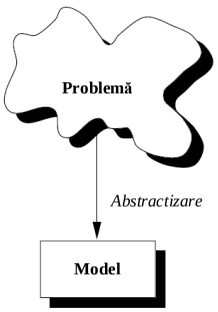
Programare orientată pe obiecte

2. Abstracţiuni şi tipuri de date abstracte
3. Java

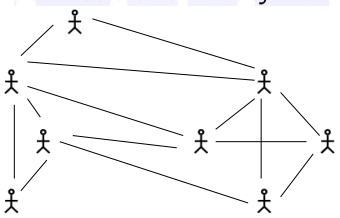
Abstractizare

- **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

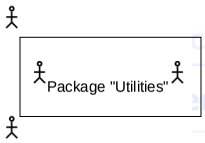
- La nivelul cel mai înalt, privim un program ca o *comunitate de obiecte care interacţionează*.



- Aici caracteristicile importante sunt liniile de *comunicare* dintre diferiţii agenţi.

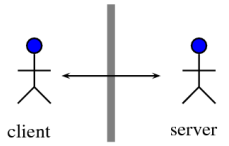
Abstractizare în limbaje OO. Packages, namespaces

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



Abstractizare în limbaje OO. Clienţi şi servitori

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



Abstractizare în limbaje OO. Interfeţe

- Putem examina acum doar agentul care oferă serviciul, independent de client. *Definim natura* serviciilor oferite, dar *nu şi cum se realizează* acele servicii.
- **Interfeţele** 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. Operaţii 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 şi adesea trecem repede înainte şi 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 şi, (adesea mai important)
  - ce detalii trebuie *omise*.
- Evident, nimeni nu doreşte
  - să elimine informaţii importante, dar
  - nici să gestioneze prea multe informaţii, sau ca volumul informaţiei să ascundă detalii critice.

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

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

Abstractizarea Are-o

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

Abstractizarea Este-o

- Abstractizarea **Este-o** ia un sistem complex şi îl vede ca o instanţa a unei abstracţiuni mai generale.
  - Caracterizată de propoziţii care conţin cuvintele "este-o" (este-un)
  - Ne permite să clasificăm artefactele şi informaţia şi să o facem aplicabilă la multe situaţii diferite.
- Exemple
  - Un autoturism *este un* vehicul cu roţi, care *este un* mijloc de transport
  - O bicicletă *este un* vehicul cu roţi
  - O căruţă cu cai *este un* mijloc de transport

Încapsulare şi interschimbabilitate

- Un aspect important a diviziunii în părţi este caracterizarea conexiunii (interfeţei) dintre componente.
- Ne permite să considerăm multiple implementări diferite ale aceleiaşi interfeţe.
  - Spre exemplu, un automobil poate avea mai multe tipuri diferite de motoare şi un tip de transmisie

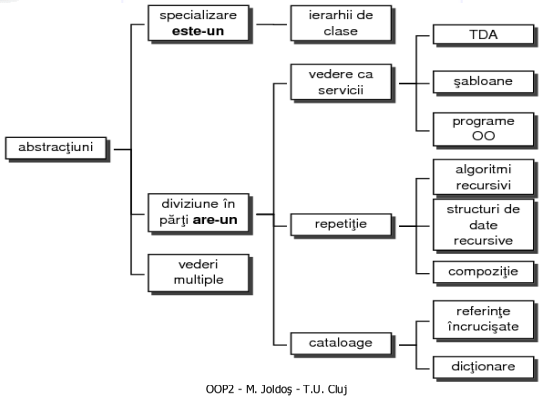
Vederea ca serviciu

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

Alte tipuri de abstracţiuni. Compoziţia

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

Forme generale de abstractizare

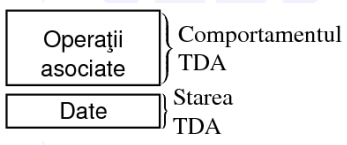


Tip de dată abstract

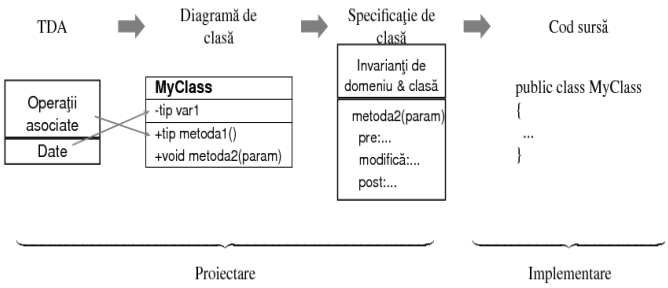
- Un tip de dată abstract (TDA) constă din:
  - O interfaţă – un set de operaţii care se pot efectua (cunoscută şi sub numele de *barieră de abstractizare*).
  - Comportamentele permise – modul în care instanţele TDA răspund la operaţii.
- Implementarea unui TDA constă din:
  - O reprezentare internă – date stocate în variabile instanţa ale obiectului
  - Un set de metode care implementează interfaţa.
  - Un set de invarianţi ai reprezentării, adevăraţi iniţial şi conservaţi 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 specificaţie a unui TDA

TIP

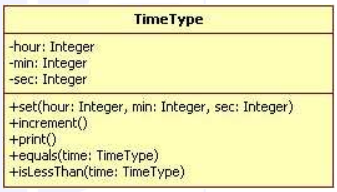
TipTimp

DOMENIU

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

OPERAŢII

Setează timpul
Afişează timpul
Incrementează timpul cu o secundă
Compară 2 timpi dacă sunt egali
Determină dacă un timp este "mai mic decât" un altul



Exemplu: TDA stivă

- Operatori abstracţi
  - 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

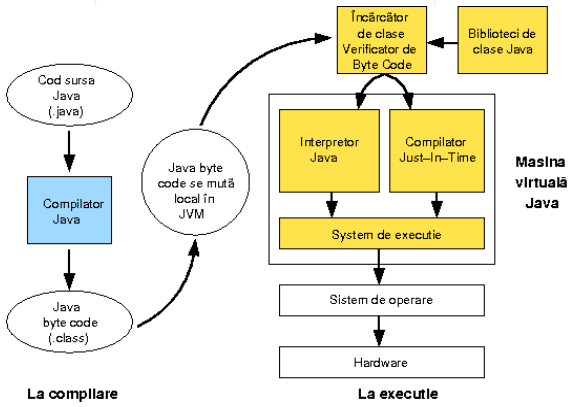
- Acum ştim cum se implementează stiva. Ceea ce trebuie să respectăm este comportamentul datelor potrivit funcţiilor definite.

Java. Caracteristici

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

"Un limbaj simplu, orientat pe obiecte, care "înţelege" reţelele de calculatoare, interpretat, robust, sigur, neutru faţă de arhitecturi, portabil, de înaltă performanţă, multi-fir, dinamic".

Mediul Java

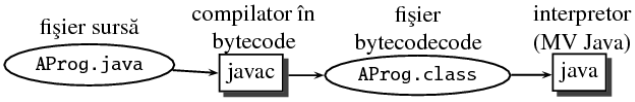


Executarea programelor Java

- Java foloseşte un proces în doi paşi
- Compilează programul în *bytecodes*
  - bytecode este apropiat de formatul instrucţiunilor în limbaj maşină, dar nu chiar la fel — este un "limbaj maşină" a generic
  - nu corespunde nici unui procesor real
- *Maşina virtuală* (VM) interpretează bytecode în limbaj maşină nativ şi-l rulează
  - există maşini virtuale diferite pentru calculatoare diferite, deoarece bytecode nu corespunde unei maşini reale

Executarea programelor Java

- Se utilizează *acelaşi* bytecode Java pe *calculatoare diferite* fără a recompila codul sursă
  - fiecare VM interpretează acelaşi bytecode Java
  - aceasta permite să se execute programe Java prin simpla obţinere 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



Aplicaţii Java

- Tipuri de aplicaţii Java:
  - *aplicaţii de-sine-stătătoare* şi *applets/servlets*
- Un *aplicaţie de-sine-stătoare* sau un program "obişnuit" este o clasă care are o metodă numită `main`
  - Când se lansează programul Java respectiv, *sistemul de execuţie* invocă automat metoda numită `main`
  - Toate *applicaţiile de-sine-stătătoare* încep din metoda `main`
- Deja aţi văzut asemenea aplicaţii în exemplele BlueJ

Applets

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

Încărcătorul de clase

- Programele Java sunt divizate în unităţi mai mici numite *clase*
  - Fiecare definiţie de clasă este în mod normal într-un fişier separat şi 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)*

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 definiţie de clasă trebuie să se afle într-un fişier al cărui nume este numele clasei urmat de extensia `.java`
  - Exemplu: Clasa `UnProgram` trebuie să se afle în fişierul numit `UnProgram.java`
- Fiecare clasă este compilată folosind comanda `javac` urmată de numele fişierului care conţine clasa

```
javac UnProgram.java
```

  - Rezultatul este un program în byte-code cu acelaşi 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
  - Rulaţi doar clasa care conţine o metodă **`main`** (sistemul va încărca şi 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
```

- Vedeţi **`Hello.java`** din exemplele BlueJ

Convenţii pentru nume

- Începeţi numele de variabile, metode şi obiecte cu o literă mică, indicaţi limitele "cuvintelor" cu o litera mare şi pentru celelalte caractere folosiţi doar litere şi cifre ("camelcase")

```
vitezaMaxima   rataDobinzii   oraSosirii
```

- Începeţi numele de clase cu majusculă şi pentru restul identificatorului aplicaţi regula de mai sus

```
UnProgram   OClasa   String
```


Declararea variabilelor

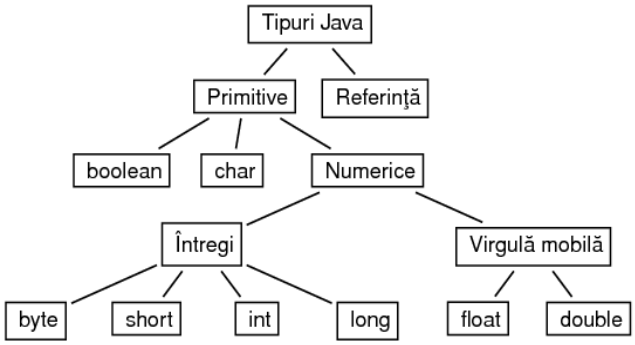
- Fiecare variabilă dintr-un program Java trebuie *declarată* înainte de utilizare
  - Declaraţia informează compilatorul asupra tipului de dată care va fi stocat în variabilă
  - Tipul variabilei este urmat de unul sau mai multe nume separate de virgule şi terminat cu punct şi virgulă
  - Variabilele 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, subclasele şi celelalte clase din acelaşi 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 ceilalţi membrii din acelaşi pachet.
  - **private** pentru oricine din afara pachetului
  - numită şi *acces în pachet.*
  - *"Vizibilă în vecinătate"*

Tipuri Java



Tipuri primitive

Tip primitiv	Biţi	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`)
- Observaţi 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 şi expresii

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

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

Reguli de precedenţă şi asociativitate

- La determinarea ordinii operaţiilor adiacente, operaţia cu precedenţă mai mare (şi argumentele sale aparente) este grupată înaintea operaţiei de precedenţă mai mică

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

Reguli de precedenţă şi asociativitate

- Operatorii *unari* de precedenţa egală sunt grupaţi de la dreapta la stânga

  `+-+rate` se evaluează ca `+(-(+rate))`
- Operatorii *binari* de precedenţă egală sunt grupaţi de la stânga la dreapta

  `base + rate + hours` se evaluează ca
  `(base + rate) + hours`
- Excepţie: Un şir 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 mobilă1.0/3.0 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, puţin mai puţin decât o treime
  - De fapt numerele sunt stocate binar, dar consecinţele sunt aceleaşi: numerele în virgulă mobilă pot pierde precizie

Împărţirea întreagă şi cea în virgulă mobilă

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

Conversia de tip explicită

- O *conversie de tip explicită (type cast)* ia o valoare de un tip şi produce o valoare "echivalentă" de celălalt tip
  - Dacă `n` şi `m` sunt întregii de împărţit şi e nevoie de partea fracţionară, atunci cel puţin un operand trebuie să fie în virgulă mobilă **înainte** de efectuarea operaţiei
    ```
    double ans = n / (double)m;
    ```
  - La fel ca în C, tipul dorit este pus între paranteze imediat înaintea variabilei de convertit
  - Tipul şi 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 *coerciţie 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 şi decrement

- Când oricare dintre operatorii ++ sau −− precede o variabilă şi 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 şi este parte a expresiei, expresia este evaluată folosind valoarea originală şi 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`, şi `char` sunt (în ordine) `Byte`, `Short`, `Long`, `Float`, `Double`, şi `Character`
- Aceste clase conţin şi un număr de constante predefinite utile, precum şi 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, creaţi un obiect de tipul corespunzător folosind valoarea primitivă ca argument
  - Noul obiect va conţine o variabilă instanţă 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 convertire a unui obiect din clasele **`Byte`**, **`Short`**, **`Integer`**, **`Long`**, **`Float`**, **`Double`**, şi **`Character`** la valorile primitive corespunzătore sunt (în ordine) **`byteValue`**, **`shortValue`**, **`intValue`**, **`longValue`**, **`floatValue`**, **`doubleValue`**, şi **`charValue`**
  - Nici una dintre aceste metode nu necesită argumente

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


Boxing şi 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 şi metode statice în clasele învelitoare

- Constante pentru valori maxime şi 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`** şi **`Boolean.FALSE`** sunt obiectele booleene corespunzătoare valorilor **`true`** şi **`false`** ale tipului primitiv **`boolean`**

Constante şi metode statice în clasele învelitoare

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

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

    returnează valoarea şir **`"123.99"`**
- Clasa **`Character`** conţine un număr de metode utile la prelucrarea şirurilor

Referinţe 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 stiva, variabila care poate stoca o referinţă la adresa obiectului.
- Despre variabilele care stochează referinţe la adrese de obiecte se spune că păstrează tipul de date *referinţă*
- Exemplu:

```
PlatitorTaxe t = new
  PlatitorTaxe(1111111120633, 30000);
```


Clasa `String`

- Nu există un tip primitiv pentru şiruri în Java
- Clasa **`String`** este o clasă Java predefinită folosită la stocarea şi prelucrarea şirurilor
- Obiectele de tipul **`String`** sunt compuse din şiruri de caractere scrise între ghilimele
  - Orice şir între ghilimele este de clasa **`String`**

```
"Fiti fericiti cu Java."
```

- Unei variabile de tipul **`String`** i se poate da valoarea unui obiect **`String`**

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

Concatenarea şirurilor

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

Metode al clasei string String

- Clasa `String` conţine multe metode utile la aplicaţiile care prelucrează şiruri
  - O metodă a clasei `String` se apelează prin numele unui obiect `String`, un punct, numele metodei şi 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());
```

  - *Poziţia* sau *indexul* unui caracter într-un şir se numără întotdeauna de la zero

Metode ale clasei `String`

Lungimea şirului

`i =`	*s*`.length()`	lungimea şirului *s*.

Comparaţii (note: folosiţi aceste metode în loc de == şi !=)

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

Metode ale clasei `String`

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

`i =`	*s*`.indexOf(`*t*`)`	indexul primei apariţii lui String *t* in *s*.
`i =`	*s*`.indexOf(`*t*`, i)`	indexul lui String *t* la sau după poziţia *i* în *s*.
`i =`	*s*`.indexOf(`*c*`)`	indexul primei apariţii caracterului *c* în *s*.
`i =`	*s*`.indexOf(`*c*`, i)`	indexul caracterului *c* la sau după poziţia *i* din *s*.
`i =`	*s*`.lastIndexOf(`*c*`)`	indexul ultimei apariţii lui *c* în *s*.
`i =`	*s*`.lastIndexOf(`*c*`, i)`	indexul ultimei apariţii lui *c* pe sau înainte de poziţia *i* în *s*.
`i =`	*s*`.lastIndexOf(`*t*`)`	indexul ultimei apariţii lui *t* în *s*.
`i =`	*s*`.lastIndexOf(`*t*`, i)`	indexul ultimei apariţii a lui *t* pe sau înainte de poziţia *i* în *s*.

Metode ale clasei `String`

Obţinerea de părţi

`c =`	*s*`.charAt(`*i*`)`	caracterul de la poziţia *i* din *s*.
`s1 =`	*s*`.substring(`*i*`)`	subşirul de la indexul *i* până la sfârşitul lui *s*.
`s1 =`	*s*`.substring(`*i*`, `*j*`)`	subşirul de la indexul *i* până ÎNAINTE de indexul *j* din *s*.

Crearea unui nou şir din original

`s1 =`	*s*`.toLowerCase()`	nou String cu toate caracterele minuscule
`s1 =`	*s*`.toUpperCase()`	nou String cu toate caracterele majuscule
`s1 =`	*s*`.trim()`	nou String fără spaţii albe la început şi sfârşit
`s1 =`	*s*`.replace(`*c1*`, `*c2*`)`	nou String cu toate *c2*-urile înlocuite prin *c1*.

Metode ale clasei `String`

Metode statice pentru conversia la String

`s =`	`String.`*s*`valueOf(`*x*`)`	Converteşte x la String, unde x este *orice* valoare de tip (primitiv sau obiect).
`s =`	`String.`*s*`format(`*f*`, x...)`	[Java >=5] Foloseşte formatul *f* pentru a converti un număr variabil de parametri, *x* la un şir.

- Lista nu este exhaustivă.

Prelucrarea şirurilor

- Un obiect **`String`** nu se poate schimba (eng. immutable), caracterele pe care le conţine nu pot fi schimbate
- Clasa **`StringBuffer`** are metode prin care se pot edita obiectele sale şir
- Se poate totuşi 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 conţine 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 alfabete nelatine
  - Exemple:

```
char c='\u0103'; // litera 'ă'
String s="\u00eencoto\u015fm\u0103ni\u0163i";// încotoşmăniţi
```


O porţiune din codul Unicode

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

Numirea constantelor

- În loc de numere "anonime", declaraţi constante simbolice (cu nume) şi folosiţi-le numele

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

  - Previne schimbarea nedorită a valorii
  - Uşurează modificarea valorii
  - Convenţia de nume pentru constante: toate literele majuscule, limitele de cuvinte marcate prin liniuţa de subliniere

Comentariile

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

Documentarea programului

- Java include programul **`javadoc`** care extrage automat documentaţia 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 şi modelul de indentare

Rezumat

- Abstracţiuni în programele OO
  - nivele: pachet, client/server, interfeţe, implementări, metode (considerate separat)
  - Are-o (agregare)
  - Este-o (specializare)
  - Compoziţie
  - TDA
- Java
  - mediul (compilator, încărcător de clase, interpretor, JIT, sistem de execuţie etc.)
  - convenţii de numire
  - declaraţii şi modificatori de acces
  - Tipuri primitive (boolean, char, short, int, long, float, double) şi clase învelitoare
  - Operatori aritmetici
  - Clasa String