



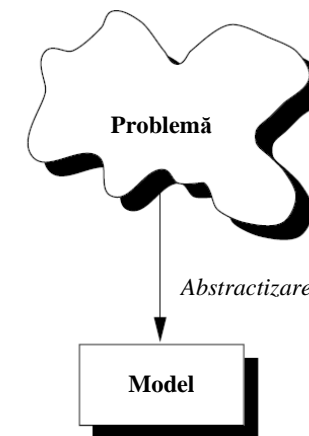
Programare orientată pe obiecte

- 2. Abstracțiuni și tipuri de date abstracte
- 3. Java



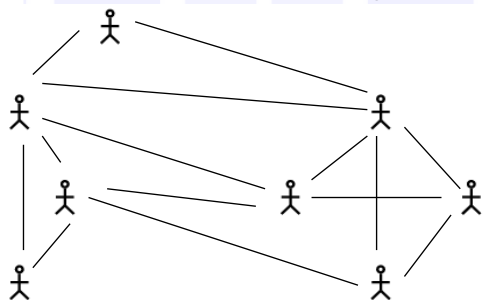
Abstractizare

- **Abstractizare** înseamnă eliminare sau ascunderea deliberată a unor detalii ale unui proces sau artefact pentru a releva mai clar alte aspecte, detalii sau structura.



Nivele de abstractizare în programe OO

- La nivelul cel mai înalt, privim un program ca o *comunitate de obiecte care interacționează*.

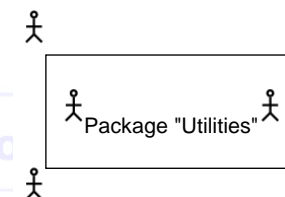


- Aici caracteristicile importante sunt liniile de *comunicare* dintre diferiții agenți.



Abstractizare în limbaje OO. Packages, namespaces

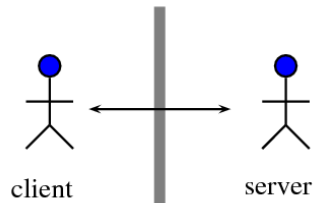
- Următorul nivel de abstractizare se regăsește în unele limbaje OO. Un *package (pachet)*, *unit* sau *namespace (spațiu de nume)* permite programatorului să înconjoare o colecție de obiecte (o comunitate mică în ea însăși) cu un strat și să-i controleze vizibilitatea din afara modulului.
 - Java are *packages*





Abstractizare în limbaje OO. Clienți și servitori

- Următorul nivel de abstractizare consideră *relațiile dintre obiectele individuale*. Tipic, unul oferă un serviciu (*server*), iar celălalt folosește serviciul (*client*).



Abstractizare în limbaje OO. Interfețe

- Putem examina acum doar agentul care oferă serviciul, independent de client. *Definim natura* serviciilor oferite, dar *nu și cum se realizează* acele servicii.
- Interfețele** sunt o cale de descriere a serviciilor la acest nivel de abstractizare.

```

interface Stack {
    public void push (Object val);
    public Object top () throws EmptyStackException;
    public void pop () throws EmptyStackException;
}
  
```

- Notă. Operații auxiliare cu tipul stivă:**
 - integer size(): returnează numărul de elemente stocate
 - boolean isEmpty(): indică dacă stiva este goală



Nivele de abstractizare. O implementare

- Apoi privim serviciile oferite din perspectiva implementării lor:

```

public class ArrayStack implements Stack ... {
    public void pop () throws EmptyStackException { ... }
    ...
}
  
```

- Aici grija este pentru *abordarea de nivel înalt* a modului de furnizare a serviciului respectiv.



Nivele de abstractizare. O metodă luată izolat

- În final, considerăm implementarea fiecărei metode luată separat.

```

public class ArrayStack implements Stack ... {
    ...
    public Object pop() throws EmptyStackException {
        if isEmpty() throw new EmptyStackException
            ("Empty stack: cannot pop");
        Object temp = S[top];
        S[top] = null;
        top = top - 1;
        return temp;
    }
    ...
}
  
```

- Fiecare nivel este important și adesea trecem repede înainte și înapoi pe nivele.



Aflarea nivelului de abstractizarea corect

- O problemă critică în stadiile de dezvoltare incipiente:
 - să determinăm ce detalii sunt *potrivite* la fiecare nivel de abstractizare și, (adesea mai important)
 - ce detalii trebuie *omise*.
- Evident, nimeni nu dorește
 - să elimine informații importante, dar
 - nici să gestioneze prea multe informații, sau ca volumul informației să ascundă detalii critice.



Abstractizarea Is-a (este-o) și Has-A (are-o)

- Divizarea în părți – abstractizarea *Are-o*
 - cunoscută și sub numele de *agregare* – clasa *conține* altă/alte structuri
- Divizarea prin specializare – abstractizarea *Este-o*
 - cunoscută și sub numele de *specializare* - noua clasă este o *particularizare* a uneia existente
 - inversul specializării: *generalizare*



Abstractizarea Are-o

- **Are-o**: ia un sistem complex și îl împarte în părți componente, care pot fi tratate individual.
 - Caracterizată de propoziții care au cuvintele "are-o", (are-un,) "conține", "cuprinde", "compusă din"
 - Ne permite să scădem nivelul de complexitate atunci când considerăm componentele izolate.
- Exemple
 - Un autovehicul *are-un* motor și *are-o* transmisie
 - O bicicletă *are-o* roată ☺
 - O fereastră (afișată pe ecran) *are-o* bară de meniu



Abstractizarea Este-o

- Abstractizarea **Este-o** ia un sistem complex și îl vede ca o instanță a unei abstracțiuni mai generale.
 - Caracterizată de propoziții care conțin cuvintele "este-o" (este-un)
 - Ne permite să clasificăm artefactele și informația și să o facem aplicabilă la multe situații diferite.
- Exemple
 - Un autoturism *este un* vehicul cu roți, care *este un* mijloc de transport
 - O bicicletă *este un* vehicul cu roți
 - O căruță cu cai *este un* mijloc de transport



Încapsulare și interschimbabilitate

- Un aspect important a diviziunii în părți este caracterizarea conexiunii (interfeței) dintre componente.
- Ne permite să considerăm multiple implementări diferite ale aceleiași interfețe.
 - Spre exemplu, un automobil poate avea mai multe tipuri diferite de motoare și un tip de transmisie



Vederea ca serviciu

- O interfață poate fi privită și ca o modalitate de descriere a *serviciilor* pe care le oferă un obiect.
- *Interfața* este un *contract* pentru servicii – dacă interfața este suportată, atunci serviciul va fi furnizat conform descrierii.

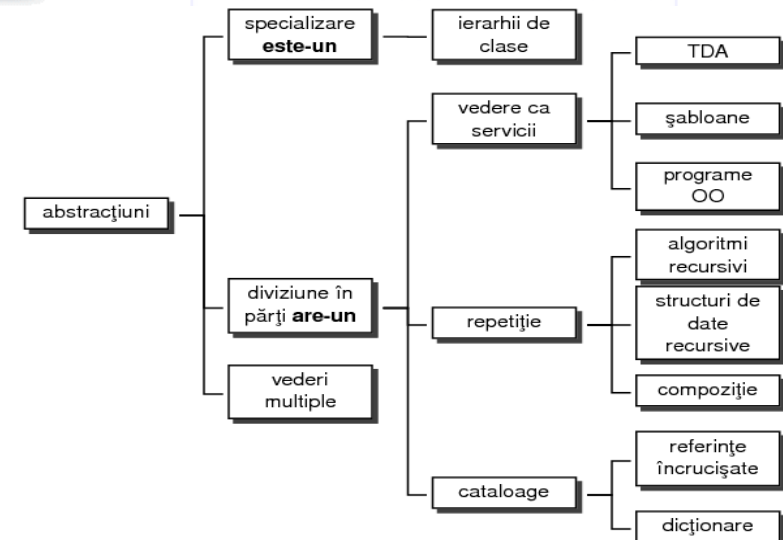


Alte tipuri de abstracțiuni. Compoziția

- **Compoziția** este un alt exemplu de abstracțiune importantă; ea este o formă de are-o, caracterizată de următoarele
 - Forme primitive
 - Reguli de combinare de valori pentru a obține noi valori
 - Ideea că noile valori pot fi și ele supuse combinării
- Exemple: sistemele de tipuri, ferestre, multe alte sisteme complexe.



Forme generale de abstractizare





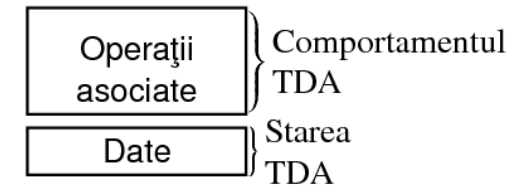
Tip de dată abstract

- Un tip de dată abstract (TDA) constă din:
 - O interfață – un set de operații care se pot efectua (cunoscută și sub numele de *barieră de abstractizare*).
 - Comportamentele permise – modul în care instanțele TDA răspund la operații.
- Implementarea unui TDA constă din:
 - O reprezentare internă – date stocate în variabile instanța ale obiectului
 - Un set de metode care implementează interfața.
 - Un set de invarianți ai reprezentării, adevărați inițial și conservați de toate metodele

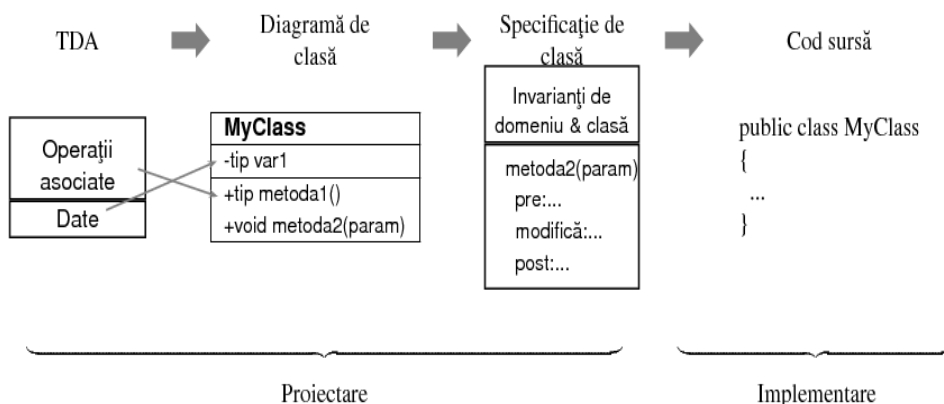


Tip de dată abstract

- Un **tip de data abstract** (TDA)
 - reprezintă un tip pentru încapsularea datelor înrudite
 - este abstract în sensul ascunderii detaliilor de implementare



De la TDA la cod sursă



Exemplu de specificație a unui TDA

TIP

TipTimp

DOMENIU

Fiecare valoare reprezintă un moment de timp în ore, minute și secunde.

OPERAȚII

Setează timpul

Afișează timpul

Incrementează timpul cu o secundă

Compară 2 timpi dacă sunt egali

Determină dacă un timp este "mai mic decât" un altul

TimeType
-hour: Integer
-min: Integer
-sec: Integer
+set(hour: Integer, min: Integer, sec: Integer)
+increment()
+print()
+equals(time: TimeType)
+isLessThan(time: TimeType)



Exemplu: TDA stivă

- Operatori abstracți
 - create :: Void → Stack a
 - destroy :: Stack a → Void
 - is_empty :: Stack a → boolean
 - top :: Stack → Element
 - push :: (Stack, Element) → Stack
 - pop :: Stack → Stack
- *Invariant*
 - pop(push (stack, e)) = stack
- Acum știm cum se implementează stiva. Ceea ce trebuie să respectăm este comportamentul datelor potrivit funcțiilor definite.



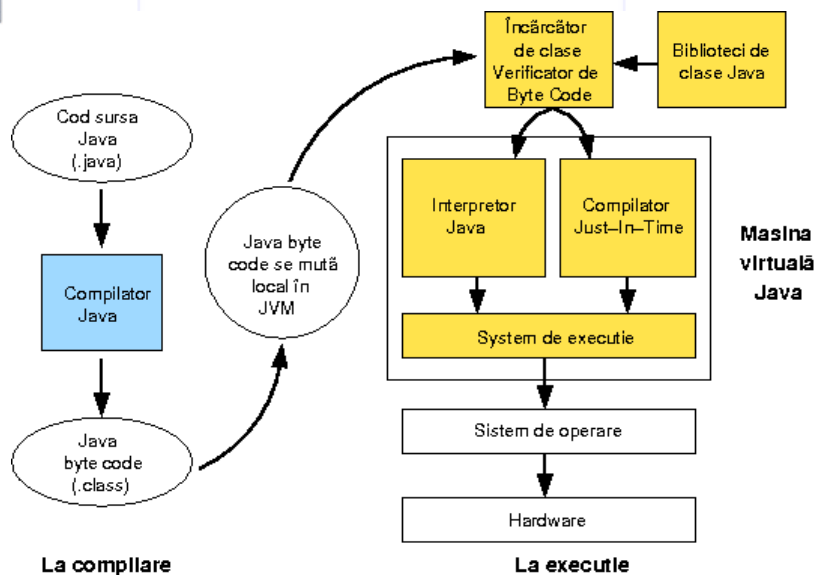
Java. Caracteristici

- James Gosling și Patrick Naughton (conducătorii echipei care a dezvoltat limbajul), au definit limbajul Java ca fiind

"Un limbaj simplu, orientat pe obiecte, care "înțelege" rețelele de calculatoare, interpretat, robust, sigur, neutru față de arhitecturi, portabil, de înaltă performanță, multi-fir, dinamic".



Mediul Java



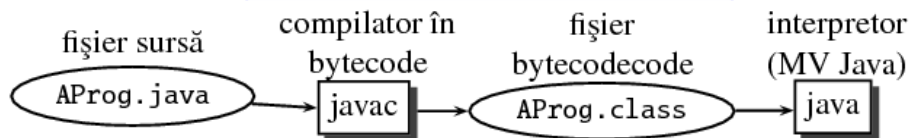
Executarea programelor Java

- Java folosește un proces în doi pași
- Compilează programul în *bytecodes*
 - bytecode este apropiat de formatul instrucțiunilor în limbaj mașină, dar nu chiar la fel — este un "limbaj mașină" a generic
 - nu corespunde nici unui procesor real
- *Mașina virtuală* (VM) interpretează bytecode în limbaj mașină nativ și-l rulează
 - există mașini virtuale diferite pentru calculatoare diferite, deoarece bytecode nu corespunde unei mașini reale



Executarea programelor Java

- Se utilizează *același* bytecode Java pe *calculatoare diferite* fără a recompila codul sursă
 - fiecare VM interpretează același bytecode Java
 - aceasta permite să se execute programe Java prin simpla obținere a bytecodes din pagini de Web
- Aceasta face codul Java să ruleze *peste-platforme*
 - marketing-ul spune, "Scrie o dată, rulează oriunde!"
 - adevărat pentru "Java pur", nu pentru variante



OOP2 - M. Joldoș - T.U. Cluj

25



Aplicații Java

- Tipuri de aplicații Java:
 - *aplicații de sine-stătătoare* și *applets/servlets*
- Un *aplicație de sine-stătătoare* sau un program "obișnuit" este o clasă care are o metodă numită **main**
 - Când se lansează programul Java respectiv, *sistemul de execuție* invocă automat metoda numită **main**
 - Toate *aplicațiile de sine-stătătoare* încep din metoda **main**
- Deja ați văzut asemenea aplicații în exemplele BlueJ

OOP2 - M. Joldoș - T.U. Cluj

26



Applets

- Un *applet* Java (o aplicație având caracteristici limitate, care necesită resurse de memorie limitate și portabilă între sistemele de operare) este un program Java destinat a fi rulat dintr-un browser de Web
 - poate fi rulat dintr-o locație de pe Internet
 - poate fi rulată și din vizualizatorul de applet-uri (**appletviewer**) pentru depanare
 - Applet-urile folosesc întotdeauna o interfață cu ferestre
- Aplicațiile de sine stătătoare pot folosi fie interfața cu ferestre sau I/O consolă (adică în modul text)
- Deja ați văzut un applet în exemplele BlueJ

OOP2 - M. Joldoș - T.U. Cluj

27



Încărcătorul de clase

- Programele Java sunt divizate în unități mai mici numite *clase*
 - Fiecare definiție de clasă este în mod normal într-un fișier separat și compilată separat
- *Încărcătorul de clase*: un program care leagă bytecod-ul claselor necesare pentru a rula un program Java
 - în alte limbaje de programare, corespondentul său este *editorul de legături* (*linkeditor*)

OOP2 - M. Joldoș - T.U. Cluj

28



Cuvinte cheie Java (cuvinte rezervate)

- Cuvinte care nu pot fi folosite la altceva decât în modul predefinit din limbaj
 - *abstract, assert, 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*
 - *null, true, false* – predefinite ca literali



Compilarea unui program sau a unei clase Java

- Fiecare definiție de clasă trebuie să se afle într-un fișier al cărui nume este numele clasei urmat de extensia **.java**
 - Exemplu: Clasa **UnProgram** trebuie să se afle în fișierul numit **UnProgram.java**
- Fiecare clasă este compilată folosind comanda **javac** urmată de numele fișierului care conține clasa


```
javac UnProgram.java
```

 - Rezultatul este un program în byte-code cu același nume ca al clasei, urmat de extensia **.class**

```
UnProgram.class
```



Rularea unui program Java

- Un program Java poate fi *rulat* (**java**) după ce i-au fost compilate toate clasele
 - Rulați doar clasa care conține o metodă **main** (sistemul va încărca și rula celelalte clase automat, dacă mai sunt)
 - Metoda **main** începe cu linia:


```
public static void main(String[ ] args)
```
 - Comanda de lansare trebuie urmată doar de numele clasei (fără extensii)


```
java UnProgram
```
- Vedeți **Hello.java** din exemplele BlueJ



Convenții pentru nume

- Începeți numele de variabile, metode și obiecte cu o literă mică, indicați limitele "cuvintelor" cu o litera mare și pentru celelalte caractere folosiți doar litere și cifre ("camelcase")


```
vitezaMaxima rataDobinzii oraSosirii
```
- Începeți numele de clase cu majusculă și pentru restul identificatorului aplicați regula de mai sus


```
UnProgram OClasa String
```




Declararea variabilelor

- Fiecare variabilă dintr-un program Java trebuie *declarată* înainte de utilizare
 - Declarația informează compilatorul asupra tipului de dată care va fi stocat în variabilă
 - Tipul variabilei este urmat de unul sau mai multe nume separate de virgule și terminat cu punct și virgulă
 - Variabilele se declară de obicei chiar înainte de folosire sau la începutul unui bloc (indicat de o acoladă deschisă {)
 - Tipurile simple în Java sunt numite *tipuri primitive*

```
int numarulDeCai;
double oLungime, lungimeaTotala;
```

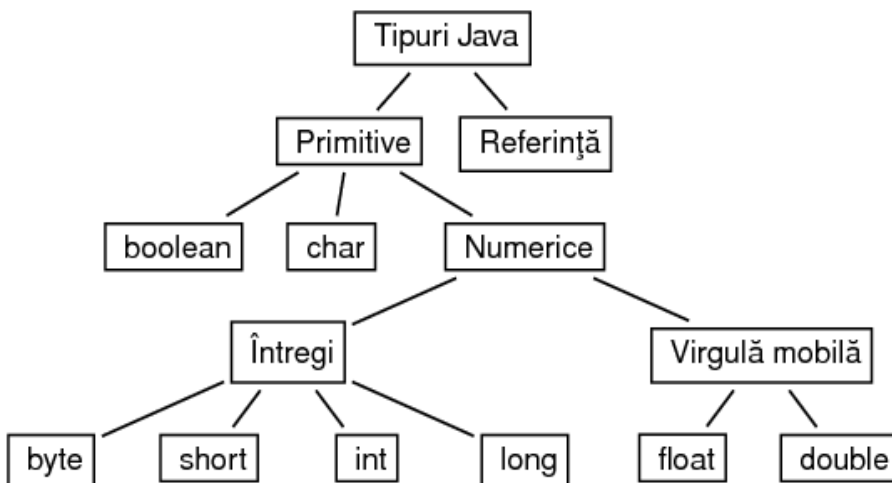


Modificatori de acces pentru variabile/metode

- **private**
 - variabila/metoda este vizibilă local în cadrul clasei.
 - "Vizibilă doar mie".
- **protected**
 - variabila/metoda poate fi văzută din toate clasele, subclassele și celelalte clase din același pachet (package).
 - "Vizibilă în familie".
- **public**
 - variabila/metoda poate fi văzută din toate clasele
 - "Vizibilă tuturor".
- Modificatorul de acces implicit, nu are cuvânt cheie.
 - **public** pentru ceilalți membrii din același pachet.
 - **private** pentru oricine din afara pachetului
 - numită și *acces în pachet*.
 - "Vizibilă în vecinătate"



Tipuri Java



Tipuri primitive

Tip primitiv	Biți	Minimum	Maximum	Wrapper type
boolean	—	—	—	Boolean
char	16-bit	Unicode 0	Unicode $2^{16} - 1$	Character
byte	8-bit	-128	+127	Byte
short	16-bit	-2^{15}	$+2^{15} - 1$	Short
int	32-bit	-2^{31}	$+2^{31} - 1$	Integer
long	64-bit	-2^{63}	$+2^{63} - 1$	Long
float	32-bit	IEEE754	IEEE754	Float
double	64-bit	IEEE754	IEEE754	Double
void	—	—	—	Void

Toate tipurile numerice sunt cu semn.



Compatibilitate la asignare

- Mai general, o valoare de orice tip din lista următoare poate fi asignată unei variabile de orice alt tip care apare la dreapta ei

byte → short → int → long → float → double
char →

- Gama valorilor de la dreapta este mai largă
- Este necesară o conversie de tip explicită (*type cast*) pentru a asigna o valoare de un tip la o variabilă care apare la stânga ei în lista de mai sus (d.e., `double` la `int`)
- Observați că în Java un `int` nu poate fi asignat la o variabilă de tip `boolean`, nici un `boolean` la o variabilă de tip `int`



Operatori aritmetici și expresii

- Ca în majoritatea limbajelor, și în Java se pot forma *expresii* folosind variabile, constante și operatori aritmetici
 - Operatori aritmetici; `+` (adunare), `-` (scădere), `*` (înmulțire), `/` (împărțire), `%` (modulo, rest)
 - Se poate folosi o expresie oriunde este legal să se folosească o valoare de tipul produs de expresie

Computer Science



Operatori aritmetici și expresii

- Dacă se combină un operator aritmetic cu operanzi de tipul `int`, atunci tipul rezultat este `int`
- Dacă se combină un operator aritmetic cu unul sau doi operanzi de tipul `double`, atunci tipul rezultat este `double`
- La combinarea de operanzi de tip diferit, tipul rezultat este cel mai din dreapta din lista de mai jos care se află în expresie

byte → short → int → long → float → double
char →

- Excepție: Dacă tipul rezultat este `byte` sau `short` (potrivit regulii date), atunci tipul produs va fi de fapt un `int`



Reguli de precedență și asociativitate

- La determinarea ordinii operațiilor adiacente, operația cu precedență mai mare (și argumentele sale aparente) este grupată înaintea operației de precedență mai mică

`base + rate * hours` se evaluează ca
`base + (rate * hours)`
- La precedență egală, ordinea operațiilor este determinată de regulile de *asociativitate*



Reguli de precedență și asociativitate

- Operatorii *unari* de precedență egală sunt grupați de la dreapta la stânga
`++rate` se evaluează ca `+(-(rate))`
- Operatorii *binari* de precedență egală sunt grupați de la stânga la dreapta
`base + rate + hours` se evaluează ca
`(base + rate) + hours`
- Excepție: Un șir de operatori de asignare este grupat de la dreapta la stânga
`n1 = n2 = n3;` se evaluează ca `n1 = (n2 = n3);`



Posibilă problemă: Erorile de rotunjire la numerele în virgulă mobilă

- Numerele în virgulă mobilă sunt, în general, doar valori aproximative
 - Matematic, numărul în virgulă mobilă 1.0/3.0 este egal cu 0.3333333 . . .
 - Un calculator are o cantitate limitată de memorie
 - Poate stoca 1.0/3.0 ca ceva în genul lui 0.3333333333, puțin mai puțin decât o treime
 - De fapt numerele sunt stocate binar, dar consecințele sunt aceleași: numerele în virgulă mobilă pot pierde precizie



Împărțirea întregă și cea în virgulă mobilă

- Dacă unul sau amândoi operanzii sunt în virgulă mobilă, împărțirea dă un rezultat în virgulă mobilă
`15.0/2` se evaluează la `7.5`
- Cum ambii operanzi întregi, împărțirea dă un întreg
 - O eventuală parte fracționară este ignorată
 - Nu se fac rotunjiri
`15/2` se evaluează la `7`
- Aveți grijă ca cel puțin un operand să fie în virgulă mobilă dacă este nevoie de partea fracționară



Conversia de tip explicită

- O *conversie de tip explicită* (*type cast*) ia o valoare de un tip și produce o valoare "echivalentă" de celălalt tip
 - Dacă `n` și `m` sunt întregii de împărțit și e nevoie de partea fracționară, atunci cel puțin un operand trebuie să fie în virgulă mobilă **înainte** de efectuarea operației
`double ans = n / (double)m;`
 - La fel ca în C, tipul dorit este pus între paranteze imediat înaintea variabilei de convertit
 - Tipul și valoarea variabilei de convertit nu se schimbă



Conversia de tip explicită

- La conversia explicită de la virgulă mobilă la întreg, numărul este trunchiat, nu rotunjit
 - `(int)2.9` se evaluează la `2`, nu `3`
- La asignarea valorii unui întreg la o variabilă în virgulă mobilă, Java realizează o conversie explicită de tip automată numită *coerciție de tip*

```
double d = 5;
```
- Nu este legal să se atribuie un `double` la un `int` fără o conversie explicită


```
int i = 5.5; // Ilegal
int i = (int)5.5 // Corect
```



Operatorii increment și decrement

- Când oricare dintre operatorii `++` sau `--` precede o variabilă și este o parte a expresiei, expresia este evaluată folosind valoarea modificată a variabilei
 - Dacă `n` este `2`, atunci `2*(++n)` se evaluează la `6`
- Când oricare dintre operatori urmează unei variabile și este parte a expresiei, expresia este evaluată folosind valoarea originală și abia apoi se schimbă valoarea variabilei
 - Dacă `n` este `2`, atunci `2*(n++)` se evaluează la `4`



Clase "învelitoare" pentru tipurile primitive

- Clasele învelitoare (wrapper)* furnizează câte un tip clasă corespunzător fiecărui tip primitiv
 - Astfel este posibil să existe clase care se comportă cumva asemănător tipurilor primitive
 - Clasele pentru tipurile primitive `byte`, `short`, `long`, `float`, `double`, și `char` sunt (în ordine) `Byte`, `Short`, `Long`, `Float`, `Double`, și `Character`
- Aceste clase conțin și un număr de constante predefinite utile, precum și metode statice utile



Clase învelitoare

- Boxing (împachetare)*: procesul de trecere de la o valoare primitivă la un obiect al clasei sale învelitoare
 - Pentru a converti valoarea la una "echivalentă" de tip clasă, creați un obiect de tipul corespunzător folosind valoarea primitivă ca argument
 - Noul obiect va conține o variabilă instanță care stochează o copie a valorii primitive
 - Spre deosebire de alte clase, clasele învelitoare nu au constructori fără argumente

```
Integer integerObject = new Integer(42);
```



Clase învelitoare

- **Unboxing (despachetare):** procesul de trecere de la un obiect al unei clase învelitoare la valoarea corespunzătoare a unui tip primitiv
 - Metodele de conversie a unui obiect din clasele **Byte**, **Short**, **Integer**, **Long**, **Float**, **Double**, și **Character** la valorile primitive corespunzătoare sunt (în ordine) **byteValue**, **shortValue**, **intValue**, **longValue**, **floatValue**, **doubleValue**, și **charValue**
 - Nici una dintre aceste metode nu necesită argumente


```
int i = integerObject.intValue();
```



Boxing și Unboxing automat

- Începând cu versiunea 5.0, Java poate face automat împachetare/despachetare
- În loc să creăm un obiect din clasa învelitoare (ca mai înainte), se poate face o conversie de tip automată:


```
Integer integerObject = 42;
```
- În loc să trebuiască să invocăm metoda corespunzătoare (ca **intValue**, **doubleValue**, **charValue** etc.) pentru a converti un obiect din clasa învelitoare la tipul său asociat, valoarea primitivă poate fi recuperată automat


```
int i = integerObject;
```



Constante și metode statice în clasele învelitoare

- Constante pentru valori maxime și minime pentru tipurile primitive
 - Exemple: **Integer.MAX_VALUE**, **Integer.MIN_VALUE**, **Double.MAX_VALUE**, **Double.MIN_VALUE** etc.
- Clasa **Boolean** are nume pentru două constante de tipul **Boolean**
 - **Boolean.TRUE** și **Boolean.FALSE** sunt obiectele booleene corespunzătoare valorilor **true** și **false** ale tipului primitiv **boolean**



Constante și metode statice în clasele învelitoare

- Metode statice pentru conversia unei reprezentări corect formate ca și de caractere a unui număr la numărul de un tip dat
 - Metodele **Integer.parseInt**, **Long.parseLong**, **Float.parseFloat** și **Double.parseDouble** sunt pentru (în ordine) **int**, **long**, **float** și **double**
- Metode statice pentru conversia duală celei anterioare
 - Exemplu. Expresia

```
Double.toString(123.99);
```

 returnează valoarea șir "123.99"
- Clasa **Character** conține un număr de metode utile la prelucrarea șirurilor



Referințe Java

- Toate obiectele sunt create în memorie
- Pentru a accesa datele dintr-un obiect sau a lucra cu un obiect, este nevoie de o variabilă pe stivă, variabila care poate stoca o referință la adresa obiectului.
- Despre variabilele care stochează referințe la adrese de obiecte se spune că păstrează tipul de date *referință*
- Exemplu:

```
PlatitorTaxe t = new
    PlatitorTaxe(111111120633, 30000);
```



Clasa string

- Nu există un tip primitiv pentru șiruri în Java
- Clasa `String` este o clasă Java predefinită folosită la stocarea și prelucrarea șirurilor
- Obiectele de tipul `String` sunt compuse din șiruri de caractere scrise între ghilimele
 - Orice șir între ghilimele este de clasa `String`
- Unei variabile de tipul `String` i se poate da valoarea unui obiect `String`

```
String binecuvintare = "Fiti fericiti cu
    Java.";
```



Concatenarea șirurilor

- *Concatenare*: folosind operatorul `+` operator aplicat asupra a două șiruri pentru a le conecta și a forma un șir mai lung
 - Dacă `salut` este egal cu `"Bună ziua, "`, și `javaClass` este egal cu `"viitori colegi"`, atunci `salut + javaClass` este egal cu `"Bună ziua, viitori colegi"`
- Orice număr de șiruri pot fi concatenate
- La combinarea unui șir cu aproape orice alt tip, rezultatul este un șir
 - `"Răspunsul este " + 42` se evaluează la `" Răspunsul este 42"`



Metode al clasei string String

- Clasa `String` conține multe metode utile la aplicațiile care prelucrează șiruri
 - O metodă a clasei `String` se apelează prin numele unui obiect `String`, un punct, numele metodei și o pereche de paranteze între care sunt cuprinse argumentele (dacă sunt)
 - O valoare returnată de o metodă `String` se poate folosi oriunde se poate folosi o valoare de tipul returnat

```
String greeting = "Hello";
int count = greeting.length();
System.out.println("Length is " +
    greeting.length());
```

- *Poziția* sau *indexul* unui caracter într-un șir se numără întotdeauna de la zero



Metode ale clasei string

Lungimea șirului

<code>i =</code>	<code>s.length()</code>	lungimea șirului <code>s</code> .
------------------	-------------------------	-----------------------------------

Comparații (note: folosiți aceste metode în loc de `==` și `!=`)

<code>i =</code>	<code>s.compareTo(t)</code>	compară cu <code>s</code> . Returnează <code><0</code> dacă <code>s<t</code> , <code>0</code> dacă <code>==</code> , <code>>0</code> dacă <code>s>t</code>
<code>i =</code>	<code>s.compareToIgnoreCase(t)</code>	la fel ca mai sus, dar fără a ține seama de majuscule/minuscule
<code>b =</code>	<code>s.equals(t)</code>	true dacă cele două șiruri au valori egale
<code>b =</code>	<code>s.equalsIgnoreCase(t)</code>	la fel ca mai sus, dar fără a ține seama de majuscule/minuscule
<code>b =</code>	<code>s.startsWith(t)</code>	true dacă <code>s</code> începe cu <code>t</code>
<code>b =</code>	<code>s.startsWith(t, i)</code>	true dacă <code>t</code> apare începând cu indexul <code>i</code>
<code>b =</code>	<code>s.endsWith(t)</code>	true dacă <code>s</code> se termină cu <code>t</code>



Metode ale clasei string

Căutare - Notă : Toate metodele "indexOf" returnează -1 dacă șirul/caracterul nu este găsit

<code>i =</code>	<code>s.indexOf(t)</code>	indexul primei apariții lui String <code>t</code> în <code>s</code> .
<code>i =</code>	<code>s.indexOf(t, i)</code>	indexul lui String <code>t</code> la sau după poziția <code>i</code> în <code>s</code> .
<code>i =</code>	<code>s.indexOf(c)</code>	indexul primei apariții caracterului <code>c</code> în <code>s</code> .
<code>i =</code>	<code>s.indexOf(c, i)</code>	indexul caracterului <code>c</code> la sau după poziția <code>i</code> din <code>s</code> .
<code>i =</code>	<code>s.lastIndexOf(c)</code>	indexul ultimei apariții lui <code>c</code> în <code>s</code> .
<code>i =</code>	<code>s.lastIndexOf(c, i)</code>	indexul ultimei apariții lui <code>c</code> pe sau înainte de poziția <code>i</code> în <code>s</code> .
<code>i =</code>	<code>s.lastIndexOf(t)</code>	indexul ultimei apariții lui <code>t</code> în <code>s</code> .
<code>i =</code>	<code>s.lastIndexOf(t, i)</code>	indexul ultimei apariții a lui <code>t</code> pe sau înainte de poziția <code>i</code> în <code>s</code> .



Metode ale clasei string

Obținerea de părți

<code>c =</code>	<code>s.charAt(i)</code>	caracterul de la poziția <code>i</code> din <code>s</code> .
<code>s1 =</code>	<code>s.substring(i)</code>	subșirul de la indexul <code>i</code> până la sfârșitul lui <code>s</code> .
<code>s1 =</code>	<code>s.substring(i, j)</code>	subșirul de la indexul <code>i</code> până ÎNAINTE de indexul <code>j</code> din <code>s</code> .

Crearea unui nou șir din original

<code>s1 =</code>	<code>s.toLowerCase()</code>	nou String cu toate caracterele minuscule
<code>s1 =</code>	<code>s.toUpperCase()</code>	nou String cu toate caracterele majuscule
<code>s1 =</code>	<code>s.trim()</code>	nou String fără spații albe la început și sfârșit
<code>s1 =</code>	<code>s.replace(c1, c2)</code>	nou String cu toate <code>c2</code> -urile înlocuite prin <code>c1</code> .



Metode ale clasei string

Metode statice pentru conversia la String

<code>s =</code>	<code>String.valueOf(x)</code>	Converteste <code>x</code> la String, unde <code>x</code> este orice valoare de tip (primitiv sau obiect).
<code>s =</code>	<code>String.format(f, x...)</code>	[Java >=5] Folosește formatul <code>f</code> pentru a converti un număr variabil de parametri, <code>x</code> la un șir.

- Lista nu este exhaustivă.



Prelucrarea șirurilor

- Un obiect **String** nu se poate schimba (eng. immutable), caracterele pe care le conține nu pot fi schimbate
- Clasa **StringBuffer** are metode prin care se pot edita obiectele sale șir
- Se poate totuși modifica valoarea unei variabile **String** prin asignare

```
String nume = "Ionescu";
nume = "Ion " + nume;
```



Seturi de caractere

- **ASCII**: Set de caractere folosit de multe limbaje de programare, care conține toate caracterele folosite în mod normal pe o tastatură pentru limba engleză plus câteva caractere speciale
 - Fiecare caracter este reprezentat de un cod numeric
 - **Unicode**: Set de caractere folosit limbajul Java care include tot setul ASCII plus multe dintre caracterele folosite cu alfabetul elen
- Exemple:
- ```
char c = '\u0103'; // litera 'ă'
String s = "\u00eencoto\u015fm\u0103ni\u0163i"; // încotoșmăniți
```



## O porțiune din codul Unicode

|   | 000         | 001         | 002        | 003       | 004       | 005       | 006       | 007       |
|---|-------------|-------------|------------|-----------|-----------|-----------|-----------|-----------|
| 0 | NUL<br>0000 | DLE<br>0010 | SP<br>0020 | 0<br>0030 | @<br>0040 | P<br>0050 | `<br>0060 | p<br>0070 |
| 1 | SOH<br>0001 | DC1<br>0011 | !<br>0021  | 1<br>0031 | A<br>0041 | Q<br>0051 | a<br>0061 | q<br>0071 |
| 2 | STX<br>0002 | DC2<br>0012 | "<br>0022  | 2<br>0032 | B<br>0042 | R<br>0052 | b<br>0062 | r<br>0072 |
| 3 | ETX<br>0003 | DC3<br>0013 | #<br>0023  | 3<br>0033 | C<br>0043 | S<br>0053 | c<br>0063 | s<br>0073 |
| 4 | EOT<br>0004 | DC4<br>0014 | \$<br>0024 | 4<br>0034 | D<br>0044 | T<br>0054 | d<br>0064 | t<br>0074 |



## Numirea constantelor

- În loc de numere "anonime", declarați constante simbolice (cu nume) și folosiți-le numele
 

```
public static final double CM_PER_INCH = 2.54;
public static final int HOURS_PER_DAY = 24;
```

  - Previne schimbarea nedorită a valorii
  - Ușurează modificarea valorii
  - Convenția de nume pentru constante: toate literele majuscule, limitele de cuvinte marcate prin liniuța de subliniere



## Comentariile

- Comentariu de o *linie*: începe cu simbolurile `//` și provoacă ignorarea a ceea ce urmează până la sfârșitul liniei
  - Folosit de cel care scrie codul sau de cel care îl modifică
- Comentariul *bloc* este la fel ca în C (perechea `/*, */`)
  - Furnizează documentație utilizatorilor programului



## Documentarea programului

- Java include programul `javadoc` care extrage automat documentația din comentariile bloc din clasele definite, dacă
  - Începutul comentariului are un asterisc suplimentar (`/**`)
- Un program bine scris este autodocumentat
  - Structura sa se clarifică prin alegerea numelor de identificatori și modelul de indentare



## Rezumat

- Abstracțiuni în programele OO
  - nivele: pachet, client/server, interfețe, implementări, metode (considerate separat)
  - Are-o (agregare)
  - Este-o (specializare)
  - Compoziție
  - TDA
- Java
  - mediul (compilator, încărcător de clase, interpretor, JIT, sistem de execuție etc.)
  - convenții de numire
  - declarații și modificatori de acces
  - Tipuri primitive (boolean, char, short, int, long, float, double) și clase învelitoare
  - Operatori aritmetici
  - Clasa String