0;

        switch (month) {
            case 1:
            case 3:
            case 5:
            case 7:
            case 8:
            case 10:
            case 12:
                numDays = 31;
                break;
            case 4:
            case 6:
            case 9:
            case 11:
                numDays = 30;

```

```

        break;
    case 2:
        if (((year % 4 == 0) && !(year % 100 == 0)) || (year % 400 == 0))
            numDays = 29;
        else
            numDays = 28;
        break;
    default:
        System.out.println("Invalid month.");
        break;
    }
    System.out.println("Number of Days = " + numDays);
}
}

```

Incepend din Java 1.7 ca expresie in switch se poate folosi si String.

3.10.4 Instructiunea while si do-while

Executa un bloc de cod atata timp cat o conditie este adevarata.

Expresia trebuie sa returneze un boolean.

La fiecare pas se retesteaza conditia si se iese din while cand conditia este falsa.

```

class WhileDemo {
    public static void main(String[] args) {
        int count = 1;
        while (count < 11) {
            System.out.println("Count is: " + count);
            count++;
        }
    }
}

```

O bucla infinita se face cu while(true) do-while: diferenta este ca expresia se testeaza la sfarsitul blocului:

```

class DoWhileDemo {
    public static void main(String[] args) {
        int count = 1;
        do {
            System.out.println("Count is: " + count);
            count++;
        } while (count < 11);
    }
}

```

3.10.5 Instructiunea for

Ofera un mod compact de a itera pe un interval de valori.

```
for (initialization; termination; increment) {  
    statements  
}
```

Sectiunea de initializare initializeaza bucla. Se executa o singura data, la inceput.

Sectiunea termination se executa la fiecare pas si cand e false for-ul se termina.

Sectiunea de incrementare se executa dupa fiecare iteratie

Ex:

```
class ForDemo {  
    public static void main(String[] args) {  
        for (int i = 1; i < 11; i++) {  
            System.out.println("Count is: " + i);  
        }  
    }  
}
```

Variabila i este declarata chiar in sectiunea de initializare . Durata ei de viata este doar pe perioada executiei for-ului.

Se recomanda intotdeauna acest tip de initializare cu exceptia cazului cand este nevoie de variabila si dupa incheierea for-ului.

Cele 3 expresii din for sunt optionale. Se poate crea o bucla infinita astfel:

```
for (;;) {  
}
```

Exista si o forma simplificata pentru iterarea pe Colectii si array-uri (for-each sau enhanced for).

Face codul mai usor de citit si se recomanda utilizarea ei oricand nu e nevoie a cunoaste index-ul iteratiei.

```
class EnhancedForDemo {  
    public static void main(String[] args) {  
        int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
        for (int item : numbers) {  
            System.out.println("Count is: " + item);  
        }  
    }  
}
```

La fiecare iteratie in variabila item se va pune pe rand cate un element din array.

Este aplicabila atunci cand nu ne intereseaza valoarea contorului ci doar sa iteram prin toate elementele.

3.10.6 Instrucțiuni de ramificare

break

Are două forme: cu label sau fără label

Este folosit pe lângă a termina un switch și pentru a termina un for, while sau do-while.

```
class BreakDemo {
    public static void main(String[] args) {

        int[] arrayOfInts = { 32, 87, 3, 589, 12, 1076, 2000, 8, 622, 127 };
        int searchfor = 12;

        int i;
        boolean foundIt = false;

        for (i = 0; i < arrayOfInts.length; i++) {
            if (arrayOfInts[i] == searchfor) {
                foundIt = true;
                break;
            }
        }

        if (foundIt) {
            System.out.println("Found " + searchfor + " at index " + i);
        } else {
            System.out.println(searchfor + " not in the array");
        }
    }
}
```

break fără label termină primul loop care îl conține un break cu loop poate termina și un loop exterior celui care îl conține.

```
class BreakWithLabelDemo {
    public static void main(String[] args) {

        int[][] arrayOfInts = { { 32, 87, 3, 589 }, { 12, 1076, 2000, 8 }, { 622, 127, 77, 955 } };
        int searchfor = 12;

        int i;
        int j = 0;
        boolean foundIt = false;

        search: for (i = 0; i < arrayOfInts.length; i++) {
            for (j = 0; j < arrayOfInts[i].length; j++) {
                if (arrayOfInts[i][j] == searchfor) {
                    foundIt = true;
                    break search;
                }
            }
        }
    }
}
```

```

}

if (foundIt) {
    System.out.println("Found " + searchfor + " at " + i + ", " + j);
} else {
    System.out.println(searchfor + " not in the array");
}
}
}

```

continue

Instructiunea continue termina doar iteratia curenta si sare la inceputul loop-ului.

Are de asemenea o forma fara label si una cu label.

Forma fara label termina iteratia care contine instructiunea, cea cu label poate termina o iteratie care contine iteratia curenta.

return

Instructiunea return termina metoda curenta si executia continua dupa linia la care aceasta metoda a fost invocata.

Are doua forme, una care returneaza o valoare, si una care nu returneaza nici o valoare.

Daca tipul returnat de metoda este void atunci metoda nu trebuie sa returneze nici o valoare.

Altfel trebuie returnat un tip de data corespunzator cu tipul din declaratia metodei.

4 Clase si obiecte

4.1 Clase

Exemplu de clasa si subclasa:

```
public class Bicycle {

    // clasa Bicycle are trei campuri
    public int cadence;
    public int gear;
    public int speed;

    // clasa Bicycle are un constructor
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    // clasa Bicycle are patru metode
    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public void applyBrake(int decrement) {
        speed -= decrement;
    }

    public void speedUp(int increment) {
        speed += increment;
    }
}

// clasa MountainBike extinde Bicycle deci ii mosteneste campurile si metodele
public class MountainBike extends Bicycle {

    // clasa MountainBike are un camp
    public int seatHeight;

    // clasa MountainBike subclass are un constructor
    public MountainBike(int startHeight, int startCadence, int startSpeed, int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }
}
```

```

    }

    // clasa MountainBike are o metoda
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }
}

```

4.1.1 Declararea claselor

Declaratia unei clase are forma:

```

class NumeClasa {
    // campuri
    // constructori
    // metode
}

```

Corpul clasei contine tot codul necesar obiectelor instantiate pe baza clasei respective (constructori pentru initializarea obiectelor, variabile membru care pastreaza starea curenta a obiectului, metode care definesc comportamentul obiectului).

Declaratia de mai sus e cea mai simpla. O versiune mai complexa ar fi:

```

class NumeClasa extends SuperClasa implements OInterfata {
    // ...
}

```

O astfel de declaratie specifica faptul ca **NumeClasa** este o subclasa a clasei **SuperClasa** si implementeaza interfata **Ointerfata**.

In general declaratia unei clase contine:

1. Specificatori de acces (ex: public/private)
2. Numele clasei, care prin conventie incepe cu litera mare si de asemenea fiecare cuvant din nume trebuie scris cu litera mare
3. Numele superclasei in caz ca exista, precedat de cuvantul cheie extends
4. O lista de interfete separate prin virgula, interfete pe care clasa le implementeaza, in caz ca exista
5. Corpul clasei inconjurat de acolade

4.1.2 Declararea variabilelor membru

Intr-o clasa exista mai multe tipuri de variabile:

- variabile membru (campuri)
- variabile declarate in metode (variabile locale)
- variabile declarate in semnaturile metodelor (parametri)

Declaratia campurilor este compusa din:

- zero sau un modificador de acces (ex: public)
- tipul campului
- numele campului

4.1.3 Modificatori de acces

- `public` – campul este accesibil din toate clasele
- `private` – campul este accesibil doar din clasa in care acesta este declarat

In spiritul ascunderii detaliilor de implementare fata de exterior este o practica obligatorie ca toate variabilele membru ale unei clase sa fie declarate `private`, si eventual pentru accesul din exterior la ele sa se foloseasca metode de tip `setter/getter`. Dar nici faptul ca facem `getter/setter` la toate campurile nu inseamna OOP.

Ideal ar fi ca initial toate campurile sa fie `private`, si sa se creeze `setter/getter` doar la campurile la care nu exista alternativa si trebuiesc apelate din exterior, dar numarul lor trebuie sa fie cat mai mic posibil.

Numele variabilelor membru incepe intotdeauna cu litera mica, iar restul cuvintelor din nume trebuie sa inceapa cu litera mare. Ex: `name`, `age`, `sex`, `dateOfBirth`.

4.1.4 Declararea metodelor

```
public void setGear(int newValue) {  
    gear = newValue;  
}
```

Declararea metodelor este compusa din:

- Modificatori de acces (`public`, `private`)
- Tipul returnat – tipul de data returnat de metoda sau `void` daca metoda nu returneaza nici o valoare
- Numele metodei – conventia e ca trebuie sa inceapa cu litera mica iar fiecare cuvant din nume sa inceapa cu litera mare (ex: `run`, `doStuff`, `executeBusinessLogic`). In plus o recomandare solida ar fi ca numele metodei sa inceapa cu un verb, pentru ca intotdeauna o metoda trebuie sa faca ceva.
- O lista separata prin virgula de parametri precedati de tipul lor, lista inconjurata de paranteze rotunde `()`. Chiar daca nu exista parametri parantezele trebuie sa apara
- O lista de exceptii – mai tarziu
- Corpul metodei intre acolade

Semnatura unei metode este compusa din numele acesteia si tipurile parametrilor.

Numele metodelor trebuie sa respecte aceleasi conventii ca si numele parametrilor, doar ca primul sau singurul cuvant din numele metodei ar trebui sa fie un verb care descrie ce face metoda. In mod normal numele metodelor ar trebui sa fie unic in cadrul unei clase.

Totusi metodele se pot supraincarca (`overload`) putand avea acelasi nume daca difera numarul si/sau tipurile parametrilor.

4.1.5 Supraincarcarea metodelor

```
public class Drawer {  
    public void draw(String s) {  
    }  
  
    public void draw(int i) {  
    }  
}
```

```

public void draw(double f) {
}

public void draw(int i, double f) {
}
}

```

Diferenta intre metode se face prin intermediul numarului si tipurilor parametrilor. Un apel de ex. `draw("abc")` va apela metoda `draw(String s)`, iar un apel de forma `draw(10)` va apela metoda `draw(int x)`.

Compilatorul nu face diferenta intre metode pe baza tipului returnat.

Acesta nu face parte din semnatura metodei.

Nu se recomanda folosirea excesiva a supraincercarii metodelor deoarece poate face urmarirea programului dificila.

4.1.6 Constructori

O clasa poate contine metode mai speciale numite constructori al caror rol este de a initializa obiectele de tipul clasei respective.

Constructorii:

- Au acelasi nume cu numele clasei
- Nu au tip returnat
- Nu se pot apela explicit ci doar prin intermediul cuvintului cheie `new`

Exemplu:

```

public Bicycle(int startCadence, int startSpeed, int startGear) {
    gear = startGear;
    cadence = startCadence;
    speed = startSpeed;
}

```

Pentru a crea o instanta de `Bicycle` folosind acest constructor se foloseste urmatorul cod:

```
Bicycle bike = new Bicycle(30, 0, 8);
```

Supraincercarea se aplica la fel si la constructori, deci pot exista mai multi constructori intr-o clasa diferentiati prin lista de parametri.

Daca nu se furnizeaza nici un constructor compilatorul creeaza automat un constructor fara parametri care permite instantierea clasei.

Daca se furnizeaza insa un constructor atunci compilatorul nu va mai crea acest constructor default si instantierea clasei se va putea realiza numai prin intermediul constructorilor declarati.

4.1.7 Pasarea parametrilor

Parametrii se refera la lista de variabile din declararea metodei. Argumentele sunt valorile actuale pasate atunci cand metoda este invocata. Argumentele folosite la apelul unei metode trebuie sa se potriveasca cu parametrii declarati atat la tip cat si la ordine.

Metode cu numar variabil de parametri O metoda poate avea un numar variabil de parametri. Pentru asta in declaratia metodei se foloseste "..." dupa tipul ultimului parametru.

Ex:

```
public void method(String... params) {  
}
```

O astfel de metoda va putea fi apelata astfel:

```
method("sir 1")
```

sau

```
method("sir 1", "sir 2", "sir 3")
```

etc... Parametrii primiti vor fi pusi intr-un array de String in cazul acesta si vor putea fi accesati ca orice array, prin indicele lor (params[0], params[1], etc...).

Pot exista metode atat cu parametri normali cat si variabili, dar parametrii variabili trebuie sa se afle pe ultima pozitie in declaratia metodei.

Numele parametrilor Numele parametrilor trebuie sa respecte aceleasi conventii ca si orice variabila.

Practica din C de prefixare a numelui parametrilor cu p este total nerecomandata in Java.

Un parametru poate avea acelasi nume cu o variabila membru a clasei, situatie in care se spune ca parametrul "umbreste" campul respectiv (shadow the field).

Este o practica nerecomandata cu exceptia constructorilor si a metodelor care seteaza un anumit camp, iar in acest caz se va folosi cuvantul cheie this (despre care se va vorbi ulterior).

Pasarea tipurilor primitive Argumentele primitive sunt pasate ca valoare.

Toate modificarile asupra valorilor parametrilor se vor vedea doar in cadrul metodei apelate.

Cand aceasta se incheie parametrii au aceeasi valoare cu cea dinainte de apel.

Ex:

```
public class PassPrimitiveByValue {  
  
    public static void main(String[] args) {  
  
        int x = 3;  
  
        // invoke passMethod() with  
        // x as argument  
        passMethod(x);  
  
        // print x to see if its  
        // value has changed  
        System.out.println("After invoking passMethod, x = " + x);  
  
    }  
}
```

```

// change parameter in passMethod()
public static void passMethod(int p) {
    p = 10;
}
}

```

Pasarea parametrilor de tip referinta Si tipurile referinta sunt pasate tot prin valoare. La intoarcerea din apelul metodei referinta va referi exact acelasi obiect. Totusi se vor vedea modificarile asupra variabilelor membru ale obiectului pasat. Ex:

```

public class Circle {
    public void moveCircle(Circle circle, int deltaX, int deltaY) {
        // code to move origin of circle to x+deltaX, y+deltaY
        circle.setX(circle.getX() + deltaX);
        circle.setY(circle.getY() + deltaY);

        // code to assign a new reference to circle
        circle = new Circle(0, 0);
    }
}

```

4.2 Obiecte

Un program Java obisnuit creaza multe obiecte care interactioneaza prin apeluri de metode. Prin aceste interactiuni intre obiecte programul isi executa sarcina. Cand un obiect si-a terminat treaba resursele ocupate de acesta pot fi reciclate pentru a fi folosite de alte obiecte.

4.2.1 Crearea obiectelor

Crearea unui obiect se face prin intermediul cuvantului cheie new.

Ex:

```
String s = new String("abc");
```

Apelul de mai sus face practic 3 lucruri:

- Declarare – se declara o variabila numita s de tip String
- Instantiere – se creeaza un obiect de tip String si referinta catre el se plaseaza in variabila s
- Initializare – se initializeaza variabilele membru relevante pentru obiectul nou creat, prin intermediul apelului constructorului clasei String, in cazul acesta valoarea curenta a String-ului, care este "abc".

4.2.2 Declararea variabilelor de tip referinta

Are forma

```
TipReferinta numeVariabila;
```

Asta va informa compilatorul ca exista o variabila numita `numeVariabila` si ea poate sa refere un obiect de tip `TipReferinta`.

O astfel de declaratie poate exista si sub forma de mai sus. In acest caz variabila nu va pointa catre nici un obiect pana in momentul cand i se va asina din cod o valoare.

Instantierea unei clase

Operatorul `new` alocă memorie pentru obiectul nou creat si returneaza o referinta catre el.

Totodata va apela si constructorul necesar pentru initializarea obiectului.

Singurul argument al lui `new` este un apel de constructor.

Referinta returnata de `new` in mod normal se va atribui unei variabile.

Ex:

```
s = new String("abc");
```

Si o constructie de tipul:

```
new String("abc");
```

este posibila, dar in acest caz valoarea returnata de `new` nu se va atribui nici unei variabile, deci practic se va pierde.

4.2.3 Initializarea obiectelor

Ex:

```
public class Point {
    public int x = 0;
    public int y = 0;

    // constructor
    public Point(int a, int b) {
        x = a;
        y = b;
    }
}
```

```
public class Rectangle {
    public int width = 0;
    public int height = 0;
    public Point origin;

    // four constructors
    public Rectangle() {
        origin = new Point(0, 0);
    }

    public Rectangle(Point p) {
        origin = p;
    }
}
```

```
public Rectangle(int w, int h) {
    origin = new Point(0, 0);
    width = w;
    height = h;
}

public Rectangle(Point p, int w, int h) {
    origin = p;
    width = w;
    height = h;
}

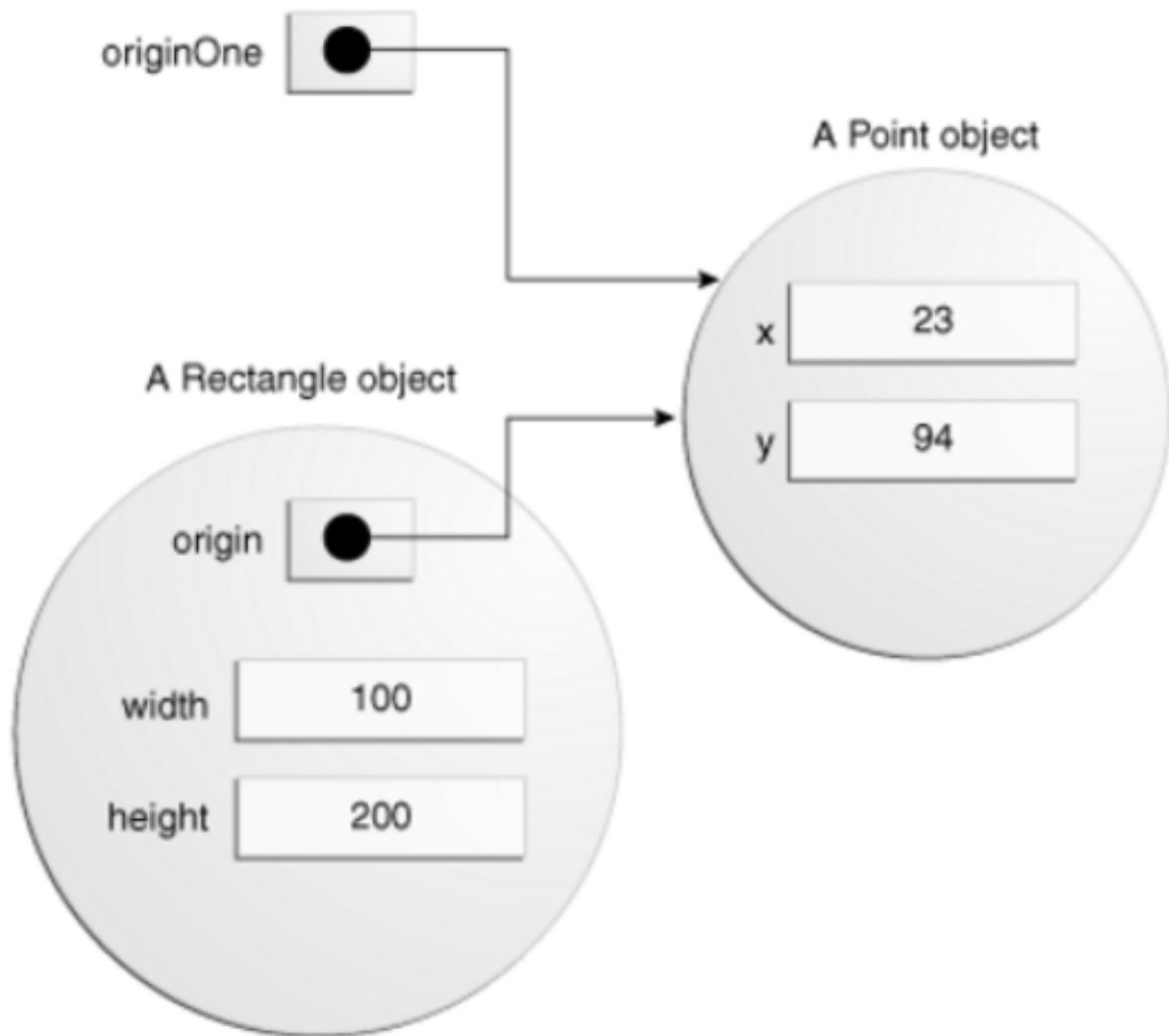
// a method for moving the rectangle
public void move(int x, int y) {
    origin.x = x;
    origin.y = y;
}

// a method for computing the area of the rectangle
public int getArea() {
    return width * height;
}
}
```


Considerand urmatorul cod care foloseste clasele de mai sus:

```
Point originOne = new Point(23, 94);  
Rectangle rectOne = new Rectangle(originOne, 100, 200);
```

reprezentarea din memorie a situatiei de mai sus este:



4.2.4 Folosirea obiectelor

Referirea la variabilele membru Din interiorul clasei in care sunt definite variabilele membru se acceseaza direct prin numele lor.

Din exterior insa pentru a ne referi la o variabila membru publica este necesar sa spunem inainte obiectul a carei variabila membru vrem s-o accesam, si va fi necesara o constructie de forma:

```
referintaObiect.numeVariabila
```

Daca variabila respectiva este **private** ea nu va putea fi accesata din exterior.

Apelul metodelor unui obiect La fel ca in cazul variabilelor membru si metodele necesita o referinta la un obiect pentru a putea fi apelate din exterior.

Apelul unei metode a unui obiect va avea forma:

```
referintaObiect.numeMetoda(listaParametri)
```

4.2.5 Garbage Collector-ul

Spre deosebire de alte limbaje (ex: C, C++) unde memoria trebuie eliberata explicit cand nu mai e nevoie de un obiect, in Java acest lucru cade in sarcina masinii virtuale.

Nu exista practic nici o modalitate de a sterge explicit un obiect.

Un obiect este eligibil de a fi sters automat de masina virtuala in momentul cand nu mai exista nici o referinta in program catre el.

```
public class GC {
    public static void main(String[] args) {
        String s1 = "abc";
        String s3 = method();
        s1 = s3;
        for (int i = 0; i < 10; i++) {
            String s4 = "" + i;
            System.out.println(s4);
        }
        s3 = null;
    }

    private static String method() {
        String s2 = "x";
        s2 = "y";
        return s2;
    }
}
```

4.2.6 Returnarea unei valori dintr-o metoda

Apelul unei metode se incheie in unul din cazurile:

- Se incheie toate instructiunile din ea
- Executia ajunge la o instructiune return
- Metoda arunca o exceptie (de discutat mai tarziu)

Tipul returnat de o metoda se specifica in declaratia metodei.

In codul metodei pentru a returna valoarea se foloseste instructiunea return.

O metoda care declara ca returneaza void nu trebuie sa returneze nici o valoare. Nu e nevoie nici macar sa contina un return dar poate sa-l contina in caz ca se doreste iesirea din metoda intr-un loc mai special din cod.

O metoda care returneaza ceva diferit de void este obligatoriu sa contina cel putin un return urmat de valoarea returnata.

Ex:

```
return returnValue;
```

Tipul valorii returnate trebuie sa coincida cu tipul din declaratia metodei.

4.2.7 Returnarea unei referinte dintr-o metoda

Regulile sunt aceleasi ca la returnarea unei valori doar ca o metoda poate sa returneze un obiect de tipul specificat in declaratie sau de o subclasa a tipului respectiv.

De asemenea exista posibilitatea ca o metoda sa returneze `null` in loc de o referinta (Nu si daca tipul returnat e un tip primitiv!).

4.2.8 Cuvantul cheie `this`

In interiorul unei metode a unei clase sau al unui constructor `this` este o referinta la obiectul curent – obiectul a carui metoda sau constructor este apelat.

Este posibila referirea la orice variabila membru sau metoda prin intermediul lui `this`.

Cel mai comun motiv pentru utilizarea lui `this` este cand un camp este "umbrat" de un parametru.

Ex (fara `this`):

```
public class Point {
    public int x = 0;
    public int y = 0;

    // constructor
    public Point(int a, int b) {
        x = a;
        y = b;
    }
}
```

Ex (cu `this`, varianta preferata) :

```
public class Point {
    public int x = 0;
    public int y = 0;

    // constructor
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Alta situatie in care se foloseste `this` este cand se apeleaza dintr-un constructor un alt constructor al aceleiasi clase:

```

public class Rectangle {
    private int x, y;
    private int width, height;
    public Rectangle() {
        this(0, 0, 1, 1);
    }

    public Rectangle(int width, int height) {
        this(0, 0, width, height);
    }

    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
}

```

4.3 Controlul accesului la membrii unei clase

Exista doua nivele de la care se poate controla accesul asupra membrilor unei clase:

- la nivel de clasa (public sau package)
- la nivel de membru (public, private, protected, package(default))

O clasa declarata public este vizibila tuturor claselor de oriunde. O clasa fara modificador de acces (acces package) va fi vizibila doar in package-ul ei.

Atat la nivel de clasa cat si de membru nu se recomanda folosirea nivelului de acces package.

La nivel de membri:

- public – membrul va fi accesibil de oriunde
- private – membrul va fi accesibil doar din propria clasa
- protected – membrul va fi accesibil din package si din subclase
- package(default) - membrul va fi accesibil in clasa si in package (nerecomandat)

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default (no modifier)	Y	Y	N	N
private	Y	N	N	N

In spiritul incapsularii, la alegerea specificatorilor de acces ar trebui urmate urmatoarele reguli:

- folositi cel mai restrictiv nivel de acces posibil pentru un membru
- folositi private daca nu aveti un motiv foarte serios sa nu
- evitati complet campurile public cu exceptia constantelor. Campurile publice limiteaza flexibilitatea la modificarile ulterioare ale codului

4.4 Membrii statici

4.4.1 Variabile statice

Membrii statici apartin clasei si nu unei instante anume.

Pentru o clasa Person de exemplu numele, prenumele, varsta, cnp sunt proprietati diferite pentru fiecare obiect in parte.

Dar daca am vrea o proprietate care sa fie comuna tuturor persoanelor ar trebui sa o declaram statica.

Orice obiect poate accesa o variabila/metoda care apartine clasei dar acestea pot fi accesate si fara a fi nevoie de o instanta a unei clase, caz in care accesul la membru se va face prin intermediul numelui clasei:

Ex:

```
public class Person {  
    public static int personCounter = 0;  
  
    public Person() {  
        Person.personCounter++;  
    }  
}
```

4.4.2 Metode statice

La fel ca in cazul variabilelor pot fi declarate si metode care apartin unei clase si nu unei instante anume.

Accesarea lor se face ca in cazul variabilelor cu ajutorul numelui clasei

```
ClassName.methodName(args);
```

Metodele statice se pot apela si prin intermediul unui obiect, dar nu este recomandata aceasta practica. Nu sunt permise toate combinatiile de membri statici si ne-statici:

- Metodele ne-statice pot sa acceseze membrii ne-statici
- Metodele ne-statice pot sa acceseze membrii statici
- Metodele statice pot sa acceseze membrii statici
- Metodele statice NU pot sa acceseze membrii ne-statici (acestia pot fi accesati doar prin intermediul unei instante, iar o metoda statica nu apartine nici unei instante)

4.5 Constante

Pentru a defini o constanta, se foloseste combinatia static final.

Static leaga constanta de o clasa si nu o instanta anume, iar final specifica faptul ca valoarea constantei nu va mai fi modificata ulterior.

Ex:

```
static final double PI = 3.1415;
```

4.6 Initializarea campurilor statice

Pentru variabilele membru ne-statice initializarea se face in constructor.

Variabilele statice apartin clasei si nu pot fi legate de o anumita instanta.

Una din posibilitati e initializarea pe loc:

```
public static int initialCapacity = 10;
```

Dar daca initializarea e ceva mai complexa se poate opta pentru un bloc static:

```
static {  
    instructiuni;  
}
```

O clasa poate avea oricate blocuri statice. Ele se vor executa in ordinea in care apar in cod.

O alta alternativa este o metoda statica:

```
class Whatever {  
    public static varType myVar = initializeClassVariable();  
  
    private static varType initializeClassVariable() {  
  
        // initialization code goes here  
    }  
}
```

Avantajul acestei abordari e ca metoda respectiva poate fi reutilizata.

4.7 Initializarea campurilor ne-statice

La fel ca blocul static exista si o alternativa ne-statica in care codul prins intre acolade direct in interiorul unei clase se va executa la fiecare creare a unui nou obiect.

```
class MyClass {  
    {  
        // non static init block  
    }  
}
```

4.8 Clase nested

Java permite definirea unei clase in interiorul altei clase. O astfel de clasa se numeste nested class (clasa incuibata).

Ex:

```
class OuterClass {  
    ...  
    class NestedClass {  
        ...  
    }  
}
```

Clasele nested se impart in doua categorii:

- static nested classes
- inner classes (non-static nested classes)

```
class OuterClass {  
    ...  
    static class StaticNestedClass {  
        ...  
    }  
    class InnerClass {  
        ...  
    }  
}
```

O clasa nested este un membru al clasei in care se afla.

Clasele inner au acces la alti membri ai clasei parinte chiar daca sunt declarate private.

Clasele nested statice nu au acces la membrii clasei parinte.

4.8.1 Scopul claselor nested

- un mod de a grupa impreuna clasele care sunt folosite doar intr-un singur loc. Daca o clasa este folosita intotdeauna de o singura clasa atunci pare logic sa includa in interiorul clasei respective pentru a le tine impreuna
- mareste incapsularea. Ex: avand doua clase A si B iar B necesita acces la membrii lui A care altfel ar fi private. Din cauza nevoii de acces membrii respectivi ar trebui sa fie publici daca nu se folosesc clasele nested.
- Dezavantajul este ca in general duc la un cod mai complicat de inteles, fapt pentru care le vom folosi foarte rar si doar in cazurile cand folosirea lor prezinta beneficii clare.

4.8.2 Clase nested statice

O clasa statica nested este asociata cu clasa parinte. Se acceseaza prin numele clasei in care se gasesc: Ex:

```
// OuterClass.StaticNestedClass
```

```
OuterClass.StaticNestedClass nestedObj = new OuterClass.StaticNestedClass();
```

4.8.3 Inner classes

Se asociaza cu o instanta a unei clase parinte.

Are acces direct asupra metodelor si campurilor obiectului.

Pentru ca este asociata cu o instanta anume nu poate declara membri statici.

Obiectele care sunt instante ale unei astfel de clase exista doar in cadrul unei instante a clasei parinte.

Pentru a instantia o clasa inner trebuie intai sa existe o instanta a clasei parinte. Ex:

```
OuterClass.InnerClass innerObj = outerObj.new InnerClass();
```

Exista doua tipuri speciale de clase inner:

- clase locale
- clase anonime

4.8.4 Conflicte de nume

Pot exista conflicte de nume in cadrul claselor nested. Acestea se rezolva ca in exemplul de mai jos.

```
public class ShadowTest {

    public int x = 0;

    class FirstLevel {

        public int x = 1;

        void methodInFirstLevel(int x) {
            System.out.println("x = " + x);
            System.out.println("this.x = " + this.x);
            System.out.println("ShadowTest.this.x = " + ShadowTest.this.x);
        }
    }

    public static void main(String... args) {
        ShadowTest st = new ShadowTest();
        ShadowTest.FirstLevel fl = st.new FirstLevel();
        fl.methodInFirstLevel(23);
    }
}
```

4.8.5 Clase anonime

Permit declararea si instantierea unor clase in acelasi loc.

Sunt anonime pentru ca nu au un nume.

Ex:

```
public class NoLambdaMain {
    interface Check {
        boolean test(Student s);
    }

    class Student {
        String name;
        double mark;

        public Student(String name, int mark) {
            this.name = name;
            this.mark = mark;
        }
    }

    public static void main(String[] args) {
        NoLambdaMain.Student s1 = new NoLambdaMain().new Student("Ion", 8);
        NoLambdaMain.Student s2 = new NoLambdaMain().new Student("Gheo", 4);
    }
}
```



```

NoLambdaMain.Student[] students = { s1, s2 };

Check checker = new Check() {
    @Override
    public boolean test(Student s) {
        if (s.mark >= 5) {
            System.out.println(s.name + " passed");
            return true;
        } else {
            System.out.println(s.name + " failed");
            return false;
        }
    }
};

for (Student s : students) {
    checker.test(s);
}
}

```

4.8.6 Lambda expressions

Problema claselor anonime este ca daca implementarea clasei anonime este foarte simpla (ex: o interfata care contine o singura metoda) atunci sintaxa claselor anonime poate sa para neclara.

In astfel de cazuri de fapt se paseaza o functionalitate ca un argument catre o alta metoda.

Expresiile lambda permit tratarea functionalitatii pasate ca un argument, sau codul ca si date.

Ex:

```

public class LambdaMain {
    interface Check {
        boolean test(Student s);
    }

    class Student {
        String name;
        double mark;

        public Student(String name, int mark) {
            this.name = name;
            this.mark = mark;
        }
    }

    public static void main(String[] args) {
        LambdaMain.Student s1 = new LambdaMain().new Student("Ion", 8);
        LambdaMain.Student s2 = new LambdaMain().new Student("Gheo", 4);
    }
}

```

```

LambdaMain.Student[] students = { s1, s2 };

Check checker = s -> {
    if (s.mark > 4) {
        System.out.println(s.name + " passed");
        return true;
    } else {
        System.out.println(s.name + " failed");
        return false;
    }
};

for (Student s : students) {
    checker.test(s);
}
}
}

```

4.9 Enum-uri

Un enum este un tip de date special care permite ca o variabila sa poata lua un set predefinit de valori.

O variabila de tip enum trebuie sa fie egala cu una din valorile predefinite pentru ea (sau null)

Ex: punctele cardinale sau zilele saptamanii

Un enum se defineste folosind cuvantul cheie enum:

```

public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}

```

Un enum se poate folosi intr-un switch:

Ex:

```

public class MainEnum {
    public static void main(String[] args) {
        Day day = Day.MONDAY;

        switch (day) {
            case MONDAY:
                System.out.println("It's monday!");
                break;
            case TUESDAY:
                System.out.println("It's tuesday!");
                break;
            // .....
        }
    }
}

```

Un enum este de fapt o clasa mai speciala.

Ea poate contine metode si campuri ca orice alta clasa.

Compilerul creeaza automat niste metode speciale cand creeaza un enum.

Fiecare enum are o metoda statica `.values()` care returneaza un array care contine toate valorile din enum in ordinea in care au fost declarate.

Toate enum-urile extind implicit clasa `java.lang.Enum`. Din aceasta cauza un enum nu poate extinde o alta clasa.

Fiecare constanta dintr-un enum poate fi privita ca un apel de constructor si este posibil sa i se paseze parametrii.

Ex:

```
public enum Day {
    SUNDAY("Duminica"), MONDAY("Luni"), TUESDAY("Marti"), WEDNESDAY("Miercuri"), THURSDAY("Joi")
    "Vineri"), SATURDAY("Sambata");

    private String translation;

    Day(String translation) {
        this.translation = translation;
    }
}
```

4.10 Adnotari

Adnotarile reprezinta o modalitate prin care se pot adauga informatii speciale intr-un program, informatii care nu afecteaza functionarea normala a codului.

Adnotarile pot fi folosite pentru:

- a oferi informatii pentru compiler: de ex pentru a permite compilerului sa inteleaga o anumita secventa de cod sau sa ignore niste warninguri
- procesarea la compilare sau la deploy: exista tooluri care pot analiza adnotarile si sa genereze cod, fisiere XML, etc...
- procesarea la runtime

4.10.1 Formatul adnotarilor

Cea mai simpla adnotare are forma:

`@Entity`

Semnul care marcheaza o adnotare este `"@"`.

O adnotare poate contine elemente, ex:

```
@SuppressWarnings(value = "unchecked")
void myMethod() { ... }
```

Adnotarile pot fi folosite la declaratii (de clase, campuri, metode).

4.10.2 Adnotările predefinite

@Deprecated

Deprecated indica un element care este învechit, depreciat și nu ar mai trebui folosit chiar dacă încă mai există.

@Override

Override indica faptul că o metodă supradefinește o altă metodă. Scopul lui este de a evita erorile accidentale și se recomandă folosirea lui oricând e cazul.

@SuppressWarnings

Spune compilatorului să ignore anumite warning-uri care ar putea apărea din cauza unui cod cel mai probabil nefinalizat. În general anotarea de SuppressWarnings ar trebui evitată, și warning-urile din cod rezolvate.

@FunctionalInterface

Începând cu Java 8 această adnotare semnalizează faptul că interfata adnotată este o interfață funcțională.

5 Package-uri

Un package reprezintă o grupare de clase înrudite pentru a le oferi protecție la acces și management al spațiului de nume (name space).

Clasele dintr-un package ar trebui să fie înrudite ca funcționalitate

Gruparea într-un package face clasele mai ușor de găsit

Prin numele package-ului se face distincție între două clase numite la fel dar aflate în package-uri diferite.

Se poate face ca tipurile dintr-un package să aibă acces unul la altul în cadrul package-ului dar să fie inaccesibile din exterior (default acces).

5.1 Crearea unui package

Pentru a crea o clasă într-un package prima linie de cod trebuie să fie de forma:

```
package nume.package;
```

Pe disc concret fișierul respectiv va trebui să apară într-un folder:

```
<folderul_radacina_al_claselor>/nume/package
```

Reprezentarea fizică pe disc a unei structuri de package-uri reprezintă o structură de foldere.

Nu se recomandă plasarea claselor direct în radacina structurii de clase (lipsa statement-ului package la începutul clasei).

Fiecare clasă ar trebui să aparțină de un package.

5.2 Denumirea package-urilor

Prin convenție numele package-urilor trebuie să conțină doar litere mici și doar în cazuri excepționale caracterul underscore.

Tot prin convenție companiile folosesc la numele claselor domeniul web inversat, ex:

`www.company.com` și-ar începe package-urile cu:

`com.company...`

Noi ne-am putea începe package-urile cu:

```
com.training.....
```

Am putea continua cu numele proiectului, etc...

```
com.training.demoproject.....
```

5.3 Folosirea membrilor unui package

Tipurile dintr-un package se numesc membrii ai package-ului. Pentru a accesa din exteriorul package-ului un membru public exista urmatoarele variante:

- referinta la membru prin numele sau intreg (fully qualified)
- importarea membrului printr-un statement import
- importarea tuturor membrilor package-ului printr-un statement import

Referirea prin numele intreg

Ex:

```
graphics.Rectangle rect = new graphics.Rectangle();
```

Nu se recomanda decat in cazuri speciale

Importarea membrului

Dupa package se plaseaza un statement import de forma:

```
import graphics.Rectangle;
```

iar apoi clasa se poate folosi direct:

```
Rectangle rect = new Rectangle();
```

Importarea tuturor membrilor package-ului

La fel ca mai sus, doar ca importul importa toate clasele:

```
import graphics.*;
```

Caz in care se pot folosi toate clasele din package:

```
Rectangle rect = new Rectangle();
```

```
Circle circle = new Circle();
```

Pentru a scuti programatorul de a scrie cod in plus java importa automat toate clasele din `java.lang` (clasele de baza Java) si package-ul curent (package-ul in care se gaseste fisierul).

5.3.1 Ambiguitati de nume

Exista posibilitatea ca in urma import-urilor dintr-o clasa sa se importe o clasa cu acelasi nume din package-uri diferite.

Ex:

Clasa `Rectangle` presupunem ca exista in package-ul `graphics`, dar mai exista si o versiune standard in `java.awt`.

Daca avem urmatoarele importuri:

```
import graphics.*;
```

```
import java.awt.*;
```

pentru a face diferenta intre cele doua clase cand vrem sa le folosim trebuie sa folosim fully qualified name.

5.3.2 Import static

În situațiile când este nevoie de acces frecvent la membrii static final ai unei clase (constante) există posibilitatea unui import static pentru a evita folosirea numelui clasei de fiecare dată.

Ex:

```
public static final double PI = 3.141592653589793;
...
double r = Math.cos(Math.PI * theta);
```

Facând un import static de forma:

```
import static java.lang.Math.PI;

sau

import static java.lang.Math.*;
```

codul de mai sus se poate scrie astfel:

```
double r = cos(PI * theta);
```

Sintaxa corectă este `import static` și nu `static import`

Importul static poate duce la probleme de conflicte de nume.

De ex. clasele `Long` și `Integer` au ambele constanta `MAX_VALUE`. Importând static ambele clase va rezulta o eroare de compilare.

Se pot importa static referințe statice, constante statice și metode statice

Nu se pot folosi construcții de tipul `import static java.*`; o astfel de construcție este legală dar nu va căuta mai jos în ierarhie

5.4 Managementul surselor și claselor

Implementările Java se bazează pe structura de foldere a sistemului de operare pentru a reprezenta structura de package-uri deși acest lucru nu este obligatoriu.

Regula e următoarea:

O clasă publică numită `A` trebuie pusă într-un fișier cu exact același nume și extensia `.java` (`A.java`)

Numele fully qualified al clasei și package-ul în care aceasta se găsește trebuie să fie paralele:

Ex:

```
package-ul graphics.Rectangle
folder-ul graphics/Rectangle
```

6 Interfete

Există situații când este important ca programatorii să se pună de acord asupra unui contract care exprimă cum interacționează modulele lor. Fiecare programator ar trebui să nu depindă de modul cum este scris codul celuilalt programator. Astfel de "contracte" sunt de fapt interfețele.

Exemplu:

Să ne imaginăm că vrem să testăm automat codul scris de un număr de studenți. Să presupunem că ei au implementat o metodă care returnează minimul numerelor dintr-un array. Dacă cerința

e de forma "Implementati o metoda care calculeaza minimul elementelor dintr-un array" atunci fiecare programator isi va numi metoda cum va dori, unii probabil vor considera elementele din array de tip int, altii double, la fel poate si cu tipul returnat. Astfel nu exista nici o modalitate viabila de a testa automat codul tuturor studentilor cu un singur program. Pentru a face acest lucru ar fi necesar ca pentru fiecare implementare a fiecarui student sa aflam cum se numeste exact implementarea furnizata de el. Daca in schimb le cerem studentilor sa "implementeze o interfata" care li se ofera, si care e comuna pentru toti atunci ei sunt obligati sa respecte semnatura metodei, si astfel putem lua implementarea fiecarui student si apela exact aceeasi metoda.

In Java o interfata este similara cu o clasa, dar poate contine doar constante, semnături de metode fara corp, metode default, metode statice si tipuri incuivate.

Ex:

```
public interface Minimum {

    // constant declarations, if any

    // method signatures

    // An enum with values RIGHT, LEFT
    int min(int[] arr);
    .....
    // more method signatures
}
```

Pentru a folosi interfata respectiva este necesara o clasa care o implementeaza:

```
public class MinimumImpl implements Minimum {
    int min(int[] arr) {
        int min = arr[0];
        for(int x:arr) {
            if (x < min) {
                min = x;
            }
        }
        return min;
    }
}

// other members, as needed -- for example, helper classes not
// visible to clients of the interface
}
```

6.1 Definirea unei interfete

Declararea unei interfete contine modifcatori de acces, cuvântul cheie interface, numele interfetei, o lista de interfete parinte daca exista vreuna si corpul interfetei.

```
public interface GroupedInterface extends Interface1, Interface2, Interface3 {

    // constant declarations
}
```

```

// base of natural logarithms
double E = 2.718282;

// method signatures
void doSomething (int i, double x);
int doSomethingElse(String s);
}

```

O interfata publica poate fi accesata din orice clasa de oriunde. O interfata care nu e publica poate fi accesata doar din package-ul in care e definita.

O interfata poate extinde mai multe interfete spre deosebire de clase care pot extinde o singura clasa. Corpul unei interfete poate sa contina:

- metode abstracte
- metode default
- metode statice
- constante

Metodele abstracte nu au implementare. Contin doar declaratia metodei.

Metodele default au modifierul default in declaratie, iar metodele statice au cuvantul cheie static in declaratie.

Toate metodele dintr-o interfata sunt implicit publice deci cuvantul cheie public poate fi omis.

Toate constantele sunt implicit public static final si acesti specificatori se pot omite.

6.2 Implementarea unei interfete

Pentru a specifica faptul ca o clasa implementeaza o interfata se foloseste cuvantul cheie implements.

O clasa poate implementa mai multe interfete in acelasi timp, acestea se separa prin virgula.

Prin conventie clauza implements urmeaza dupa clauza extends.

6.3 Folosirea unei interfete ca si un tip

O interfata reprezinta un tip referinta. O variabila care are ca tip o interfata poate fi asignata cu orice obiect de un tip care implementeaza interfata respectiva.

6.4 Metode default. Metode statice

Daca o clasa implementeaza o interfata ea trebuie sa implementeze toate metodele interfetei.

In cazul adaugarii de noi metode in interfata respectiva toate clasele care implementeaza interfata ar fi obligate sa o implementeze. Daca din diferite motive acest lucru nu este necesar atunci metoda respectiva se poate declara default in interfata, i se furnizeaza o implementare in interfata si nu mai trebuie implementata neaparat in clasele care implementeaza interfata.

La fel ca metodele statice din clase se pot defini metode statice in interfete. Respectiva metoda va apartine tipului interfetei respective.

Toate metodele dintr-o interfata sunt implicit public si abstract, chiar daca aceste cuvinte cheie lipsesc din declaratie.

Toate variabilele dintr-o interfata sunt implicit public, static si final.

Pentru ca metodele dintr-o interfata sunt abstracte ele nu pot fi marcate ca final, strictfp sau native.

O interfata poate extinde una sau mai multe interfete

- O interfata poate extinde doar interfete
- O interfata nu poate implementa alta interfata sau clasa

7 Mostenirea

O clasa care mosteneste(extinde) alta clasa se numeste subclasa.

Clasa mostenita de alta clasa se numeste superclasa sau clasa de baza.

Exceptand clasa `Object` in Java orice clasa are o singura superclasa. Daca aceasta nu este specificata explicit, atunci este clasa `Object`.

O subclasa mosteneste toti membrii public, protected ai superclasei.

De asemenea mai mosteneste si membrii package-private daca e in acelasi package cu superclasa.

Constructorii superclasei nu sunt mosteniti dar pot fi invocati din subclasa.

Campurile mostenite pot fi folosite direct ca si cum ar fi fost definite in clasa.

Se poate declara un camp nou in subclasa, ascunzand astfel campul din superclasa (nu se recomanda).

Se pot declara campuri noi in subclasa.

Metodele superclasei se pot invoca direct.

Se poate defini o noua metoda ne-statica in subclasa cu aceeasi semnatura ca in clasa de baza, supradefinind astfel metoda din superclasa (override)

Se poate defini o noua metoda statica in subclasa cu aceeasi semnatura ca in clasa de baza, ascunzand astfel metoda din superclasa

Se pot declara metode noi care nu sunt in superclasa, specializand astfel subclasa.

Se pot scrie constructori noi in subclasa care pot apela constructorii superclasei prin intermediul cuvantului cheie `super`.

7.0.1 Operatorul cast

Prin operatorul de cast (`()`) putem spune ca vrem ca un anumit obiect sa poate fi folosit ca si un alt obiect. Acest lucru este posibil din cauza relatiei de extindere in cadrul ierarhiei de clase.

Ex:

Clasa `String` extinde clasa `Object`.

Daca avem:

```
Object obj = new Object();
String s = new String();
Object ob2 = new String();
Object o = s; // ok, s (String) stie sa faca tot ce ar trebui sa stie sa faca un Object.
String s2 = obj; // nu-i ok, obj e doar un obiect, nu stie sa
// faca ce face un String => eroare de compilare
String s2 = (String) obj; // prin cast fortam assignment-ul, dar codul
// va da eroare la runtime: ClassCastException deoarece Object nu e subclasa a lui String
String s2 = (String) ob2; // ok, pentru ca ob2 e de tip String
```

Castul poate fi facut doar daca obiectul care trebuie "castuit" stie sa faca toate operatiile pe care stie sa le faca tipul la care se face cast. Acest lucru rezulta din ierarhia de clase. O subclasa stie sa faca tot ce face superclasa (si probabil ceva in plus). Reciproc nu e adevarat.

Pentru a evita `ClassCastException` se foloseste operatorul `instanceof`.

7.1 Supradefinirea si ascunderea metodelor

O metoda ne-statica cu acelasi nume, parametri, si tip returnat ca metoda din superclasa supradefineste(overrides) metoda respectiva din superclasa.

Aceasta abilitate permite subclasei sa altereze comportamentul clasei de baza.

O subclasa poate returna si un subtip al tipului returnat de metoda din superclasa.

Cand se supradefineste o metoda se recomanda folosirea anotarii `@Override` in fata metodei pentru a specifica compilatorului faptul ca intentia programatorului este sa supradefineasca o metoda din superclasa. Daca o astfel de metoda nu exista va rezulta o eroare de compilare.

In cazul unei metode statice redefinite in subclasa nu se aplica acelasi mecanism, ci metoda respectiva ascunde metoda din superclasa.

In cazul metodelor abstracte si default din interfete se aplica aceleasi principii ca le metodele ne-statice, dar datorita posibilitatii ca o clasa sa implementeze mai multe interfete trebuie sa existe un mecanism de protectie pentru cazul cand mai multe interfete furnizeaza aceeasi metoda.

Exista doua reguli pentru a preveni aceste conflicte:

- metodele din clasele ne-statice au prioritate fata de metodele din interfete
- metodele deja supradefinite de alte clase din ierarhie sunt ignorate

O metoda supradefinita poate oferi mai mult dar nu mai putin acces prin specificatorii de acces.

7.2 Polimorfismul

Polimorfismul reprezinta capacitatea unei secvente de cod de a se comporta diferit in functie de tipul obiectului a carei metode se apeleaza.

7.3 Ascunderea campurilor

Este posibil desi nu si recomandat sa existe in subclasa un camp cu acelasi nume ca cel din superclasa. In caz ca trebuie apelat cel din superclasa se va folosi cuvantul cheie `super`.

7.4 Cuvantul cheie `super`

Este folosit pentru a putea referi cu el membrii superclasei.

Foarte util atunci cand o metoda supradefineste o metoda din superclasa. Atunci doar cu `super` poate fi invocata metoda din superclasa.

De asemenea constructorii superclasei pot fi invocati cu ajutorul cuvantului cheie `super`.

Daca din constructorul subclasei nu se apeleaza explicit un constructor al superclasei prin intermediul lui `super` se introduce automat un apel la constructorul fara parametru al superclasei. Daca un astfel de constructor nu exista va rezulta o eroare de compilare a carei rezolvare necesita automat un apel la un constructor al superclasei.

7.5 Clasa `Object`

Clasa `Object` este direct sau indirect mostenita de toate clasele din Java.

Din aceasta cauza toate clasele din Java mostenesc metodele clasei `Object`, iar cele mai importante sunt prezentate mai jos:

- Metoda `clone()`
- Metoda `equals()`

- Metoda finalize()
- Metoda getClass()
- Metoda hashCode()
- Metoda toString()

7.6 Cuvantul cheie final

O variabila **final** nu poate fi modificata.

O metoda **final** nu poate fi supradefinita.

O clasa **final** nu poate fi extinsa.

7.7 Cuvantul cheie abstract

O clasa abstracta nu poate fi instantiata. In mod normal o clasa devine abstracta daca are cel putin o metoda abstracta.

O metoda abstracta nu are implementare.

Scopul unei metode abstracte este de a fi implementata (probabil diferit) in subclase.

Metodele ne-stactice din interfete sunt implicit abstracte.

Clase abstracte vs interfete:

clase abstracte ar trebui folosite cand:

- trebuie ca mai multe clase inrudite sa foloseasca bucati de cod comun
- este de asteptat ca subclasele clasei abstracte sa aibe multe campuri si metode comune
- este nevoie in superclasa de campuri ne-stactice

Interfete ar trebui folosite cand:

- este de asteptat ca interfata sa fie implementata de clase fara legatura intre ele (Ex: Serializable, Cloneable, Comparable)
- este nevoie de mostenire multipla.

7.8 Mostenire vs agregare

Este foarte important ca relatiile dintre clase sa fie corecte, sa nu se foloseasca mostenirea atunci cand nu e cazul. Folosirea mostenirii gresit duce la probleme serioase.

Relatiile intre doua clase pot fi de doua tipuri:

- agregare
- mostenire

Tipul de relatie corect se alege raspunzand la intrebarile:

- are un (has a) = agregare
- is a (este un, este un fel de) = mostenire

Exemple de relatii:

- motor – masina
- student – profesor
- student – curs
- student – persoana
- profesor – persoana
- masina – vehicul
- bicicleta – vehicul
- vehicul – persoana

8 Clase pentru manipulat numere si caractere

8.1 Clasele Number

Cand lucram cu numere, cel mai simplu si eficient este sa folosim tipuri primitive.

Exista insa situatii cand in loc de primitive avem nevoie sa folosim obiecte:

- cand lucram cu colectii
- cand avem nevoie ca o variabila numerica sa poata avea si valoarea null care semnifica de ex. ca nu a fost initializata

Pentru asta exista asa-numitele clase "wrapper" care "infasoara" un tip primitiv.

Exista situatii cand compilatorul face wrapping-ul automat – daca folosim un tip primitiv acolo unde se asteapta la un obiect (autoboxing) sau invers (unboxing).

Toate clasele wrapper numerice au ca superclasa clasa abstracta Number.

Mai exista o clasa care "infasoara" tipul boolean si tot ce vorbim despre numere este valabil si pentru ea.

Toate clasele wrapper au o serie de metode importante (ex: pt. clasa Integer):

- `int intValue()` : converteste obiectul apelat la o primitiva
- `int compareTo(Integer anotherInteger)`: compara valoarea obiectului apelat cu alt obiect
- `equals(Object obj)` : verifica daca valoarea obiectului curent este egala cu a obiectului primit ca parametru

In plus mai exista o serie de metode statice importante:

- `decode`
- `parseInt`
- `toString`
- `valueOf`

8.2 Formatarea numerelor

- metoda `System.out.format`
- clasa `DecimalFormat`

8.3 Clase pentru operatii aritmetice complexe

- clasa `Math`

8.4 Problemele cu aritmetica reala de precizie

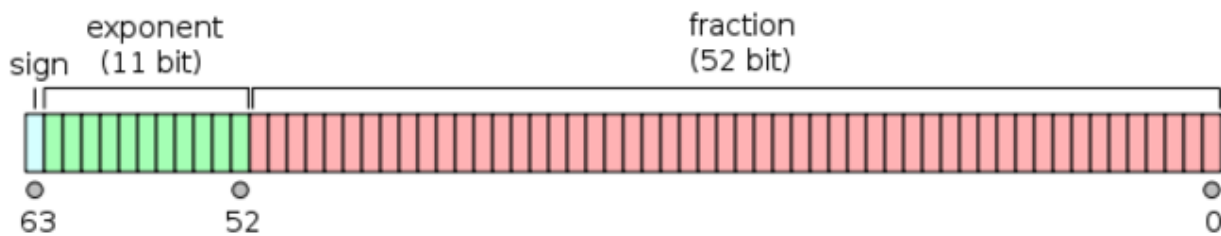
Daca se doresc niste calcule de foarte mare precizie cu numere reale (ex: operatii financiare, software pentru avioane, nave spatiale), atunci trebuie stiut ca aritmetica in Java are mici probleme. Ex:

```
public class FPExample {
    public static void main(String[] args) {
        double d = 0;
        for (int i = 1; i <= 10; i++) {
            d += 0.1;
        }
        System.out.println(d);
    }
}
```

Pentru a rezolva aceasta problema cand e nevoie de precizie se foloseste clasa `BigDecimal`. Ex:

```
public class FPExampleBigDecimal {
    public static void main(String[] args) {
        BigDecimal bd = new BigDecimal("0.0");
        for (int i = 1; i <= 10; i++) {
            bd = bd.add(new BigDecimal("0.1"));
        }
        System.out.println(bd);
    }
}
```

Figure 1: Floating point representation



$$(-1)^{\text{sign}} \left(1 + \sum_{i=1}^{52} b_{52-i} 2^{-i} \right) \times 2^{e-1023}$$

8.5 Caractere

Clasa wrapper pentru tipul primitiv `char` se numeste `Character`.

Metode utile:

- `isLetter`
- `isDigit`
- `isWhitespace`
- `isUpperCase`
- `isLowerCase`
- `toUpperCase`
- `toLowerCase`
- `toString`

Caractere speciale:

- `\t` tab
- `\b` backspace
- `\n` newline
- `\r` carriage return
- `\'` apostrof
- `\"` ghilimele
- `\` backslash

Caracterul care este folosit pentru trecerea la linia urmatoare este diferit in Windows fata de Linux.

Windows: `\n\r`

Linux: `\n`

Din aceasta cauza cand e nevoie de a produce un fisier corect din acest punct de vedere atat pe linux cat si pe windows se foloseste `System.getProperty("line.separator")`.

9 Clase pentru manipulat siruri de caractere

Clasa care se ocupa de sirurile de caractere in java este `String`.

Clasa `String` este imutabila, adica odata ce un obiect `String` a fost creat valoarea lui nu se poate modifica. Clasa `String` are o serie de metode care aparent modifica stringuri. Ele insa nu modifica stringuri ci returneaza tot timpul un nou obiect `String` cu rezultatul operatiei.

Stringurile se pot crea atat in forma clasica, folosind `new` dar si intr-o forma prescurtata.

Lungimea unui `String` se obtine prin intermediul metodei `length()`.

Concatenarea `String`-urilor: metoda `concat` si operatorul `+`.

Formatarea `String`-urilor: metoda `format`.

9.1 Conversia intre numere si siruri

`String`-urile se convertesc in numere prin intermediul metodelor statice de forma `valueOf(String)` ale tipurilor primitive.

Convertirea unui numar in `String`

- `"" + nr`
- `String.valueOf()`
- `WrapperClass.toString()`

9.2 Manipularea caracterelor dintr-un String

- `charAt()`
- `substring()`
- `split()`
- `trim()`
- `toLowerCase()`
- `toUpperCase()`

9.3 Cautarea in String

- `indexOf()`
- `lastIndexOf()`
- `contains()`

9.4 Inlocuirea caracterelor intr-un String

- `replace()`
- `replaceAll()`
- `replaceFirst()`

9.5 Compararea String-urilor si a portiunilor de String-uri

- `endsWith()`
- `startsWith()`
- `compareTo()`
- `compareToIgnoreCase()`
- `equals()`
- `equalsIgnoreCase()`
- `matches()`

9.6 Clasa StringBuilder

Ofera functionalitati similare cu clasa `String` cu diferenta ca nu e imutabila, continutul `String`-ului se poate modifica.

Clasa `StringBuilder` trebuie folosita in momentul cand se creeaza `String`-uri din foarte multe concatenari. In acest caz diferenta de timp si resurse este clar in favoarea `StringBuilder`-ului.

10 Exceptii

10.1 Ce este o exceptie?

Notiunea de exceptie este o scurtatura pentru "eveniment exceptional".

Ca definitie: o exceptie este un eveniment care apare in timpul executiei programului si intrerupe flowul normal al acestuia.

Cand apare o exceptie intr-o metoda, aceasta creeaza un obiect exceptie care contine informatii despre eroare. Acest mecanism poarta numele de aruncare a exceptiei.

Dupa ce exceptia este aruncata sistemul cauta modalitati de a o trata. Posibilele tratari a exceptiei apar in lista ordonata de metode care au fost apelate pentru a ajunge la metoda unde exceptia a aparut. Aceasta lista de metode poarta numele de stiva de apeluri (call stack).

Sistemul cauta in aceasta stiva de apeluri un bloc de cod care poate trata exceptia. Acest bloc de cod poarta numele de exception handler. Cand un handler a fost gasit i se paseaza obiectul exceptie. Un exception handler poate trata o exceptie daca tipul exceptiei corespunde cu cel pe care il poate trata handlerul.

Handler-ul respectiv se spune ca "prinde" exceptia.

Daca nu a fost gasit nici un handler potrivit programul se incheie si afiseaza exceptia ne tratata.

10.2 Cerintele de tratare a exceptiilor

Codul java trebuie sa respecte cerinta de prindere sau specificare:

O secventa de cod care arunca o exceptie trebuie inconjurata cu:

- un statement try care prinde exceptia. Try-ul trebuie sa aibe un catch corespunzator care sa handle- uiasca exceptia.
- o metoda care specifica faptul ca arunca o exceptie cu ajutorul clauzei throws

Daca codul nu respecta cerinta de mai sus nu se va compila.

Excepsiile se impart in exceptii verificate si ne-verificate (checked exceptions si unchecked exceptions)

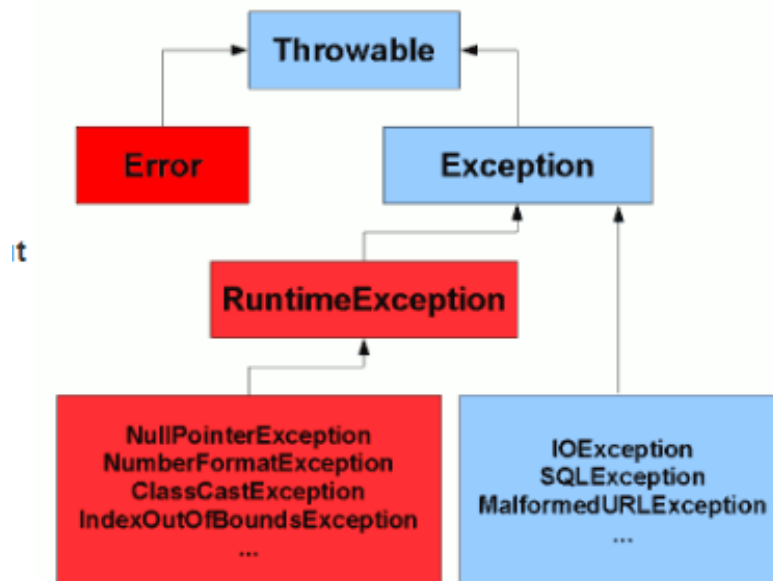
Checked exceptions sunt exceptiile pe care un programator trebuie sa le anticipeze si sa aibe o solutie de tratare a lor. Toate exceptiile sunt checked exceptions cu exceptia Error si RuntimeException si a subclaselor lor.

Unchecked exceptions sunt exceptiile pe care programatorul nu este obligat sa le trateze explicit in cod, desi poate face asta. Ele se impart in erori (subclase ale clasei Error) si runtime exceptions (subclase ale clasei RuntimeException).

Erorile reprezinta probleme grave care nu permit continuarea programului (ex: OutOfMemory, StackOverflow). In mod normal ele duc la intreruperea programului.

Runtime exceptions sunt erori care tin de aplicatie si in general semnifica erori in logica programului (ex: NullPointerException, NumberFormatException)

Figure 2: Java exception hierarchy



10.3 Prinderea si tratarea exceptiilor

10.3.1 Blocul try

Blocul try identifica secventa de cod care poate arunca o exceptie.

10.4 Blocul catch

Blocul catch identifica handlerele exceptiilor care pot sa apara in blocul de try asociat. Argumentul trebuie sa fie o clasa care implementeaza interfata Throwable.

Se pot prinde mai multe tipuri de exceptii intr-un singur bloc incepand cu java 1.7 cu o constructie de forma:

```
catch (IOException|SQLException ex) {  
    logger.log(ex);  
    throw ex;  
}
```

10.5 Blocul finally

Blocul `finally` se executa intotdeauna cand blocul `try` se incheie, si daca apare o exceptie si daca nu. Blocul `finally` este folosit pentru a evita posibilitatea ca eventualul cod de curatare a resurselor din `try` sa fie exclus de un eventual `return`, `continue` sau `break`. E o idee foarte buna ca eventualul cod de curatare sa fie pus in `finally`.

10.6 Blocul try-with-resources

Incepand din Java 1.7 a aparut notiunea de try-with-resources care e un try mai special care se ocupa automat de inchiderea resurselor deschise in timpul blocului try. Pentru a putea folosi astfel de obiecte ele trebuie sa implementeze interfata `Closeable`.

```
static String readFirstLineFromFile(String path) throws IOException {
    try (BufferedReader br =
        new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

10.7 Specificarea exceptiilor aruncate de o metoda

Pentru a specifica exceptiile pe care le poate arunca o metoda se foloseste clauza `throws`

10.8 Aruncarea exceptiilor

Aruncarea exceptiilor se face cu ajutorul clauzei `throw`. Parametrul lui `throw` trebuie sa fie subclasa a clasei `Throwable`. O exceptie contine informatii despre stiva de apeluri si aceasta poate fi consultata prin intermediul metodei `getStackTrace()`

10.9 Crearea claselor de exceptie

Programatorii pot sa-si defineasca propriile clase de exceptie. Clasele de exceptie sunt niste clase normale doar ca mostenesc `Throwable` sau subclasele ei.

10.10 Avantajele exceptiilor

- Separa codul de tratare a erorilor de codul normal => cod mai simplu
- Permite propagarea usoara a erorilor la metodele apelante
- Permite gruparea tipurilor posibile de erori

11 Colectii

11.1 Ce sunt colectiile?

O colectie (container) este un obiect care grupeaza mai multe obiecte la un loc.

Sunt folosite pentru a stoca, regasi si manipula date.

In general reprezinta date care formeaza un grup dupa diferite criterii.

Java Collections Framework reprezinta o arhitectura unificata de reprezentare si manipulare a colectiilor. Contine:

- interfete: tipuri de date abstracte care reprezinta colectii. Interfetele permit manipularea colectiilor independent de implementarea lor
- implementari: implementarile concrete ale interfetelor de mai sus
- algoritmi: metode care rezolva calcule complexe cum ar fi cautarea si sortarea pe obiectele care implementeaza interfețele. Algoritmii sunt polimorfici in sensul ca pot fi folositi pe tipuri diferite de colectii si date

Beneficiile JCF:

- reduc efortul de programare
- cresc viteza si calitatea programului

11.2 Interfete de baza

- **Collection** – radacina ierarhiei de colectii. O colectie reprezinta un grup de elemente.
- **Set** - o colectie care nu poate contine elemente duplicate. Modeleaza conceptul matematic de multime. Ordinea elementelor nu este relevanta.
- **List** – o colectie ordonata de elemente. Listele pot contine elemente duplicate. Intr-o lista se poate controla ordinea elementelor, iar acestea se pot accesa pe baza unui index.
- **Queue** – o coada la care se poate insera pe la un singur capat
- **Deque** – o coada in care se poate insera pe la ambele capete
- **Map** – un obiect care poate asocia chei si valori. Nu poate contine chei duplicate.
- **SortedSet, SortedMap** – versiuni sortate ale Set si Map

11.2.1 Interfata Collection

Este folosita unde este nevoie de maximul de generalitate la pasarea colectiilor.

De exemplu fiecare colectie are un constructor care primeste ca parametru un argument de tip Collection prin care colectia respectiva poate fi initializata cu membrii altei colectii (convertirea unei colectii de la un tip la altul).

Parcurea colectiilor:

- Operatii agregate:

Din Java 1.8

```

myShapesCollection.stream()
    .filter(e -> e.getColor() == Color.RED)
    .forEach(e -> System.out.println(e.getName()));

int total = employees.stream()
    .collect(Collectors.summingInt(Employee::getSalary));

```

- for-each

```

for (Object o : collection){
    System.out.println(o);
}

```

- iteratori

Iteratorul este singura modalitate sigura de modificare a unei colectii in timpul iterarii asupra ei.
Un astfel de cod va produce `ConcurrentModificationException`:

```

public class Main {
    public static void main(String[] args) {
        ArrayList<String> arr = new ArrayList<String>();
        arr.add("a");
        arr.add("b");
        arr.add("c");
        for (String s : arr) {
            arr.remove(s);
        }
    }
}

```

Versiunea corecta:

```

    Iterator<String> iterator = arr.iterator();
    while (iterator.hasNext()) {
        iterator.next();
        iterator.remove();
        // iterator.remove(); IllegalStateException pentru al doilea apel remove()
    }

```

Operatii la nivel de intreaga colectie:

- containsAll
- addAll
- removeAll
- retainAll
- clear

Operatii de interfata cu array-urile

- toArray

11.2.2 Interfata Set

Un **Set** este un **Collection** care nu poate avea elemente duplicate.

Implementarile interfetei **Set**:

- **HashSet**
- **TreeSet**
- **LinkedHashSet**

Operatiile de baza ale interfetei **Set**

- **size**
- **isEmpty**
- **add**
- **remove**
- **iterator**

Operatiile la nivelul intregii colectii:

- **s1.containsAll(s2)** – returneaza true daca s2 este o submultime a lui s1
- **s1.addAll(s2)** – transforma s1 in reuniunea dintre s1 si s2
- **s1.retainAll(s2)** – transforma s1 in intersectia dintre s1 si s2
- **s1.removeAll(s2)** – transforma s1 in diferenta dintre s1 si s2

11.2.3 Interfata List

O lista reprezinta o colectie ordonata de elemente.

O lista poate contine elemente duplicate.

Lista permite urmatoarele operatii:

- acces pozitional (bazat pe indexul elementului in lista) -get, set, add, addAll, remove
- cautare – indexOf, lastIndexOf
- iterare
- extragere a unor portiuni din lista
- operatii algoritmice asupra listei:
 - sort
 - shuffle
 - reverse
 - rotate
 - replaceAll
 - fill
 - copy
 - binarySearch

11.2.4 Interfata Queue

O coada este o colectie care pastreaza niste elemente inainte de procesarea lor.

Fata de un Collection normal are in plus metode de inserare, scoatere si inspectie.

Rar folosita.

```
public interface Queue<E> extends Collection<E> {  
    E element();  
    boolean offer(E e);  
    E peek();  
    E poll();  
    E remove();  
}
```

11.2.5 Interfata Deque

Reprezinta o coada la care se pot insera/scoate elemente la ambele capete.

Foarte rar folosita.

11.2.6 Interfata Map

Un Map este un obiect care mapeaza chei si valori.

Nu poate contine chei duplicate.

Modeleaza conceptul matematic de functie.

Metode importante:

- operatii de baza: put, get, remove, containsKey, containsValue, size, empty
- operatii in grup: putAll, clear
- imagini ale colectiei: keySet, entrySet, values

11.2.7 Ordonarea obiectelor

- Collections.sort()
- Comparable
- Comparators

11.2.8 Interfata SortedSet

Un Set care isi tine elementele ordonate dupa un Comparator.

```
public interface SortedSet<E> extends Set<E> {  
    // Range-view  
    SortedSet<E> subSet(E fromElement, E toElement);  
    SortedSet<E> headSet(E toElement);  
    SortedSet<E> tailSet(E fromElement);  
  
    // Endpoints  
    E first();  
    E last();  
}
```

```

    // Comparator access
    Comparator<? super E> comparator();
}

```

11.2.9 Interfata SortedMap

Un map care isi tine cheile sortate dupa un Comparator.

```

public interface SortedMap<K, V> extends Map<K, V>{
    Comparator<? super K> comparator();
    SortedMap<K, V> subMap(K fromKey, K toKey);
    SortedMap<K, V> headMap(K toKey);
    SortedMap<K, V> tailMap(K fromKey);
    K firstKey();
    K lastKey();
}

```

11.3 Operatii agregate

Considerand clasa:

```

public class Person {

    public enum Sex {
        MALE, FEMALE
    }

    String name;
    LocalDate birthday;
    Sex gender;
    String emailAddress;

    // ...

    public int getAge() {
        // ...
    }

    public String getName() {
        // ...
    }
}

```

Daca am avea o colectie de `Person` pentru a afisa toate numele din ea avem urmatoarele optiuni:

```

for (Person p : roster) {
    System.out.println(p.getName());
}

```



```
roster
    .stream()
    .forEach(e -> System.out.println(e.getName()));
```

11.3.1 Pipeline-uri si stream-uri

Un pipeline e o secventa de operatii agregate. Ex:

```
roster
    .stream()
    .filter(e -> e.getGender() == Person.Sex.MALE)
    .forEach(e -> System.out.println(e.getName()));
```

Un pipeline consta din:

- o sursa (collection, array, canal I/O) ex: roster
- zero sau mai multe operatii intermediare (ex: filter) – care produc un nou stream. Un stream e o secventa de elemente. Spre deosebire de o colectie stream-ul nu stocheaza elementele ci le poarta de la o sursa printr-un pipeline. In exemplul nostru se creaza un stream din colectia roster prin intermediul metodei stream(). Operatia filter produce un nou stream care pastreaza doar elementele care verifica predicatul specificat.
- O operatie terminala. (ex: forEach) care produce un rezultat care nu e un stream (ex: o primitiva, o colectie sau chiar nici o valoare cum e cazul exemplului)

```
double average = roster
    .stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .mapToInt(Person::getAge)
    .average()
    .getAsDouble();
```

Diferente intre operatii agregate si iteratori:

- Iteratia se face intern. In acest caz masina virtuala determina cum va itera colectia, de ex poate optimiza acest proces.
- Operatiile agregate proceseaza elementele dintr-un stream, nu direct din colectie
- Operatiile agregate suporta lambda expressions.

11.4 Implementari

11.4.1 Implementarile interfetei Set

Implementari generale

- HashSet: rapid, dar nu ofera sortare
- TreeSet: ofera sortare

- `LinkedHashSet`: o solutie intermediara, foarte rar folosita

Implementari speciale

- `EnumSet` – un set cu operatii speciale pentru elemente de tip `Enum`
- `CopyOnWriteArraySet` – un set care este implementat cu ajutorul unui array si fiecare modificare asupra setului produce un nou array. Util pentru set-uri modificate rar dar iterate des. Foarte rar folosit.

11.4.2 Implementarile interfetei `List`

Generale

- `ArrayList`: cel mai frecvent utilizat si rapid
- `LinkedList`: se preateaza cand se adauga frecvent elemente la inceput si se sterg frecvent din interior

Speciale

- `CopyOnWriteArrayList`: la fel ca varianta de la `Set`. Foarte rar folosit.

11.4.3 Implementarile interfetei `Map`

Generale

- `HashMap`
- `TreeMap`
- `LinkedHashMap`

Speciale

- `EnumMap`
- `Properties`: permite citirea unui fisier in format cheie-valoare intr-un map

11.4.4 Implementarile interfetei `Queue`

- `LinkedList`
- `PriorityQueue`: o coada care isi tine elementele sortate

11.4.5 Implementarile interfetei `Deque`

- `LinkedList`
- `ArrayDeque`

11.4.6 Implementari wrapper

Delega functionalitatea unei colectii specifice, dar adauga functionalitati in plus peste acestea. Design pattern: decorator.

Astfel de implementari se obtin prin metode statice de tip factory. Toate implementarile se gasesc in clasa Collections.

Wrappere de sincronizare: Adauga sincronizare automata unei colectii. Pentru a garanta accesul sincronizat toate accesele colectiei din spate trebuie sa se realizeze prin intermediul colectiei returnate. Pentru asta o idee buna e de a nu pastra o referinta catre colectia din spate.

```
List<Type> list = Collections.synchronizedList(new ArrayList<Type>());
```

Wrappere ce nu permit modificarea. Acestea practic ascund functionalitate. Intercepteaza operatiile care modifica o colectie si arunca `UnsupportedOperationException`. Utilitate:

- pentru a face o colectie imutabila dupa ce a fost construita
- pentru a permite acces read-only asupra unei colectii.

```
Collections.unmodifiableList(java.util.List<? extends T>);
```

11.5 Algoritmi polimorfici

- sortare (`sort`)
- amestecare (`shuffle`)
- cautare (`binarySearch`)
- manipularea datelor (`reverse`, `fill`, `copy`, `swap`, `addAll`)
- compozitie (`frequency`, `disjoint`)
- gasire min, max (`min`, `max`)

12 Operatii I/O

12.1 Stream-uri I/O

Un stream I/O reprezinta o sursa de intrare sau o destinatie de iesire. (Ex: fisiere, device-uri, alte programe, siruri din memorie). Stream-urile permit tipuri diferite de date, ex: bytes, tipuri primitive, caractere sau obiecte).

12.1.1 Stream-uri de bytes

Toate clasele care manipuleaza streamuri de bytes descend din `InputStream` si `OutputStream`.

Ex: `FileInputStream`, `FileOutputStream`.

Ex: Citirea unui fisier binar si copierea lui in alt fisier.

! Intotdeauna inchideti stream-urile la sfarsit !

12.1.2 Stream-uri de caractere

Clasele care manipuleaza stream-uri de caractere descend din `Reader` si `Writer`.

Acestea sunt imbunatatite datorita faptului ca in general streamurile de caractere sunt organizate pe linii: `BufferedReader`, `PrintWriter`

12.1.3 Stream-uri buffered

Stream-urile ne-buffered comunica direct cu sistemul de operare pentru fiecare operatie de I/O. Asta poate duce la lipsa eficientei programului deoarece aceste operatii pot fi costisitoare. Pentru a imbunatati acest proces exista stream-uri buffered (BufferedReader, BufferedWriter).

Conceptul de flush reprezinta fortarea scrierii fizice a informatiei din stream la un moment ales de program. Pentru asta exista metoda flush.

12.1.4 Formatare si scanare

- Scanare: Clasa Scanner
- Formatare: Metodele print si println si metoda format.

12.1.5 I/O din linia de comanda

- System.in
- System.out
- System.err
- Clasa Console (1.6)

12.1.6 Stream-uri de date

Permit operatii cu tipurile de date primitive.

DataInput, DataOutput

12.1.7 Stream-uri de obiecte

- ObjectOutputStream
- ObjectInputStream
- Serializarea.

12.2 I/O la nivel de fisier

O cale(path) este modul de a identifica locatia unui fisier.

Cai:

- absolute (pornesc intotdeauna de la radacina sistemului de operare)
- relative (pornesc de la o locatie curenta)

12.2.1 Clasa Path

Reprezentarea unei cai in sistemul de fisiere.

Crearea unui Path

```
Path p1 = Paths.get("/tmp/foo");
Path p2 = Paths.get(args[0]);
Path p3 = Paths.get(URI.create("file:///Users/joe/FileTest.java"));
Path p4 = FileSystems.getDefault().getPath("/users/sally");
```

Obtinerea de informatii dintr-un Path

- toString
- getFileName
- getName(n)
- getNameCount
- subpath(a,b)
- getParent
- getRoot

Eliminarea redundantelor dintr-un path

Ex:

```
/home/./joe/foo  
/home/sally/../joe/foo
```

normalize()

Convertirea unui Path

```
toUri  
toAbsolutePath  
toRealPath
```

Combinarea cailor

resolve()

Crearea unei cai intre doua cai diferite

relativize()

Compararea cailor

```
equals  
startsWith  
endsWith
```

12.2.2 Operatii cu fisierele

- prinderea exceptiilor, inchiderea resurselor: try-with-resources
- `Files.newDirectoryStream(Path path, String glob)` – cautare bazata pe pattern-uri
- verificarea existentei (`exists`)
- verificarea accesibilitatii (`isReadable`, `isExecutable`, `isRegularFile`, `isWritable`)
- verificarea daca doua cai diferite pointeaza catre acelasi fisier (`isSameFile`)
- stergerea: `delete`
- copierea: `copy`
- mutarea: `move`
- analiza metadatelor (`size`, `isDirectory`, `isRegularFile`, `isSymbolicLink`, `isHidden`, `getLastModifiedTime`, `setLastModifiedTime`, `getOwner`, `setOwner`, ...)

13 Anexe

13.1 Eclipse tips

Shortcut	Locatie	Efect
Ctrl-S	In Java Editor	Salveaza fisierul curent
Ctrl-1	pe o linie cu probleme	Sugereaza rezolvare utile pentru problemele din cod
Ctrl-Space	in timp ce scrieti un nume de clasa/metoda/variabila/etc...	face autocomplete
Alt-Shift-R	pe un nume de metoda/variabila/clasa/... selectat	urmat de introducerea noului nume si de Enter - redenumirea unei metode inclusiv in toate locurile unde e folosita
Ctrl-Shift-G	pe un nume de metoda/variabila/clasa/... selectat	gaseste locurile unde este folosita respectiva entitate in cod
Double click	imediat dupa o paranteza/acolada	selecteaza automat blocul de cod din interiorul parantezelor respective
Ctrl-Shift-B	In Java Editor	seteaza un breakpoint la linia curenta
Ctrl-Shift-T	In Java Editor	cauta o clasa dupa nume
Ctrl-Shift-R	In Java Editor	cauta o resursa dupa nume
F3	cu cursorul positionat pe un nume de metoda/variabila	merge la declaratia metodei/variabilei de la pozitia cursorului
Ctrl-D	In Java Editor	Sterge linia curenta
Ctrl-stanga, Ctrl-dreapta	In Java Editor	Navigare mai rapida si utila printr-o linie de cod
Ctrl-Shift-stanga, Ctrl-Shift-dreapta	In Java Editor	Navigare mai rapida si utila cu selectare printr-o linie de cod
Shift-End, Shift-Home	In Java Editor	Selecteaza codul de la pozitia cursorului pana la sfarsitul/inceputul liniei curente
Alt-Shift-M	Pe o secventa de cod selectata	Extrage o metoda din secventa respectiva de cod daca acest lucru este posibil
Alt-Shift-L	Pe o expresie	Extrage o variabila locala careia ii da valoarea "returnata" de expresia selectata
Meniul Source	-	Se pot genera automat setteri, getteri, toString, equals, constructori...
Ctrl-O	In Java Editor	Cautare rapida a unei metode intr-o clasa
Ctrl-F11	In Java Editor	Ruleaza ultimul program rulat
Alt-Shift-X, J	In Java Editor	Ruleaza clasa curenta
Alt-stanga, Alt-dreapta	In Java Editor	Navigare intre ultimele editari

13.2 Total commander tips

Shortcut	Efect
Ctrl-F1	Show brief info
Ctrl-F2	Show full info
Ctrl-F4	Sort by extension
Ctrl-F5	Sort by date
Tab	Switch between tabs
F3	View file
F4	Edit file
F5	Copy
F6	Move
F7	New folder
F8	Delete (with backup in Recycle bin)
Shift-delete	Delete forever
Ctrl-Page Down pe un fisier .ZIP, .JAR	Intra in structura de foldere a fisierului
Ctrl-D	Directory hotlist
Ctrl-A	Select all
Insert sau right click	Select/unselect file/folder
Long Right click	Windows right click menu
Alt-F7	Search files
Ctrl-Q pe folder/fisier	Quick view
Ctrl-R	Refresh
Ctrl-E	Commands history