

4. FACILITATI DE LIMBAJ SI CLASE PREDEFINITE UTILE

4.1. Cuvantul cheie static: ...si nu mai trebuie sa instantiez!.....	<u>2</u>
4.1.1. Campuri statice.....	<u>2</u>
4.1.2. Metode statice.....	<u>3</u>
4.1.3. Blocuri de initializare statice.....	<u>3</u>
4.2. Ordinea initializarii.....	<u>4</u>
4.3. Lucrul cu constante: cuvantul cheie final.....	<u>4</u>
4.4. Tipuri de date enumerate.....	<u>6</u>
4.4.1. Necesitatea tipurilor de date enumerate.....	<u>6</u>
4.4.2. O solutie intermediara.....	<u>6</u>
4.4.3. Definirea tipurilor de date enumerate.....	<u>6</u>
4.4.4. Utilizarea unui tip de date enumerate.....	<u>7</u>
4.4.5. Metode predefinite ale tipurilor de date enumerate.....	<u>8</u>
4.5. Clase predefinite utile.....	<u>8</u>
4.5.1. Clasa Math.....	<u>8</u>
4.5.2. Clasa Console.....	<u>9</u>
4.5.3. Clasele care impacheteaza primitive.....	<u>9</u>
4.5.3.1. Utilitatea si modul de creare a obiectelor de impachetare.....	<u>9</u>
4.5.3.2. Facilitati ale claselor de impachetare.....	<u>10</u>
4.5.3.4. Clasele String, StringBuffer si StringBuilder.....	<u>11</u>
4.5.4.1. Aspecte generale.....	<u>11</u>
4.5.4.2. Clasa String.....	<u>12</u>
4.5.4.3. Clasa StringBuffer.....	<u>13</u>
4.5.5. Clase pentru lucrul cu date calendaristice.....	<u>14</u>
4.5.5.1. Clase disponibile si scopul lor.....	<u>14</u>
4.5.5.2. Clasa Date.....	<u>14</u>
4.5.5.3. Clasele Calendar si GregorianCalendar.....	<u>14</u>
4.5.5.4. Formatare si parsare de date: clasa SimpleDateFormat.....	<u>16</u>
4.6. BIBLIOGRAFIE.....	<u>18</u>
4.7. Anexa 1 - specificatori de format pentru SimpleDateFormat.....	<u>19</u>

4.1. Cuvantul cheie static: ...si nu mai trebuie sa instantiez!

4.1.1. Campuri statice

Pana acum, fiecare obiect (instanta a unei clase) avea propria copie a setului de atribute din definitia clasei. In momentul in care declarăm un camp ca static, acesta nu se va mai multiplica - va exista intr-un singur exemplar in memorie, indiferent de numarul de instante ale clasei. Campul este vizibil tuturor instantelor clasei si poate fi modificat de oricare dintre acestea.

Un astfel de camp se declara folosind calificatorul **static** ca parte a definitiei sale:

```
class Statice {
    public static int x = 8;    // camp static
    public float y;           // camp de instantă
}
```

Nota: numim campuri/metode de instantă acei membri ai clasei a caror definitie nu include calificatorul static.

Un camp static poate fi accesat in doua moduri:

- folosind o referinta la un obiect al clasei, cu sintaxa obisnuita pentru campuri (referinta.numeCamp). Nu uitati insa - campul este intr-un singur exemplar!

```
class Test{
    public static int a = 5;
}
class Test2{
    public static void main(String[] args){
        Test t1 = new Test();
        Test t1 = new Test();
        t1.a = 8;
        System.out.println(t2.a); // 8, deoarece este vorba de acelasi a, singurul!
    }
}
```

- in masura in care nivelul sau de acces o permite, un camp static poate fi accesat din afara clasei folosind urmatoarea sintaxa:

```
NumeClasa.numeCamp
```

In cazul exemplului de mai sus putem scrie:

```
System.out.println(Test.a); // 8
```

Observam asadar ca un camp static poate fi accesat fara a fi necesara instantierea clasei. Acest lucru este posibil deoarece campurile statice sunt create si initialize in momentul incarcarii clasei in masina virtuala. Luand in considerare acest aspect si inca doua - faptul ca un camp static exista intr-un singur exemplar si ca poate fi accesat direct folosind numele clasei - nu este de mirare ca campurile statice mai sunt numite "campuri ale clasei" (prin contrast cu campurile de instantă care sunt "ale obiectelor" - caci fiecare obiect dispune de propria copie a acelui camp).

Iata cateva posibilitati de utilizare a campurilor statice:

- un camp static este echivalentul unei variabile globale din limbajele procedurale. In Java neexistand posibilitatea crearii unei variabile independente, singura posibilitate de emulare a conceptului procedural de variabila globala este crearea unui camp static public al unei clase

- aplicatiile folosesc deseori constante ce trebuie sa fie disponibile in multiple puncte ale programului. Acestea pot fi definite sub forma unor campuri statice de clasa care au in plus calificatorul final (vezi sectiunea dedicata acestui calificator)
- tot un camp static este solutia si atunci cand este nevoie de o variabila care sa nu depinda de o instanta a clasei (spre exemplu, o variabila care numara instante!)

4.1.2. Metode statice

Cazul metodelor statice este usor diferit fata de campuri: codul metodelor era oricum partajat intre obiecte (exista intr-un singur exemplar); calificatorul static isi mentine insa efectul in privinta apelarii. O metoda declarata **static** va fi disponibila si va putea fi apelata de indata ce clasa ei a fost incarcata, fara a se instantia nici un obiect. Aceasta este o modalitate de a face disponibil cod reutilizabil - echivalentul functiilor globale din procedural - executabil direct, fara a fi nevoie sa instantiem obiecte.

Atentie! **Metodele statice nu pot accesa membrii non-statici (de instanta) ai clasei!** (fie ei atribute sau alte metode) Acest lucru este normal avand in vedere ca metodele statice exista inainte de instantierea oricarui obiect si, in plus, nefiind apelate printre-o referinta la un obiect, nu pot prelua valoarea corecta pentru *this* astfel incat sa stie pe atributele carui obiect al clasei sa actioneze.

Exemple:

- metoda *parseInt()* a clasei predefinite *Integer* este una statica. Metoda se foloseste pentru a transforma un *String* intr-o valoare numérica - operatie des intalnita in Java si deci de dorit a fi disponibila fara a instantia obiecte in acest scop. Metoda a fost atasata clasei *Integer* si poate fi apelata folosind numele clasei:

```
int i=Integer.parseInt("123");
```

- metoda *main()*, folosita ca punct de intrare pentru programele noastre pana acum, este tot statica – ceea ce are sens daca ne gandim ca ea trebuie apelata *inainte* de a instantia obiecte:

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

4.1.3. Blocuri de initializare statice

Asa cum putem executa cod cu ocazia instantierii unui obiect folosind constructorul, exista o posibilitate de a executa cod in momentul incarcarii unei clase in masina virtuala: blocurile de initializare statice. Un astfel de bloc reprezinta un set de instructiuni incadrate intre { si } (la fel ca in cazul oricarei alte instructiuni bloc) si precedate de cuvantul static, plasat in rand cu membrii clasei:

```
class BlocInitializare{
    int x;
    static int y;
    static void f(){ /* ...o implementare oarecare... */ }
    static{
        System.out.println("S-a incarcat clasa BlocInitializare!");
        x = 15; // eroare de compilare - din bloc static nu pot accesa membri de instanta
        f();
    }
}
```

In cadrul unui bloc de initializare static pot fi accesati membri statici ai clasei, insa nu si membri de instanta. **Atentie insa! Campurile statice accesate trebuie sa fi fost declarate inaintea blocului static!**

O clasa poate avea mai multe blocuri de initializare statice, plasate in orice ordine fata de membrii clasei (inclusiv intercalate cu ei). Ele vor fi execute toate la incarcarea clasei, in ordinea in care apar in definitia acesteia.

4.2. Ordinea initializarii

In momentul in care avem atat campuri statice cat si non-statice, initializate fie la declarare, fie in cadrul constructorului, trebuie sa ne fie foarte clara ordinea in care se efectueaza operatiunile de initializare la incarcarea, respectiv instantierea unei clase. Regulile sunt urmatoarele, respectand pe cat posibil cronologia evenimentelor:

- o clasa este incarcata abia in momentul in care apare prima referire la acel tip de date:
 - cand este declarata o variabila de acel tip, sau
 - cand este accesat un membru static al acesteia
- la incarcarea clasei
 - sunt create si initializate campurile statice
 - cele initializate explicit (la declarare) primesc valoarea specificata
 - cele neinitializate raman cu valoarea default a tipului de date respectiv (null pentru referinte si valoarea default a tipului de date in cauza pentru primitive)
 - sunt execute blocurile de initializare statice. Ordinea in care apar campurile si blocurile statice este relevanta, deoarece:
 - un camp static trebuie sa fi fost declarat deja in momentul in care este referit direct dintr-un bloc static; in caz contrar rezulta o eroare de compilare ("illegal forward reference")
 - un bloc static poate totusi utiliza indirect valoarea unui camp static inca nedeclarat (ai bine spus, declarat mai jos in definitia clasei) in conditiile in care apeleaza o metoda statica si aceasta utilizeaza campul in cauza; un astfel de camp va avea atunci valoarea default a tipului sau de date. Priviti urmatorul exemplu:

```
class Statice{
    static int x=f();      // x e initializat cu rezultatul returnat de f(); f() foloseste campul
                           // y, care nu a fost inca initializat si in consecinta are valoarea 0!
    static{
        System.out.println("x="+x); // x=0
        System.out.println(y);     // eroare: illegal forward reference. y nu e declarat inca
        System.out.println(f());   // 0, caci y-ul folosit in f() inca nu este initializat
    }

    static int f(){
        System.out.println("f()");
        return y;
    }
    static int y = 6;

    static{
        System.out.println(y);     // 6, in sfarsit
    }
}
```

- la instantierea clasei:
 - sunt initializate campurile de instanta
 - cele initializate la declarare primesc valoarea specificata
 - cele neinitializate primesc valoarea default a tipului de date respectiv
 - ordinea initializarii campurilor este cea in care sunt declarate in cadrul clasei; initializarea campurilor se face in acest stadiu, in ordinea in care apar in definitia clasei, indiferent in ce punct in cadrul sursei programului apar declaratiile acestora (de exemplu, declaratii de campuri care apar dedesubtul declaratiilor unor metode)
 - se executa constructorul, care poate la randul sau modifica valori ale campurilor

4.3. Lucrul cu constante: cuvantul cheie final

Cuvantul cheie **final** este folosit pentru a crea constante de diferite nati. El poate fi aplicat pentru:

- variabile locale. Valoarea unei astfel de variabile trebuie specificata la declarare sau ulterior acesteia, insa inainte ca valoarea variabilei sa fie citita. Odata ce variabila a primit valoare, ea nu mai poate fi modificata

- campuri de instantă. Valoarea unui astfel de camp poate fi specificată fie în definiția clasei, la declararea campului, fie în constructor. În ambele cazuri, odată valoarea atribuită variabilei, ea nu mai poate fi schimbată.
- campuri statice. Valoarea campului poate fi stabilită fie la declarare, fie într-un bloc static ce-i urmează
- metode ale clasei. O metodă ce are calificatorul **final** nu poate fi rescrisă în subclase (vezi lectia despre mostenire)
- clase. O clasa ce are calificatorul **final** nu poate fi derivată (vezi lectia despre mostenire)

Atentie! Variabilele corespunzătoare obiectelor sunt **referinte**, de aceea la declararea lor ca **final** numai *referinta catre obiect* va fi constantă, nu și continutul acestuia!

```
class Numere {
    public final int ZERO=0;
}
```

Valoarea unui camp de instantă declarat ca *final* poate rezulta într-unul dintre urmatoarele moduri:

- constanta este initializată la declarare; compilatorul nu ne va lăsa să-i atribuim o altă valoare ulterior, nici chiar în constructor
- constanta este doar declarată („*blank final*”) și initializarea să se facă în constructor; odată initializată însă, valoarea să nu mai poate fi schimbată, nici în constructor, nici în altă metodă a clasei
- constanta este doar declarată, însă nu este initializată explicit nici la declarare, nici în constructor. În acest caz, campul va rămâne cu valoarea default a tipului sau de date și nu va mai putea fi modificat

Campurile statice declarate ca *final* pot fi initializate fie la declarare, fie într-un bloc de initializare static ulterior, cu mențiunea că, odată ce campul a primit valoare, el nu mai poate fi modificat. Diferența față de campurile de instantă este că nu mai putem declara blank finals fără a le initializa explicit.

```
class Constante {
    public final int x=3,y;           // de aici încolo x-ul nu mai poate fi modificat
    public static final int z;        // eroare - z nu este initializat explicit mai jos
    public static final int t=2;
    public static final int u;
    public final StringBuffer s;     // o viitoare referință constantă

    static{
        t=4;                      // eroare - t a fost deja initializat
        u=2;                      // permis, u nu fusese initializat
        u=5;                      // eroare - u nu mai poate fi modificat
    }

    public C2() {
        s=new StringBuffer(); // este permisă initializarea unui camp final în constructor
        s="hi";               // a două atribuire nu mai e permisă - eroare
        x=2;                 // x a fost deja initializat la declarare - eroare
        y=5;                 // permis; y e variabilă de instantă și suntem încă în cadrul constructorului
        z=9;                 // a trecut timpul cand puteam să-l initializam pe z; eroare
    }
    public static void main(String[] args) {
        final int a;
        System.out.println(a);      // eroare - a este neinitializat
        a=4;                      // ok
        a=2;                      // a două atribuire - eroare;

        Constante c=new Constante();
        c.x=6; // x a fost deja initializat - eroare
        c.t=5; // t era initializat încă de dinainte de constructor; eroare
        c.s=new StringBuffer();    // s a fost deja initializat în constructor - eroare
        c.s.append("yupee");       // doar referință c este constantă,
                                   // nu și continutul obiectului!
    }
}
```

4.4. Tipuri de date enumerate

4.4.1. Necesitatea tipurilor de date enumerate

Deseori avem nevoie de un tip date ale carui valori formeaza un set discret, finit, cunoscut inca de la compilare (de ex: zilele saptamanii, notele muzicale, anotimpurile, culorile semaforului etc). Desi exista posibilitatea definirii unor constante numerice (sau obiecte String) care sa denumeasca sugestiv fiecare valoare concreta, procedeul este departe de a fi optim:

```
public class Orar{
    public final int LUNI = 1;
    public final int MARTI = 2;
    // ... etc ...
    public void suntOcupat(int zi){
        switch(zi){
            case LUNI: System.out.println("Multă treaba");
                break;
            case MARTI: System.out.println("Mi-am mai intrat un ritm...");  

                break;
            //...etc...
        }
    }
}
```

Problema portiunii de cod de mai sus este ca metodei *suntOcupat()* i se poate pasa ca argument orice valoare permisa de tipul de date *int*; asadar setul nostru de valori nu este *type-safe* - programatorul poate trata oricare dintre valori ca pe un simplu *int*, poate aduna doua zile (operatie lipsita de sens) sau poate pasa metodei o valoare ce nu se regaseste in setul de constante definit. Avem nevoie de un tip de date distinct, care sa poata lua numai un set predefinit de valori.

4.4.2. O solutie intermediara

In cartea sa "Effective Java", Bruce Eckel prezinta o solutie pentru crearea de tipuri de date enumerate type-safe, prin definirea unei clase cu constructor privat si care contine pe post de campuri instance de-ale sale. Campurile sunt declarate cu calificatorii static si final. In acest fel, instance ale clasei nu pot fi create decat din interiorul ei, iar interiorul contine numai cele cateva constante ca reprezinta valorile tipului de date:

```
class CuloareCarti{
    private String sinonim;
    private CuloareCarti(String s){ sinonim = s; }
    public static final CuloareCarti CUPA = new CuloareCarti("Inima rosie");
    public static final CuloareCarti TREFLA = new CuloareCarti("Spatie");
    public static final CuloareCarti PICA = new CuloareCarti("Inima neagra");
    public static final CuloareCarti CARO = new CuloareCarti("Romb");
}
```

Fiecare dintre cele 4 culori exista intr-im singur exemplar, ceea ce inseamna ca putem compara intre ele valorile de tip CuloareCarti folosind operatorul `==`, ca la valorile de tip primitiva!

4.4.3. Definirea tipurilor de date enumerate

In Java 1.5 s-au introdus tipurile de date enumerate ("enum"), care se definesc asemanator unei clase si practic automatizeaza solutia prezentata anterior.

```
public enum Anotimp { PRIMAVARA, VARA, TOAMNA, IARNA }
```

Enum-ul poate fi declarat in afara unei clase sau in interiorul ei ca membru al clasei, insa **nu si in metode** (un enum nu poate fi local).

Un enum este o clasa care nu poate avea alte instance decat cele declarate in interiorul sau. El poate avea membri, ca orice clasa, dar cu urmatoarele restrictii:

- constantele componente (valorile posibile ale tipului de date enumerat) sunt automat *public, static si final*, cu implicatiile de rigoare (pot fi accesate din afara clasei folosind *NumeEnum.numeConstanta*)
- daca exista campuri in plus fata de constante:
 - campurile trebuie declarate ca final
 - constantele ce reprezinta valorile noului tip de date trebuie declarate inaintea campurilor, iar lista lor trebuie urmata de ; (punct si virgula)
- constructorul
 - daca programatorul nu scrie niciun constructor, va fi introdus cel default, cu nivel de acces private
 - constructorii definiti de programator pot avea argumente si trebuie sa aiba nivel de acces default sau private
- la declararea constantelor componente, fiecare dintre acestea va fi urmat de valorile pentru parametrii pasati constructorului, plasate intre paranteze rotunde ca in exemplul de mai jos

```
public enum CuloareSemafor{
    ROSU("Stop!"),
    GALBEN("Asteapta"),
    VERDE("Mergi");

    private final double semnificatie;

    CuloareSemafor(String s){
        semnificatie = s;
    }

    public String getSemnificatie(){ return semnificatie; }
}
```

4.4.4. Utilizarea unui tip de date enumerat

Un tip de date enumerat se manifesta ca o clasa in toata regula, si de aceea i se aplica aceleasi reguli de vizibilitate ca pentru clase.

Utilizarile unui tip de date enumerat sunt diverse:

- se pot declara variabile de acel tip de date. Ele vor putea fi initializate numai cu una dintre valorile posibile ale enum-ului
- se pot folosi pe post de parametri in metode. In acest fel, parametrii sunt type-safe - compilatorul va putea verifica ca metodei i se paseaza la apelare valori cu tip de date corect
- se pot utiliza in constructia *switch* (vezi exemplul de mai jos)

Referirea la constantele componente ale unui enum se realizeaza astfel:

- cu numele lor scurt (ex: IARNA) atunci cand valorile sunt folosite in case-urile unui switch
- cu numele lor complet (ex: Anotimp.IARNA) in rest

Exemplu: o metoda care primeste ca argument un anotimp:

```
enum Anotimp{PRIMAVARA, VARA, TOAMNA, IARNA, NEDEFINIT};
public void activitate(Anotimp a){
    if(a == Anotimp.NEDEFINIT) return;          // folosim numele lung al constantei; pentru enum-
                                                // uri este permisa compararea folosind ==
    switch(a){                                // putem folosi valorile enumerarii ca argument in switch!
        case VARA:                           System.out.println("Mergem la mare!"); // in case folosim numele scurt
                                                break;
        case IARNA:                          System.out.println("Mergem la ski!");
    }
}
```

```

        break;
    default:
        System.out.println("Neinteresant...prea multe ploi!");
        break;
    }
}

```

4.4.5. Metode predefinite ale tipurilor de date enumerate

Orice obiect enum dispune de cateva metode predefinite, provenite din doua surse:

- asa cum un obiect tablou are din oficiu un atribut length, enum-urile au doua metode statice default:
 - **values()** - intoarce un array cu constantele tipului de date enumerat, in ordinea in care apar in definitie
 - **valueOf(String)** - intoarce constanta al carei nume a fost pasat ca argument sub forma de String
- enum-urile mostenesc clasa Enum, care le transmite metodele sale de instanta; dintre ele amintim:
 - **name()** - produce numele constantei asa cum apare el in definitia clasei, sub forma de String
 - **ordinal()** - intoarce pozitia constantei in lista (conform definitiei enum-ului), numerotata de la 0
 - **equals(constanta)** - returneaza o valoare de tip boolean, indicand daca constanta pentru care este apelata metoda este aceeasi cu cea primita ca argument
 - **compareTo(constanta)** - metoda compara constanta pentru care a fost apelata cu cea primita ca argument. Apelul *c1.compareTo(c2)* va returneaza: un numar negativ daca *c1 < c2*, 0 in caz de egalitate si un numar pozitiv daca *c1 > c2*. Constanta "mai mare" este cea aflata mai tarziu in lista din definitia enum-ului (ex: IARNA este considerata mai mare decat toate celelalte constante ale clasei *Anotimp*)
 - **toString()** - produce o reprezentare textuala a constantei. Spre deosebire de *name()*, aceasta metoda poate fi rescrisa de catre programator pentru a oferi o descriere a constantei in locul simplului sau nume (vezi capitolul despre mostenire, sectiunea overriding)

Remarcam urmatoarele:

- atunci cand avem nevoie de simpla comparare de egalitate, putem folosi fie *equals()*, fie *==*, gratie faptului ca fiecare constanta se gaseste intr-un singur exemplar
- atunci cand ne intereseaza relatia exacta dintre doua constante (care dintre ele este "mai mare"), putem fie sa utilizam *compareTo()*, fie sa extragem pozitia fiecarei constante folosind metoda *ordinal()* si apoi sa comparam pozitiile

```

class DemoEnum{
    enum CertificareJava{ ASSOCIATE, PROGRAMMER, DEVELOPER, ARCHITECT };
    public static void main(String[] args){
        CertificareJava c = CertificareJava.DEVELOPER;
        CertificareJava c1 = CertificareJava.valueOf("DEVELOPER");
        System.out.println(c.name());                                // DEVELOPER
        System.out.println(c.ordinal());                            // 2
        System.out.println(c.compareTo(CertificareJava.ASSOCIATE)); // numar pozitiv
        System.out.println(c.equals(CertificareJava.DEVELOPER));   // true
        System.out.println(c == CertificareJava.DEVELOPER);        // true
        for(CertificareJava cert:CertificareJava.values()){
            System.out.println(cert.toString());                   // numele certificarii
        }
    }
}

```

4.5. Clase predefinite utile

4.5.1. Clasa Math

Clasa Math este o colectie de metode statice corespunzatoare operatiilor matematice uzuale. Iata cativa dintre cei mai importanti membri ai clasei:

- constante publice (campuri public, static si final): **Math.PI** si **Math.E**, celebrele constante matematice
- metode

- modul: **Math.abs(valoare)**. Metoda este supraincarcata, existand variante pentru toate tipurile de date numerice
- ridicare la putere: **Math.pow(double baza, double exponent)**
- determinarea celui mai mare sau mai mic numar dintr-o pereche: **Math.max(nr1,nr2), Math.min(nr1,nr2)**. Ambele metode sunt supraincarcate pentru a accepta ca argument toate tipurile de date numerice
- generare de valori aleatoare: **Math.random()**, care produce o valoare de tip double mai mare sau egala cu 0 si strict mai mica decat 1
- determinarea semnului unui numar: **Math.signum(numar)**. Produce -1.0 pentru numere negative, 0 pentru numere nule si 1.0 pentru numere pozitive
- extragerea radacinii patrate: **Math.sqrt(double)**
- trunchiere si rotunjire numerica:
 - **Math.floor(double)** - produce intregul imediat inferior valorii primite ca argument
 - **Math.ceil(double)** - produce intregul imediat superior valorii primite ca argument
 - **Math.round(numar)** - produce cel mai apropiat numar intreg (fie el superior sau inferior)
- functii trigonometrice: **Math.sin(double), Math.cos(double)** etc.

***Nota:** in descrierile de mai sus, acolo unde metodele nu sunt supraincarcate s-a specificat tipul exact de date primit ca argument, iar unde exista supraincarcare s-a specificat "numar" pe post de argument.*

4.5.2. Clasa Console

Consola Java este un spatiu text in care masina virtuala poate trimite output si din care isi poate citi input. In Windows, cand pornim o aplicatie Java din Command Prompt, consola este insasi fereastra in cauza. In Linux, atunci cand rulam masina virtuala dintr-un terminal, acesta constituie consola masinii virtuale. Output-ul pe care l-au generat pana acum aplicatiile noastre (folosind `System.out.println()` si metodele inrudite) ajungea automat in consola Java.

Nu intotdeauna masina virtuala are consola. Limbajul Java este des folosit pentru crearea de aplicatii grafice - fie ele applet-uri sau aplicatii de sine statatoare - iar o astfel de aplicatie nu dispune de consola decat in cazuri speciale. Iata cateva situatii in care consola este prezenta:

- atunci cand rulam o aplicatie Java din Command Prompt-ul de Windows sau dintr-un terminal Linux
- atunci cand rulam o aplicatie dintr-un mediu de dezvoltare, care are grija sa ne prezinte output-ul generat de aplicatie intr-o fereastra de consola dedicata
- in cazul aplicatiilor grafice, in unele sisteme de operare si versiuni de Java exista posibilitatea activarea explicita a consolei Java. Oricum insa, programatorul nu are garantata existenta ei

Concluzia majora este ca nu putem baza intreaga interactivitate cu utilizatorul a unui program Java pe `System.out.println()`, deoarece este posibil ca output-ul generat astfel sa nu ajunga deloc la utilizator! Consola reprezinta mai degrabă un spatiu de diagnostic, chiar daca exista si aplicatii care o folosesc ca mod de interactivitate principal.

Atunci cand masina virtuala are consola, aceasta este reprezentata prin intermediul unei unice instante a clasei `Console`. Aceasta se obtine apeland metoda `System.console()`; in cazul in care nu exista consola, aceasta metoda returneaza null;

Iata cateva metode de interes ale clasei `Console`:

- **readLine()** - citeste un sir de caractere de la tastatura si il returneaza. Utilizatorul trebuie sa incheie sirul de caractere cu ENTER
- **readPassword()** - analog, insa caracterele scrise de utilizator nu vor fi afisate pe ecran sau vor fi inlocuite cu asterisc (util pentru citire de parolă)
- **format()** si **printf()** - metode pentru afisarea de text ce respecta un format dorit

4.5.3. Clasele care impacheteaza primitive

4.5.3.1. Utilitatea si modul de creare a obiectelor de impachetare

Pana acum am subliniat faptul ca primitivele si obiectele sunt amandoua tipuri de date, dar de natura diferita si, mai ales, tratate diferit (valoarea unei primitive este pusa in corespondenta directa cu un nume de variabila, pe cand obiectele se

manevreaza indirect, prin intermediul unei referinte). JRE (Java Runtime Environment) ne pune la dispozitie un număr de clase care permit lucrul cu datele primitive incapsulate in obiecte: este vorba de clasele de impachetare (*wrapper classes*), care nu fac decât sa creeze obiecte care **impacheteaza** date de tip primitiva. Acest lucru este necesar cand dorim folosirea primitivelor in cadrul unei infrastructuri deja existente insa gandita pentru obiecte (spre exemplu, Collections Framework - infrastructura de colectii de obiecte Java). De asemenea, clasele de impachetare pun la dispozitia utilizatorului constante utile (ex: limita superioara si inferioara a primitivei coresponzatoare, sub forma de constanta a clasei) si metode pentru conversia din String in primitiva, din primitiva in alte tipuri de primitive sau in alte baze de numeratie etc.

Clasele de impachetare dispun de constructori multipli, care permit crearea unui obiect de impachetare pornind fie de la tipul de date primitiv coresponzator, fie de la un obiect de tip String:

Corespondenta clasa-primitiva	Exemplu de instantiere
Short<=>short	Short s=new Short("1234"); SAU Short s=new Short((short)1234);
Integer<=>int	Integer i=new Integer(1234); SAU Integer i=new Integer("1234");
Long<=>long	Long l=new Long(1234567); SAU Long l=new Long("1234567");
Float<=>float	Float f=new Float(12.3f); SAU Float f=new Float("12.3");
Double<=>double	Double d=new Double(12.3); SAU Double d=new Double("12.3");
Character<=>char	Character c=new Character('c');
Boolean<=>boolean	Boolean b=new Boolean(true); SAU Boolean b=new Boolean("True");

Trebuie subliniat ca prin instantierea claselor de impachetare se obtin **obiecte**, care pana la versiunea Java 1.4 inclusiv **nu pot fi folosite ca atare in locul primitiveelor**; operatorii obisnuiti nu se aplica obiectelor de impachetare. De exemplu, nu vom putea scrie:

```
Boolean b=new Boolean(true); if(b) { /*fa ceva */ } // eronat in Java <= 1.4
Integer x=new Integer(1),y=new Integer(2); x+=y; // eronat in Java <= 1.4
Integer x = 5; // eronat in Java <= 1.4
```

Incepand cu versiunea 1.5 a fost introdusa facilitatea de automatic boxing/unboxing, care permite folosirea interschimbabila a primitivelor si a obiectelor de impachetare.

***Nota:** pe langa clasele de impachetare de mai sus exista inca doua clase ale caror obiecte incapsuleaza numere, dar cu scop diferit: BigDecimal si BigInteger. BigDecimal impacheteaza numere cu multe zecimale si defineste operatii in care precizia calculelor este stabilita de catre programator. Obiectele de tip BigInteger sunt folosite pentru memorarea si lucrul cu numere intregi foarte mari. Ambele clase dispun de metode pentru operatiile standard cu numere, pastrand insa precizia deosebita.*

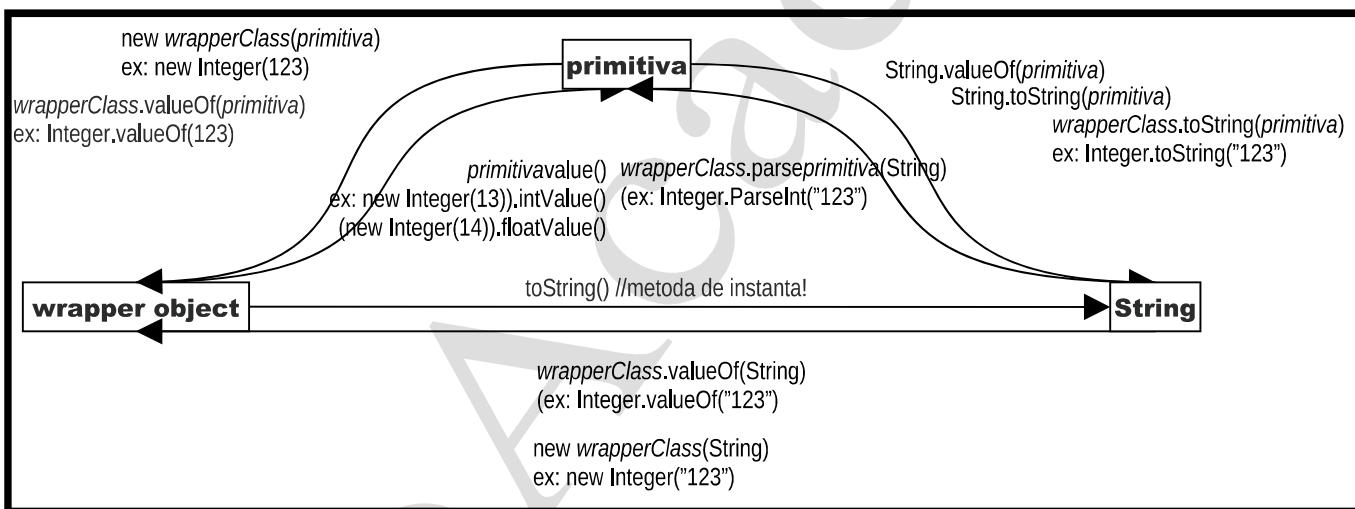
4.5.3.2. Facilitati ale claselor de impachetare

Desi fiecare dintre clase are si propriile-i particularitati, exista cateva categorii de elemente comune acestor clase pe care le vom expune in continuare.

- constante publice. Fiecare dintre clase dispune de un set de constante de interes public - iata cateva exemple:
 - in clasele coresponzatoare primitive numerice sunt definite constante cu valorile minima si maxima ale tipului de date coresponzator. Exemplu: `Integer.MIN_VALUE` si `Integer.MAX_VALUE`
 - in clasa Boolean sunt definite constantele `Boolean.TRUE` si `Boolean.FALSE` ce au ca valori obiectele Boolean coresponzatoare celor doua valori posibile ale primitivei
- metode de instantanta
 - obtinerea reprezentarii textuale a datelor impachetate. Metoda de instantanta `toString()` intoarce un String reprezentand valoarea incapsulata
 - extragerea valoiei incapsulate. Metodele de instantanta `integerValue()`, `booleanValue()`, `byteValue()` etc intorc valoarea incapsulata in wrapper object, sub forma primitivei dorite (si care nu corespunde neaparat cu tipul incapsulat! - un fel de cast). Exemplu: `Float f=new Float(12.3s); int i=f.intValue();`

- metode de comparare - utilizeaza pentru a determina relatia dintre doua obiecte de impachetare (indispensabile in Java <= 1.4, unde operatorii nu functionau pentru obiecte de impachetare)
 - **boolean equals(obiect_impachetare)** - indica daca obiectul pentru care a fost apelata incapsuleaza aceiasi valoare cu cel primit ca argument
 - **int compareTo(obiect_impachetare)** - indica relatia dintre cele doua obiecte, conform conventiei prezentate anterior in cadrul acestui material
- metode statice - reprezinta cod atasat unei clase de impachetare si destinat refolosirii publice, fara a presupune instantierea clasei. Cateva exemple:
 - obtinerea unei valori de tip primitiva pe baza unui String ce contine reprezentarea textuala a valorii. Exista in acest scop metodele de parseare - **Integer.parseInt(String s)**, **Long.parseLong(String s)**, **Double.parseDouble(String s)** etc. Exemplu: **Integer.parseInt("1234")** returneaza valoarea de tip int 1234.
 - obtinerea unei valori de tip String pe baza uneia de tip primitiva - actiune realizata de metodele **Integer.toString(int)**, **Boolean.toString(boolean)** etc.
 - obtinerea unui obiect de impachetare pe baza unei reprezentari String sau de tip primitiva. In acest scop exista metodele **valueOf()** ce au un rol asemenea cu constructorul - produc o valoare de tip obiect de impachetare. Exemple: **Integer.valueOf(String)**, **Integer.valueOf(int)**, **Boolean.valueOf(String)** etc.

Actiunea realizata de metodele **parse...()** si cele **toString()** poate fi privita ca un fel de conversie - nu in sensul de cast, ci in cel de reprezentare diferita a acelasi informatie. Un acelasi numar - sa spunem, 13 - poate fi reprezentat atat ca primitiva, cat si ca String ("13") sau ca obiect *Integer* sau *Byte*. Programatorul Java este deseori pus in situatia de a realiza transformarea dintr-o reprezentare in alta si de aceea trebuie cunoscute modalitatatile de a tranzitiona intre cele trei "stari de agregare" posibile:



4.5.4. Clasele String, StringBuffer si StringBuilder

4.5.4.1. Aspecte generale

Un sir de caractere ar presupune in mod normal un tablou de primitive **char**:

```
char[] ab=new char[4];
char [0]='a';
char[1]='b';
```

Observam ca dimensiunea maxima a sirului de primitive **char** este definitiv stabilita in momentul initializarii variabilei referinta *ab*: este alocata memorie pentru patru date de tip char si adresa de inceput este asignata lui *ab*. Daca sirul nostru nu foloseste toata memoria alocata (sau se va micsora mai tarziu) este necesar sa memoram lungimea sa intr-o alta variabila. De asemenea, operatiile de adaugare/eliminare de caractere nu sunt simple: adaugarea unui al cincilea caracter

in exemplul de mai sus presupune crearea unui sir nou cu lungime 5, copierea pe primele 4 pozitii a elementelor vechiului sir si adaugarea pe pozitia 5 a ultimului.

Clasele *String*, *StringBuffer* si *StringBuilder* impacheteaza siruri de caractere, ascunzand complexitatea gestionarii unui tablou de primitive de tip **char**. Aceste clase ofera o multitudine de metode de cautare, extragere, adaugare, inserare, eliminare etc. de caractere sau subsiruri.

Dar de ce mai multe clase? Diferenta intre cele doua este urmatoarea:

- obiectele de tip *String* incapsuleaza siruri de caractere constante („immutable”). Continutul unui obiect de tip *String* nu mai poate fi modificat dupa creare
- clasa *StringBuffer* este gandita special pentru siruri de caractere care sa se poata modifica – motiv pentru care dispune de metode specifice de inserare/eliminare de caractere sau siruri in obiectul de tip *StringBuffer*
- clasa *StringBuilder* indeplineste acelasi rol cu *StringBuffer*, avand si o interfata (API) identica, insa nu este recomandabila in cazul lucrului cu mai multe fire de executie si al accesului concurrent la informatie. *StringBuffer* este protejata pentru astfel de cazuri (dar cu o penalitate de viteza din acest motiv), pe cand *StringBuilder* nu dispune de protectie dar va avea performante mai bune in aplicatii single-thread (un singur fir de executie)

4.5.4.2. Clasa String

Obiectul de tip **String** este reprezentarea standard pentru siruri de caractere in Java. De exemplu, toate sirurile de caractere constante care apar in codul sursa (de ex:`System.out.println("Infoacademy");`) sunt de fapt obiecte de tip **String** (instante ale acestei clase), deoarece compilatorul transforma automat o valoare incadrata intre ghilimele intr-un obiect de tip **String** (cu unele exceptii - vezi mai jos facilitarea *string pool*).

Clasa **String** dispune de o multitudine de constructori. Se pot construi obiecte de tip **String** pornind de la:

nimic; obiectul rezultat incapsuleaza un sir gol	<code>String s=new String();</code>
un alt obiect de tip String	<code>String s=new String("Info");</code> sau <code>s="Info";</code>
un tablou de primitive char	<code>char[] c=new char[15]; String s=new String(c);</code>
o portiune a unui tablou de primitive char , specificata ca index de inceput si lungime	<code>char[] c=new char[15]; String s=new String(c,3,10);</code>
un obiect de tip StringBuffer	<code>StringBuffer sb=new StringBuffer();</code> <code>String s=new String(sb);</code>

Obiectele de tip **String** sunt *immutable* - sirul de caractere incapsulat in obiect este **constant**. Odata obiectul de tip **String** creat, nu mai putem modifica sirul de caractere incapsulat, deoarece clasa nu ne pune la dispozitie niciun fel de metode in acest scop.

Clasa **String** dedica o zona de memorie (numita **string pool**) memorarii de siruri unice. La intalnirea unei noi constante (“literal”) de tip **String** in codul sursa, se verifica daca nu cumva sirul continut se regaseste in *string pool*, si in caz afirmativ nu se creeaza un obiect nou, ci este returnata referinta catre obiectul deja existent. Din acest motiv este posibila refolosirea sirurilor care apar in program: de exemplu, daca declaram **String** `s1="abc"` si **String** `s2="abc"`, `s1` si `s2` vor fi referinte catre acelasi obiect de tip **String**. (si deci `s1==s2` va produce true!)

Există o multitudine de operații ce se pot realiza cu obiecte de tip **String**. Dat fiind sirul:

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

putem efectua prin intermediul metodelor clasei **String** urmatoarele operațiuni:

aflarea lungimii sirului	<code>int i=s.length();</code> → <code>i=11</code>
returnarea caracterului de pe o anumita pozitie	<code>char c=s.charAt(2);</code> → <code>c='f'</code>
comparare alfabetica cu alt String	<code>int i=s.compareTo("abc");</code> → <code>"Infoacademy"<"abc"</code> si deci <code>i<0</code> <code>boolean b = s.equals("InfoAcademy");</code> → <code>b=true</code>
comparare case-insensitive	<code>s.compareToIgnoreCase("AbC");</code> → <code>"infoacademy">"abc", i>0</code> <code>s.equalsIgnoreCase("infoacademy");</code> → <code>true</code>
conversie la majuscule si invers	<code>String s1=s.toLowerCase();</code> → <code>s1="infoacademy"</code> <code>String s2=s.toUpperCase();</code> → <code>s2="INFOACADEMY"</code>

extragerea subsirului aflat intre doi indecsi: sunt returnate pozitiile index1...index2-1	String s1="012345678" String s2=s.substring(4,8); → s1="4567";
concatenare cu alt String	String s1=s.concat(" rules"); → s1="Infoacademy rules"; String s1=s+" rules"; operatorul + returneaza sir nou!
pozitia primei sau ultimei aparitii a unui caracter sau sir	int primaChar=s.indexOf('a'); → i=4 int primaSir=s.indexOf("cad"); → i=5 int ultimaChar=s.lastIndexOf('a'); → i=6 int ultimaSir=s.lastIndexOf("dem"); → i=7
transformarea unei primitive intr-un obiect de tip String	String sChar=s.valueOf('i'); String sInt=s.valueOf(13); String sLong=s.valueOf(13L); String sFloat=s.valueOf(13.2f); String sDouble=s.valueOf(13.2); String s=" Grigore Pisculescu "; String numaiNume = s.trim(); → numaiNume = "Grigore Pisculescu"
eliminarea spatiilor ce bordeaaza informatia (util pentru citirea datelor de la utilizator)	
verificarea compositiei/formatului string-ului	s.endsWith("my"); → true s.startsWith("Info"); → true s.matches("[A-Za-z]{11}"); → true (vezi lectia despre regex-uri)
impartire sir in subsiruri conform unui delimiter	String steag = "rosu,galben,albastru"; String[] culori = steag.split(","); System.out.println(culori[0]); → rosu

De remarcat ca, desi am scris *s1=s+ " rules"*, nu are loc o adaugare a lui "rules" la sfarsitul lui *s*, ci este generat un nou obiect de tip *String* avand ca valoare concatenarea celor doua siruri si a carui adresa este atribuita lui *s1*.

Atentie! Toate metodele clasei *String* care aparent modifica sirul de caractere encapsulat nu fac altceva decat sa returneze un **nou** obiect de tip *String* care contine rezultatul operatiei!

4.5.4.3. Clasa StringBuffer

Clasa **StringBuffer** impacheteaza siruri care pot fi modificate, si din acest motiv atat gestionarea datelor in interiorul clasei cat si setul de metode difera fata de **String**.

Pentru a putea lucra eficient cu tabloul de primitive **char** impachetat, **StringBuffer** trebuie sa aloce memoria in avans, sub forma unui buffer (zona tampon). Marimea bufferului alocat la un moment dat formeaza **capacitatea** obiectului **StringBuffer**, si reprezinta dimensiunea maxima a sirului de caractere pe care poate sa-l impachteze acel obiect fara a avea nevoie de memorie suplimentara. **Lungimea** sirului impachetat va fi intotdeauna mai mica sau egala cu capacitatea. Bufferul initial (pentru cazul in care sirul encapsulat e gol) este de 16 caractere, si este extins daca in urma unei operatii append(adaugare) sau insert (inserare) lungimea sirului depaseste capacitatea obiectului **StringBuffer**.

Obiectele de tip **StringBuffer** pot fi create prin intermediul a trei constructori, ca in exemplele de mai jos:

```
StringBuffer sb=new StringBuffer(); //sirul de caractere are lungime 0; capacitatea insa este 16!
StringBuffer sb=new StringBuffer(20); //este specificata capacitatea initiala; sirul de caractere
                                   //este de asemenea gol, insa capacitatea este 20
StringBuffer sb=new StringBuffer("Infoacademy");      // se creaza un obiect StringBuffer ce va
                                                       // incapsula acelasi sir de caractere ca si obiectul String primit ca parametru
```

Pe langa metodele prezente in clasa *String*, *StringBuffer* aduce o serie de metode noi, ce dau posibilitatea modificarii sirului de caractere encapsulat. Dat fiind obiectul *sb* ce contine sirul „Infoacademy”, putem incerca:

aflarea capacitatii obiectului	int i=sb.capacity(); // i=16
adaugarea oricarui tip de primitiva la sfarsitul sirului (append)	sb.append(!); // sb="Infoacademy!"
adaugarea unui sir la sfarsit (append)	sb.append(" rules"); // sb="Infoacademy rules"
inserarea pe o anumita pozitie a unei primitive (orice primitiva)	sb.insert(4,10); // sb="Info10academy" sb.insert(8,12.3f); // sb="Info10ac12.3ademy"
inserarea unui sir pe pozitia specificata	sb.insert(4, ""); // sb="Info academy"
stergerea unui caracter de pe pozitia specificata	sb.deleteCharAt(5); // sb="Infoaademy"
stergerea subsirului cuprins intre doua pozitii specificate	sb.delete(0,4); // sb="academy";
inlocuirea unui caracter	sb.setCharAt(0,'i'); // sb="infoacademy"

Studentul poate utiliza prezentul material si informatiile continute in el exclusiv in scopul asimilarii cunostintelor pe care le include, fara a afecta dreptul de proprietate intelectuala detinut de InfoAcademy.

inlocuirea unui intreg subsir cu altul	<code>sb.replace(0,4,"Cisco "); // sb="Cisco academy"</code>
transformarea obiectului StringBuffer intr-unul de tip String	<code>String s=sb.toString();</code>

4.5.5. Clase pentru lucrul cu date calendaristice

4.5.5.1. Clase disponibile si scopul lor

Iata principalele clase disponibile programatorului Java pentru lucrul cu date calendaristice:

- **Date** – memoreaza un moment in timp cu precizie de milisecunda. Clasa dispune de suport limitat pentru obtinerea sau modificarea momentului de timp encapsulat; pentru orice alte operatii se recomanda folosirea clasei *Calendar* si a subclasei sale *GregorianCalendar*
- **Calendar** – memoreaza un moment de timp cu precizie de milisecunda, mentionand informatii de fus orar, locale si permitand operatii complexe cu date calendaristice (adaugare/scadere de intervale etc). Clasa *Calendar* este abstracta (nu se poate instantia), in practica folosindu-se subclasele ei, in special *GregorianCalendar*
 - **GregorianCalendar** – subclasa a lui *Calendar* care implementeaza un calendar gregorian (cel folosit in ziua de astazi in majoritatea zonelor de pe glob)
- **DateFormat** – clasa ale carei metode permit atat aducerea la formatul dorit a unei date, cat si parsarea unui string ce reprezinta o data si transformarea sa intr-un obiect *Date* sau *Calendar*. In practica nu se foloseste direct aceasta clasa, ci subclase ale sale
 - **SimpleDateFormat** - subclasa a lui *DateFormat* ce permite specificarea facilă a formatului datei prin intermediul unor sechete speciale (detalii mai jos)

4.5.5.2. Clasa Date

Obiectele de tip *Date* memoreaza un moment de timp cu precizie de milisecunda. Intern, momentul de timp este stocat sub forma unei valori de tip *long* ce reprezinta numarul de milisecunde scurse de la 1 ian 1970 ora 00:00:00 (asa-numitul “Unix time”). Clasa contine multiple metode, insa majoritatea sunt marcate ca “deprecated” (depasite) in favoarea celor echivalente din clasa *Calendar*. Aceste metode vor fi retrase din API-ul clasei pe viitor si de aceea nici nu sunt mentionate in acest material.

Metodele non-deprecated din clasa *Date* permit programatorului un set de operatii destul de restrans:

- crearea unui obiect *Date* – se realizeaza cu ajutorul celor doi constructori:
 - **Date()** – creeaza un obiect *Date* ce incapsuleaza momentul de timp al crearii obiectului
 - **Date(long)** – creeaza un obiect *Date* ce incapsuleaza momentul de timp specificat ca argument, sub forma numarului de milisecunde scurse de la 1 ian 1970 ora 00:00:00
- obtinerea sau modificarea momentului de timp encapsulat
 - **long getTime()** – furnizeaza momentul de timp encapsulat sub forma de long (numar de milisecunde, ca mai sus)
 - **void setTime(long)** – schimba momentul de timp memorat intr-un obiect de tip *Date*
- compararea cu un alt obiect Date
 - **boolean after(Date)** – returneaza true daca momentul de timp al obiectului pentru care se apeleaza metoda este ulterior celui din obiectul primit ca argument
 - **boolean before(Date)** – returneaza true daca momentul de timp al obiectului pentru care se apeleaza metoda este anterior celui din obiectul primit ca argument
 - **boolean equals(Object)** – returneaza true daca obiectul pentru care se apeleaza metoda si cel primit ca argument indica momente de timp egale
 - **int compareTo(Date)** – returneaza un intreg negativ, nul sau pozitiv in functie de relatia dintre cele doua momente de timp comparate (vezi metoda *compareTo()* din orice alta clasa prezentata in acest material)

4.5.5.3. Clasele Calendar si GregorianCalendar

Asemanator cu *Date*, un obiect de tip *Calendar* memoreaza un anumit moment de timp, cu precizie de milisecunda. Intern, momentul de timp este stocat in doua moduri:

- sub forma numarului de milisecunde scurse de la 1 ian 1970 ora 00:00 pana la momentul curent de timp conform UTC (asa-numitul Unix time)
- sub forma urmatorului set de campuri: an, luna, zi, ora, minut, secunda, milisecunda (programatorul avand posibilitatea de a modifica fiecare camp individual)

Obiectul Calendar pastreaza sincronizate cele doua informatii, ajustand-o automat pe una dupa modificarea celeilalte.

In cadrul metodelor lui Calendar, fiecare dintre campuri este desemnat prin intermediul unei constante a clasei, dupa cum urmeaza:

- anul: **Calendar.YEAR**
- luna: **Calendar.MONTH** (atentie! valorile valide sunt 0-11!)
- ziua din luna - **Calendar.DATE** sau **Calendar.DAY_OF_MONTH**
- ora din zi, 0-23: **Calendar.HOUR_OF_DAY**
- ora din zi in format 1-12: **Calendar.HOUR**
- minut: **Calendar.MINUTE**
- secunda: **Calendar.SECOND**
- milisecunda: **Calendar.MILLISECOND**
- diverse alte campuri utile (ex: **Calendar.DAY_OF_YEAR**, **Calendar.AM_PM** folosit in conjunctie cu **Calendar.HOUR** etc.)

Diferenta fata de tipul de date *Date* este ca un obiect *Calendar* poate tine cont de fusul orar si de alte particularitati regionale, si in plus pune la dispozitia programatorului metode pentru extragerea sau setarea individuala a diferitelor componente ale datei incapsulate, pentru adaugarea sau scaderea unor intervale de timp la valoarea incapsulata, luarea in calcul a anilor bisecti etc., dupa cum urmeaza:

- obiectele Calendar pot fi create pasand ca argumente in constructor “componentele” datei dorite (an, luna, zi etc). Iata cativa constructori in acest sens (nota: o parte dintre ei se mapeaza pe metode *set()* cu aceeasi semnatura):
 - **GregorianCalendar()** - produce un obiect ce incapsuleaza momentul de timp curent (cel al crearii sale)
 - **GregorianCalendar(int an, int luna, int zi)** – obiectul va avea data dorita, ora 00:00:00
 - **GregorianCalendar(int an, int luna, int zi, int ora, int minut)** – stabilirea momentului incapsulat cu precizie de minut
 - **GregorianCalendar(int an, int luna, int zi, int ora, int minut, int secunda)** – stabilirea timpului incapsulat cu precizie de secunda
- momentul de timp stocat intr-un obiect de tip *Calendar* poate setat sau extras cu ajutorul urmatoarelor metode:
 - **Date getTime()** - produce un obiect Date ce incapsuleaza momentul de timp al obiectului Calendar
 - **setTime(Date)** - memoreaza in obiectul Calendar momentul de timp specificat ca argument
 - **getTimeInMillis()** - returneaza un long ce reprezinta numarul de milisecunde scurse de la 1 ian 1970 ora 0
 - **setTimeInMillis()** - realizeaza operatia inversa metodei anterioare
- pot fi extrase sau setate in mod independent diversele componente ale datei incapsulate. Iata metode utile:
 - **int get(int camp)** – returneaza valoarea componentei dorite a datei incapsulate. Campul dorit se specifica sub forma unei constante din clasa Calendar
 - **void set(int camp, in valoare)** – modifica valoarea campului specificat. Exemplu: pentru a seta ora 16, scriem *c.set(Calendar.HOUR_OF_DAY,16)* unde *c* este o referinta la un obiect de tip Calendar
 - **void set(int an, int luna, int zi)** - seteaza data dorita, campurile corespunzatoare orei nefind alterate (daca aveau deja valori, acestea vor fi pastrate)
 - **void set(int an, int luna, int zi, int ora, int minut)** - seteaza momentul de timp cu precizie de minut, celelalte campuri pastrandu-si valoarea curenta
 - **void set(int an, int luna, int zi, int ora, int minut, int secunda)** - seteaza momentul de timp cu precizie de secunda
- obiectul Calendar memoreaza si timezone-ul (fusul orar), ceea ce permite conversii de timp intre diferite fusuri. Timezone-ul poate fi setat inca de la crearea obiectului sau ulterior:
 - **GregorianCalendar(TimeZone zone)** – creeaza un calendar ce memoreaza momentul de timp curent al fusului orar specificat
 - **void setTimeZone(TimeZone zone)** - seteaza fusul orar pentru un obiect *Calendar* deja existent
- clasa Calendar dispune de metode ce permit operatii temporale utile (ex: adaugari/scaderi de intervale de timp)

- **void add(int camp, int cantitate)** – adauga cantitatea specificata la campul dorit. Campul se specifica sub forma unei constante din clasa Calendar (vezi mai sus). Cantitatea poate fi si negativa. Data interna a obiectului se va modifica in consecinta, modificarea putand afecta si alte campuri decat cel vizat la apelarea metodei (ex: daca adunam 2 zile la 31 ianuarie vom obtine 2 februarie, asadar o modificare de zi si de luna)
- **void roll(int camp, int cantitate)** – functionalitate asemanatoare cu add, insa este modificat doar campul vizat, fara a le afecta pe celelalte
- alte metode utile
 - **setLenient(boolean)** - un obiect Calendar care este “lenient” va comuta automat pe o data valida chiar si atunci cand ii setam o data inexistentă. Spre exemplu, incercand sa setam luna ianuarie si ziua 41, calendarul va “umple” luna ianuarie (consumand 31 din cele 41 de zile) iar restul de zile ramase le va reporta pentru februarie; data rezultata va fi 10 februarie!
 - **boolean isLenient()** - raporteaza daca calendarul se afla in modul “lenient” (ajustare automata)

Nota: clasa GregorianCalendar contine si alti constructori/alte metode utile. Pentru detalii suplimentare consultati documentatia clasei.

```

Calendar c = Calendar.getInstance(); // crearea unui GregorianCalendar cu momentul curent
Date d = c.getTime(); // obtinerea momentului curent de timp al calendarului
int ora = c.get(Calendar.HOUR_OF_DAY); // extragerea orei curente (format 0-24)

// meninem ora curenta dar aducem obiectul la ora fixa (punem minutele si secundele pe 0)
c.set(Calendar.MINUTE,0);
c.set(Calendar.SECOND,0);

// setam data 21.12.2011, ora 18:37:45 in obiectul Calendar
c.set(2011,11,21,18,37,45); // decembrie este luna 11! alternativ putem folosi Calendar.DECEMBER

// care este data/ora cu 100 de ore mai tarziu?
c.add(Calendar.HOUR,100);System.out.println(c.getTime()); // Sun Dec 25 22:37:45 EET 2011

// daca am fi folosit roll() pentru aceeasi operatie, se modifica NUMAI campul de ora:
// el se “da peste cap” de 4 ori (4 x 24h) ramanand 4h diferenta fata de data initiala:
c.roll(Calendar.HOUR,100);System.out.println(c.getTime()); // Wed Dec 21 22:37:45 EET 2011

```

Atentie la urmatoarele aspecte!

- *Calendar.DATE* nu inseamna data calendaristica, ci ziua din luna! (echivalent cu *Calendar.DAY_OF_MONTH*)
- campul *MONTH* al clasei *Calendar* are valori intre 0 si 11, nu intre 1 si 12! Asadar ianuarie este luna 0, iar decembrie luna 11! Putem face abstractie de acest lucru daca folosim constantele special definite in clasa *Calendar* pentru lunile anului (*Calendar.JANUARY* cu valoarea 0, *Calendar.FEBRUARY* cu valoarea 1 etc)
- nu confundati *Calendar.HOUR_OF_DAY* (valori 0-23) cu *Calendar.HOUR* (care desemneaza ora in format 1-12, si care specifica ora exacta doar in colaborare cu *Calendar.AM_PM*)!
- zilele saptamanii (asa cum sunt ele intelese de campul *DAY_OF_WEEK* al calendarului) sunt prezente sub forma de constante in clasa *Calendar*: *Calendar.SUNDAY*, *Calendar.MONDAY* etc. **ATENTIE!** Ele sunt numerotate de la 1 la 7, insa 1 este duminica, 2 este luni etc. iar 7 este sambata!

4.5.5.4. Formatare si parsare de date: clasa *SimpleDateFormat*

4.5.5.4.1. Principii si utilizare

Clasa *SimpleDateFormat* faciliteaza doua operatii principale care tin de formatul unei date:

- **formatare** - dat fiind momentul de timp stocat intr-un obiect de tip *Date* sau *Calendar*, se doreste obtinerea unui String ce exprima data respectiva intr-un anume format (ex: 21.12.2011, 12/21/2011, 2011-12-21 11:45:32 etc)
- **interpretare** (“parsing”) - extragerea momentului de timp descris de un String. Este operatia inversa celei anterioare, prin care se obtine un obiect *Date* sau *Calendar*

Un obiect de tip *SimpleDateFormat* incapsuleaza inca de la creare un anumit format al datei, pe care il reutilizeaza apoi in toate operatiile ce implica obiectul. Din acest punct de vedere, obiectul este un fel de filtru presetat folosit atat la formatarea cat si la parsarea informatiei.

Nota: clasa `SimpleDateFormat` este derivata din `DateFormat` - o clasa abstracta, care nu se poate instantia. Clasa `DateFormat` dispune de o multitudine de metode statice ce permit crearea de obiecte de formatare cu diverse setari default, tinand cont de setarile regionale ("locale"); clasa `SimpleDateFormat` este utila pentru a stabili "de mana" formatul dorit, in afara acelor preset-uri.

Obiectele de tip `SimpleDateFormat` poate fi create cu ajutorul constructorilor clasei, dupa cum urmeaza:

- **SimpleDateFormat(String format)** - creeaza un obiect `SimpleDateFormat` care formateaza/parseaza date conform formatului specificat ca argument (vezi detalii mai jos)
- **SimpleDateFormat(String format, Locale l)** - idem, insa impunand anumite setari regionale. Astfel se pot obtine, spre exemplu, numele lunilor sau ale zilelor saptamanii in limba dorita, fara ca programatorul sa depuna eforturi suplimentare pentru a le defini in limba dorita
- **SimpleDateFormat()** - creeaza un obiect `SimpleDateFormat` care foloseste setarile default atat pentru locale cat si pentru formatul datei (al carei default depinde la randul sau de locale)

Formatul dorit se specifica sub forma unui sir compus din diferite caractere, in care fiecare litera mica sau mare are o anumita semnificatie (vezi anexa materialului pentru lista completa), iar restul de caractere sunt preluate ca atare. Iata cateva exemple de formate (si reprezentarile aferente) pentru data 20 ianuarie 2012 ora 14:38:

```
dd.MM.yyyy → 20.01.2012
dd/MM hh:mm → 20/01 14:38
EEE dd MMM yyyy → Wed 20 Jan 2012
```

Daca dorim ca reprezentarea datei sa contina o anumita secventa care include si litere, insa literele sa nu fie interpretate ca specifatori de format, este necesara incadrarea secventei in cauza intre apostroafe:

```
'Day of week:' EEEE → Day of week: Wednesday
```

Nota: pentru a obtine insusi apostroful, il vom dubla in cadrul pattern-ului (ex: 'Ziua este'" EEEE ")

Fiecare litera se poate repeta in cadrul pattern-ului ce descrie formatul, numarul de repetitii determinand forma in care este prezentata informatia. Repetitiile sunt in general relevante la operatiile de formatare si mai putin importante (sau chiar ignorante in unele cazuri) la operatiile de parsare. Iata principalele reguli ce guverneaza repetitiile:

- pentru informatii de tip numeric, numarul de repetitii indica numarul minim de cifre cu care va fi afisata valoarea. Valorile mai mici vor fi completate la stanga cu 0. Exemplu: conform formatului MM, luna decembrie este 12 iar luna ianuarie este 01
 - exceptie: pentru formatarea lunii calendaristice, daca se folosesc cel putin 3 litere (MMM) secventa va fi inlocuita cu numele lunii, nu cu valoarea ei numerica (ex: pentru ianuarie, M ar inseamna 1, MM ar inseamna 01, iar MMM ar inseamna Jan si nicidecum 001)
- pentru informatii de tip text, folosirea a cel mult 3 repetitii produce formatul scurt al informatiei in cauza (ex: pt miercuri, EEE produce Wed iar pentru luna martie MMM produce Mar); de la 4 repetitii in sus este produsa forma integrala (ex: EEEE produce Wednesday, iar MMMM produce March)

Nota: observam asadar ca, pentru luna, M sau MM inseamna formatare numérica, MMM inseamna numele scurt al lunii, iar de la MMMM in sus avem numele intreg al lunii in cauza.

Iata cateva exemple, puse la dispozitie in documentatia clasei `SimpleDateFormat`:

Date and Time Pattern	Result
"yyyy.MM.dd G 'at' HH:mm:ss z"	2001.07.04 AD at 12:08:56 PDT
"EEE, MMM d, ''yy"	Wed, Jul 4, '01
"h:mm a"	12:08 PM
"hh 'o''clock' a, zzzz"	12 o'clock PM, Pacific Daylight Time
"K:mm a, z"	0:08 PM, PDT
"yyyyy.MMMMM.dd GGG hh:mm aaa"	02001.July.04 AD 12:08 PM
"EEE, d MMM yyyy HH:mm:ss z"	Wed, 4 Jul 2001 12:08:56 -0700

```
"yyMMddHHmmssZ"          010704120856-0700
"yyyy-MM-dd'T'HH:mm:ss.SSSZ" 2001-07-04T12:08:56.235-0700
"yyyy-MM-dd'T'HH:mm:ss.SSSXXX" 2001-07-04T12:08:56.235-07:00
"YYYY-'W'ww-u"           2001-W27-3
```

Odata creat, un obiect de tip *SimpleDateFormat* ofera metode precum:

- **String format(Date)** - formateaza momentul de timp primit ca argument in concordanta cu pattern-ul curent al obiectului *SimpleDateFormat*
- **Date parse(String)** - creeaza un obiect *Date* pe baza unui moment de timp specificat ca *String*, al carui format trebuie sa corespunda cu cel curent al obiectului *SimpleDateFormat*
- **applyPattern(String)** - permite schimbarea formatului pentru un obiect *SimpleDateFormat* deja existent
- **setLenient(boolean)** - stabileste daca, la parsarea unei date, obiectul *SimpleDateFormat* va actiona in modul strict (generand erori pentru datele invalide) sau va incerca ajustarea automata a datei pentru a obtine una valida

4.5.5.4.2. Particularizari regionale - lucrul cu clasa Locale

Un obiect de tip *Locale* reprezinta o alaturare de setari ce oglindesc particularitatile unei anumite zone geografice sau culturale; cele mai importante dintre ele sunt limba si tara. De la o zona/tara la alta, putem avea diferente precum:

- limba vorbita. O implicatie imediata o reprezinta numele diferite ale zilelor saptamanii si ale lunilor anului
- reprezentari diferite ale numerelor (ex: separatorul zecimal poate fi virgula sau punct)
- reprezentari default diferite ale datei (ex: in Romania scriem data sub forma 20.01.2012, pe cand in alte tari aceeasi data este reprezentata sub forma 01/20/2012)

Un obiect de tip *Locale* se poate obtine in mai multe moduri, dintre care amintim:

- folosind una dintre constantele predefinite ale clasei *Locale* (ex: *Locale.US*, *Locale.FRENCH* etc). Există constante doar pentru tarile/limbile mai des intalnite
- prin intermediul constructorilor clasei *Locale*
 - **Locale(String limba)** - creeaza un obiect *Locale* specific limbii dorite. Limba trebuie specificata sub forma unui cod ISO 639-1 (de 2 litere) sau 639-2 (de 3 litere). In cazul in care pentru o limba exista ambele coduri, va fi folosit cel scurt
 - **Locale(String limba, String tara)** - creeaza un obiect *Locale* in care tara este precizata separat, prin intermediul codului sau ISO 3166. Util pentru limbile vorbite in mai multe tari

Nota: pentru a determina *Locale*-urile disponibile in sistemul pe care ruleaza masina virtuala exista metoda *Locale.getAvailableLocales()* care produce un array cu *Locale*-urile suportate.

```
SimpleDateFormat sdf = new SimpleDateFormat("EEEE", new Locale("ro"));
Date d = new GregorianCalendar(2012,0,17).getTime();
System.out.println(sdf.format(d));      // Ziua este marti
sdf.applyPattern("dd MMMM yyyy");
System.out.println(sdf.format(d));      // 20 ianuarie 2012
```

4.6. BIBLIOGRAFIE

- Calificatorul static, initializarea campurilor
 - Java Tutorial:
 - <http://docs.oracle.com/javase/tutorial/java/javaOO/classvars.html>
 - <http://docs.oracle.com/javase/tutorial/java/javaOO/initial.html>
- Tipuri de date enumerate in Java:
 - Enum-uri (Java Tutorial): <http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html>
 - Exemple explicate de enum-uri: <http://www.deepakgaikwad.net/index.php/2009/07/08/enums-in-java-5-code-examples.html>
 - Java Language Specification - enum types:
http://java.sun.com/docs/books/jls/third_edition/html/classes.html#8.9
 - Interfata Enum: <http://docs.oracle.com/javase/7/docs/api/java/lang/Enum.html>
- Clase predefinite (documentatie API)

- Math: <http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html>
- Integer: <http://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html>
- Console: <http://docs.oracle.com/javase/7/docs/api/java/io/Console.html>
- String: <http://docs.oracle.com/javase/7/docs/api/java/lang/String.html>
- StringBuffer: <http://docs.oracle.com/javase/7/docs/api/java/lang/StringBuffer.html>
- StringBuilder: <http://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html>
- Date: <http://docs.oracle.com/javase/7/docs/api/java/util/Date.html>
- GregorianCalendar: <http://docs.oracle.com/javase/7/docs/api/java/util/GregorianCalendar.html>
- SimpleDateFormat: <http://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html>
- Coduri de limba conform ISO 639: http://www.loc.gov/standards/iso639-2/php/code_list.php
- Coduri de tara conform ISO 3166: http://ro.wikipedia.org/wiki/ISO_3166-1

4.7. Anexa 1 - specificatori de format pentru SimpleDateFormat

Letter	Date or Time Component	Presentation	Examples
G	Era designator	Text	AD
Y	Year	Year	1996; 96
Y	Week year	Year	2009; 09
M	Month in year	Month	July; Jul; 07
w	Week in year	Number	27
W	Week in month	Number	2
D	Day in year	Number	189
d	Day in month	Number	10
F	Day of week in month	Number	2
E	Day name in week	Text	Tuesday; Tue
u	Day number of week (1 = Monday, ..., 7 = Sunday)	Number	1
a	Am/pm marker	Text	PM
H	Hour in day (0-23)	Number	0
k	Hour in day (1-24)	Number	24
K	Hour in am/pm (0-11)	Number	0
h	Hour in am/pm (1-12)	Number	12
m	Minute in hour	Number	30
s	Second in minute	Number	55
S	Millisecond	Number	978
z	Time zone	General time zone	Pacific Standard Time; PST; GMT-08:00
Z	Time zone	RFC 822 time zone	-0800
X	Time zone	ISO 8601 time zone	-08:00; -0800; -08:00