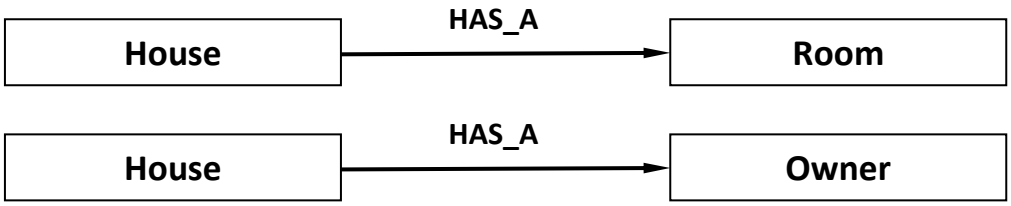
AGREGARE ȘI COMPOZIȚIE

- Agregarea și compoziția reprezintă alte două modalități de interconectare (asociere) a două clase, alături de mecanismul de extindere a claselor (moștenire).
- Practic, agregarea și compoziția reprezintă alte modalități de reutilizare a codului.
- Asocierea a două clase se realizează prin încapsularea în clasa container a unei referințe, de un tip diferit, către un obiect al clasei asociate (încapsulate).
- Conceptual, compoziția este diferită de agregare în raport de ciclul de viață al obiectului încapsulat, astfel:
  - dacă ciclul de viață al obiectului încapsulat este dependent de ciclul de viață al obiectului container, atunci relația de asociere este de tip *compoziție* (strong association);
  - dacă obiectul încapsulat poate să existe și după distrugerea containerului său, atunci relația de asociere este de tip *agregare* (weak association).

Exemple:



- Se poate observa cu ușurință faptul că relația de asociere dintre clasele `House` și `Room` este una de tip compoziție (dacă este distrusă întreaga casă, atunci, în mod automat, va fi distrusă și camera respectivă), iar relația de asociere dintre clasele `House` și `Owner` este una de tip agregare (chiar dacă este distrusă întreaga casă, proprietarul său poate să trăiască în continuare).
- Din punct de vedere al implementării, diferențierea dintre cele două tipuri de asocieri se realizează prin modul în care obiectul container încapsulează referința sper obiectul asociat. Astfel, în cazul unei relații de agregare este suficientă încapsularea în clasa container a unei referințe spre obiectul asociat, deoarece acesta poate exista și independent:

```
class Person{
   private String name;
   private String SSN;
   ……
}
```

```
class House{
   private String address;
   private Person owner;
   ……

   public House(Person owner,…){
      this.owner = owner;
      ……
   }
}
```

În cazul unei relații de compoziție se va încapsula în clasa container o referință a unei copii locale a obiectului asociat:

```
class Room{
   private float width;
   private float length;
   ……

   public Room(Room r){
      this.width = r.width;
      ……
   }
}
```

```
class House{
   private String address;
   private Room dining;
   ……

   public House(Room dining,…){
      this.dining = new Room(dining);
      ……
   }
}
```

- În exemplul de mai sus, am presupus faptul că în clasa `Room` este definit un constructor care să inițializeze obiectul curent de tip `Room` cu valorile altui obiect de acest tip ("constructor de copiere"). Evident, dacă acest constructor nu există, se va utiliza unul dintre constructorii existenți, eventual împreună cu metode de tip `set/get`.

- În concluzie, compoziția și agregarea sunt relații de tip HAS_A, care se folosesc în momentul în care dorim să reutilizăm o clasă existentă, dar nu există o relație de tipul IS_A între ea și noua clasă, deci nu putem să utilizăm moștenirea.

- Cu alte cuvinte, dacă noua clasă este asemănătoare, din punct de vedere al modelarii, cu o clasă definită anterior, atunci se va utiliza extinderea, realizându-se o specializare a sa prin redefinirea unor metode. Dacă noua clasă nu este asemănătoare cu o clasă deja definită, dar are nevoie de metodele sale (fără a le modifica!), atunci se va utiliza compoziția sau agregarea.

- Agregarea se va utiliza în cazul în care obiectul container nu poate controla complet obiectul asociat (extern), acesta fiind creat/modificat de alte obiecte, iar compoziția se va utiliza când obiectul container trebuie să aibă control complet asupra obiectului asociat (dacă nu furnizăm metode de acces pentru obiectul asociat, atunci nimeni din exterior nu-l poate modifica).

ȘIRURI DE CARACTERE

- În limbajul Java sunt predefinite 3 clase pentru manipularea la nivel înalt a șirurilor de caractere:
  1. clasa `String`
  2. clasa `StringBuilder`
  3. clasa `StringBuffer`

- De asemenea, șirurile de caractere poate fi implementate și manipulate direct (fără a utiliza metode predefinite din cele 3 clase menționate mai sus), prin intermediul tablourilor cu elemente de tip `char`. Deși această abordare are dezavantajul unor implementări mai complicate, acest lucru este compensat de o viteză de executare mai mare și o utilizare mai eficientă a memoriei!

CLASA `String`

- Folosind clasa String, un șir de caractere poate fi instanțiat în două moduri:
  1. `String s = "exemplu";`
  2. `String s = new String("exemplu");`
- Diferența dintre cele două metode constă în zona de memorie în care va fi alocat șirul respectiv:
  1. Se va utiliza o zona de memorie specială, numită *tabelă de șiruri* (string literal/constant pool). Practic, în această zonă se păstrează toate șirurile deja create, iar în momentul în care se va inițializa un nou șir de caractere se va verifica dacă acesta există deja în tabelă. În caz afirmativ, nu se va mai aloca un nou șir în tabelă, ci se va utiliza referința șirului deja existent, ceea ce va conduce la o optimizare a utilizării memoriei (vor exista mai multe referințe spre un singur șir). În momentul în care spre un șir din tabelă nu va mai exista nicio referința activă, șirul va fi eliminat din tabelă.
  2. Se va utiliza zona de memorie heap.

- **Exemplu:**

```
String sir_1 = "exemplu";
String sir_2 = "exemplu";
String sir_3 = new String("exemplu");
String sir_4 = new String("exemplu");
System.out.println(sir_1 == sir_2);     // se va afișa true
System.out.println(sir_3 == sir_4);     // se va afișa false
System.out.println(sir_1 == sir_3);     // se va afișa false
```

- Un avantaj foarte important al utilizării tabelei de șiruri îl constituie faptul că operația de comparare a două șiruri din punct de vedere al conținuturilor lor se poate realiza direct, prin compararea referințelor celor două șiruri, utilizând operatorul ==. Evident, această variantă este mai rapidă decât utilizarea metodei `boolean equals(String șir)`, care verifică egalitatea celor două șiruri caracter cu caracter.
- Un șir de caractere alocat dinamic, folosind operatorul `new`, poate fi plasat în tabela de șiruri folosind metoda `String intern()`:

```
String sir_1 = "exemplu";
String sir_2 = new String("exemplu");
System.out.println(sir_1 == sir_2);       // se va afișa false
sir_2 = sir_2.intern();
System.out.println(sir_1 == sir_2);       // se va afișa true
```

- Odată creat un șir de caractere, conținutul său nu mai poate fi modificat. Orice operație de modificare a conținutului său va conduce la construcția unui alt șir! Astfel, după executarea secvenței de cod:

```
String sir_1 = "programare";
sir_1.toUpperCase();
System.out.println(sir_1);
```

se va afișa `programare`! Practic, prin instrucțiunea `sir_1.toUpperCase()` se va crea un nou șir având conținutul `PROGRAMARE`, deci fără a modifica șirul `sir_1`! Astfel, vor exista două șiruri, unul având conținutul "`programare`" și referința păstrată în `sir_1`, respectiv unul având conținutul "`PROGRAMARE`" a cărui referință nu este stocată în nicio variabilă! Evident, chiar dacă șirul nu poate fi modificat din punct de vedere al conținutului, se poate modifica conținutul unei variabile care conține referința sa: `sir_1 = sir_1.toUpperCase()`. Astfel, șirul de caractere `sir_1` va conține acum referința șirului "`PROGRAMARE`"!

- În general, dacă instanțele unei clase nu mai pot fi modificate din punct de vedere al conținutului după ce au fost create, spunem că respectiva clasă este o *clasă imutabilă*.

- Clasa `String` pune la dispoziția programatorilor metode pentru:
  1. determinarea numărului de caractere:
     - `int length()`
  2. extragerea unui subșir:
     - `String substring(int beginIndex)`
     - `String substring(int beginIndex, int endIndex)`
  3. extragerea unui caracter:
     - `char charAt(int index)`

4. compararea lexicografică a două șiruri:
   - `int compareTo(String anotherString)`
   - `int compareToIgnoreCase(String anotherString)`
   - `boolean equals(Object anotherObject)`
   - `boolean equalsIgnoreCase(String anotherString)`
5. transformarea tuturor literelor în litere mici sau în litere mari:
   - `String toLowerCase()`
   - `String toUpperCase()`
6. eliminarea spațiilor de la începutul și sfârșitul șirului:
   - `String trim()`
7. căutarea unui caracter sau a unui subșir:
   - `int indexOf(int ch)`
   - `int indexOf(int ch, int fromIndex)`
   - `int indexOf(String str)`
   - `int indexOf(String str, int fromIndex)`
   - `int lastIndexOf(int ch)`
   - `int lastIndexOf(int ch, int fromIndex)`
   - `int lastIndexOf(String str)`
   - `int lastIndexOf(String str, int fromIndex)`
   - `boolean startsWith(String prefix)`
   - `boolean startsWith(String prefix, int toffset)`
   - `boolean endsWith(String suffix)`
8. reprezentarea unei valori de tip primitiv sau a unui obiect sub forma unui șir de caractere:
   - `static String valueOf(boolean b)`
   - `static String valueOf(char c)`
   - `static String valueOf(double d)`
   - `static String valueOf(float f)`
   - `static String valueOf(int i)`
   - `static String valueOf(long l)`
   - `static String valueOf(Object obj)`

- Informații detaliate despre toate metodele din clasa `String` pot fi găsite în pagina: https://docs.oracle.com/javase/7/docs/api/java/lang/String.html.

- În afara metodelor menționate anterior, în clasa `String` există mai multe metode care necesită utilizarea unor *expresii regulate* (regex).

- O *expresie regulată* (regex) este o secvență de caractere prin care se definește un șablon de căutare. De obicei, expresiile regulate se utilizează pentru a testa validitatea datelor de intrare (de exemplu, pentru a verifica dacă un șir conține un CNP formal corect) sau pentru realizarea unor operații de căutare/înlocuire/parsare într-un șir de caractere.

- Câteva reguli uzuale pentru definirea unei expresii regulate sunt următoarele:
  - `[abc]` – șirul este format doar dintr-una dintre literele a sau b sau c
  - `[^abc]` – șirul este format din orice caracter, mai puțin literele a, b și c
  - `[a-z]` – șirul este format dintr-o singură literă mică
  - `[a-zA-Z]` – șirul este format dintr-o singură literă mică sau mare
  - `[a-z][A-Z]` – șirul este format dintr-o literă mică urmată de o literă mare
  - `[abc]+` – șirul este format din orice combinație a literelor `a`, `b` și `c`, iar lungimea sa este cel puțin 1
  - `[abc]*` – șirul este format din orice combinație a literelor `a`, `b` și `c`, iar lungimea sa poate fi chiar 0
  - `[abc]{5}` – șirul este format din orice combinație a literelor `a`, `b` și `c` de lungime exact 5
  - `[abc]{5,}` – șirul este format din orice combinație a literelor `a`, `b` și `c` de lungime cel puțin 5
  - `[abc]{5,10}` – șirul este format din orice combinație a literelor `a`, `b` și `c` cu lungimea cuprinsă între 5 și 10

- Câteva exemple de utilizare a metodelor care necesită expresii regulate:
  1. pentru a verifica dacă un șir de caractere are o anumită formă particulară se folosește metoda `boolean matches(String regex)`:
     a) șirul `s` începe cu o literă mare, apoi conține doar litere mici (cel puțin una!):
     ```
     boolean ok = s.matches("[A-Z][a-z]+");
     ```
     b) șirul `s` conține doar cifre:
     ```
     boolean ok = s.matches("[0-9]+");
     ```
     c) șirul `s` conține un număr de telefon Vodafone:
     ```
     boolean ok = s.matches("(072|073)[0-9]{7}");
     ```

  2. pentru a înlocui în șirul `s` un subșir de o anumită formă cu un alt șir, folosind metodele `String replaceAll(String regex, String replacement)`, respectiv `String replaceFirst(String regex, String replacement)`:
     a) înlocuim spațiile consecutive cu un singur spațiu:
     ```
     s = s.replaceAll("[ ]{2,}", " ");
     ```
     b) înlocuim cuvântul "`are`" cu "`avea`":
     ```
     s = s.replaceAll("\\bare\\b", "avea");
     ```
     c) înlocuim fiecare vocală cu *:
     ```
     s = s.replaceAll("[aeiouAEIOU]", "*");
     ```
     d) înlocuim prima vocală cu *:
     ```
     s = s.replaceFirst("[aeiouAEIOU]", "*");
     ```
     e) înlocuim grupurile formate din cel puțin două vocale cu *:
     ```
     s = s.replaceAll("[aeiouAEIOU]{2,}", "*");
     ```

3. pentru a împărți un șir `s` în subșiruri (stocate într-un tablou de șiruri), în raport de anumiți delimitatori, folosind metoda `String[] split(String regex)`:
   a) împărțirea textului în caractere:
   ```
   String[] w = s.split("");
   ```
   b) împărțirea textului în cuvinte de lungime nenulă:
   ```
   String[] w = s.split("[ .,:;!?]+");
   ```
   c) extragerea numerelor naturale :
   ```
   String[] w = s.split("[^0-9]+");
   ```

CLASA `StringBuilder`

- Un dezavantaj major al obiectelor imutabile de tip `String` este dat de faptul că orice modificare a unui șir de caractere necesită construcția unui nou șir sau chiar a mai multora. De exemplu, pentru a înlocui al patrulea caracter dintr-un șir `s` cu *, se vor construi în tabela de șiruri alte 4 șiruri de caractere:

```
String s = "exemplu";
String t = s.substring(0, 3) + "*" + s.substring(4);
```

  Practic, în exemplu de mai sus se vor crea în tabela de șiruri, dacă nu există deja, șirurile "exe", "exe*", "plu" și "exe*plu"!
- Așadar, sunt situații în care se preferă utilizarea unui șir de caractere care să poate fi modificat direct, de exemplu, când se construiește dinamic un șir prin concatenarea mai multor șiruri.
- Obiectele de tip `StringBuilder` sunt asemănătoare cu cele de tip `String`, însă nu mai sunt imutabile, deci pot fi direct modificate.
- Intern, obiectele de tip `StringBuilder` sunt alocate în zona de memorie heap și sunt tratate ca niște tablouri de caractere. Dimensiunea tabloului se modifică dinamic, pe măsură ce șirul este construit (inițial, șirul are o lungime de 16 caractere):

```
StringBuilder sb = new StringBuilder();
sb.append("exemplu");
```

- Deoarece nu sunt imutabile, șirurile de tip `StringBuilder` nu sunt *thread-safe*, respectiv două sau mai multe fire de executare pot modifica simultan același șir, efectele fiind imprevizibile!

- Clasa `StringBuilder` conține, în afara unor metode asemănătoare celor din clasa `String` (de exemplu, metodele `indexOf`, `lastIndexOf` și `substring`), mai multe metode specifice:
  - modificarea lungimii șirului prin trunchiere sau extindere cu caracterul `'\u0000'`:
    - `void setLength(int newLength)`
  - adăugarea la sfârșitul șirului a unor caractere obținute prin conversia unor valori de tip primitiv sau obiecte:
    - `StringBuilder append(boolean b)`
    - `StringBuilder append(char c)`
    - `StringBuilder append(double d)`
    - `StringBuilder append(float f)`
    - `StringBuilder append(int i)`
    - `StringBuilder append(long lng)`
    - `StringBuilder append(Object obj)`
    - `StringBuilder append(String str)`
    - `StringBuilder append(StringBuffer sb)`
  - inserarea în șir, începând cu poziția `offset`, a unor caractere obținute prin conversia unor valori de tip primitiv sau obiecte:
    - `StringBuilder insert(int offset, boolean b)`
    - `StringBuilder insert(int offset, char c)`
    - `StringBuilder insert(int offset, double d)`
    - `StringBuilder insert(int offset, float f)`
    - `StringBuilder insert(int offset, int i)`
    - `StringBuilder insert(int offset, long l)`
    - `StringBuilder insert(int offset, Object obj)`
    - `StringBuilder insert(int offset, String str)`
  - ștergerea unor caractere din șir:
    - `StringBuilder delete(int start, int end)`
    - `StringBuilder deleteCharAt(int index)`
  - înlocuirea unor caractere din șir:
    - `StringBuilder replace(int start, int end, String str)`
    - `void setCharAt(int index, char ch)`

CLASA `StringBuffer`

- Singura diferență dintre clasa `StringBuilder` și clasa `StringBuffer` constă în faptul că aceasta este thread-safe, adică metodele sale sunt sincronizate, fiind executate pe rând, sub excludere reciprocă! Din acest motiv, metodele sale sunt mai lente decât cele echivalente din clasa `StringBuilder`.

CLASE IMUTABILE

- Așa cum deja am menționat anterior, o clasă este imutabilă dacă nu mai putem modifica conținutul unei instanțe a sa (un obiect) după creare. Astfel, orice modificare a obiectului respectiv presupune crearea unui nou obiect și înlocuirea referinței sale cu referința noului obiect creat.

- În limbajul Java există mai multe clase imutabile predefinite: `String`, clasele înfășurătoare (`Integer`, `Float`, `Boolean` etc.), `BigInteger` etc.

- Principalele avantaje ale utilizării claselor imutabile sunt următoarele:
  - sunt implicit thread-safe (nu necesită sincronizare într-un mediu concurent);
  - sunt ușor de proiectat, implementat, utilizat și testat;
  - sunt mai rapide decât clasele mutabile;
  - obiectele pot fi reutilizate folosind o tabelă de referințe și o metodă de tip factory pentru instanțierea lor;
  - pot fi utilizate pe post de chei în structuri de date asociative (de exemplu, tabele de dispersie - `HashMap`);
  - programele care utilizează doar clase mutabile pot fi ușor adaptate pentru utilizarea într-un mediu distribuit.

- Singurul dezavantaj important al claselor imutabile îl constituie faptul că sunt create mai multe obiecte intermediare, respectiv câte unul pentru fiecare operație efectuată.

- De obicei, crearea unei clase imutabile trebuie să respecte următoarele reguli:
  1. clasa nu va permite rescrierea metodelor sale, fie declarând clasa de tip `final`, fie declarând constructorii ca fiind `private` și folosind metode de tip `factory` pentru a crea obiecte;
  2. toate câmpurile vor fi declarate ca fiind `final` (li se vor atribui valori o singură dată, printr-un constructor cu parametri) și `private` (nu li se pot modifica valorile direct);
  3. clasa nu va conține metode de tip `set` sau alte metode care pot modifica valorile câmpurilor;
  4. dacă există câmpuri care sunt referințe spre obiecte mutabile, se va împiedica modificarea acestora, astfel:
     a. nu se vor folosi referințe spre obiecte externe, ci spre copii ale lor (se va folosi compoziția, ci nu agregarea!)

Exemplu: Fie o clasă `Student` care conține data membră `facultate` de tipul unei clase mutabile `Facultate`:

Greșit:
```
public Student(Facultate f, …){
  this.facultate = f;              //agregare, deci obiectul
  ……                               //extern poate fi modificat!
}
```

Corect:
```
public Persoana(Date dn, …){
  this.facultate = new Facultate(f);    //compoziție, deci obiectul
  ……                                    //extern nu poate fi modificat
}
```

b. nu se vor returna referințe spre câmpurile mutabile, ci se vor returna referințe spre copii ale lor:

Greșit:
```
public Facultate getFacultate(){
  return this.facultate;
}
```

Corect:
```
public Facultate getFacultate(){
  return new Facultate(facultate);
}
```

- În continuare, vom prezenta complet clasele `Facultate` și `Student` menționate anterior:

Clasa mutabilă `Facultate`:

```
public class Facultate {
    private String denumire;
    private String adresa;
    private String email;
    private String telefon;

    public Facultate(String denumire, String adresa, String email,
                                                      String telefon) {
        this.denumire = denumire;
        this.adresa = adresa;
        this.email = email;
        this.telefon = telefon;
    }
```

```
    public Facultate(Facultate facultate) {
        this.denumire = facultate.denumire;
        this.adresa = facultate.adresa;
        this.email = facultate.email;
        this.telefon = facultate.telefon;
    }

    public String getDenumire() { return denumire; }

    public void setDenumire(String denumire) { this.denumire = denumire; }

    public String getAdresa() { return adresa; }

    public void setAdresa(String adresa) { this.adresa = adresa; }

    public String getEmail() { return email; }

    public void setEmail(String email) { this.email = email; }

    public String getTelefon() { return telefon; }

    public void setTelefon(String telefon) { this.telefon = telefon; }
}
```

Clasa imutabilă Student:

```
//regula 1
public final class Student
{
    //regula 2
    private final String nume;
    private final Facultate facultate;
    private final int grupa;
    private final double medie;

    public Student(String nume, Facultate facultate, int grupa, double medie) {
        this.nume = nume;
        //regula 4a
        this.facultate = new Facultate(facultate);
        this.grupa = grupa;
        this.medie = medie;
    }

    public String getNume() {
        return nume;
    }
```

```
    public Facultate getFacultate() {
        //regula 4b
        return new Facultate(facultate);
    }

    public int getGrupa() {
        return grupa;
    }

    public double getMedie() {
        return medie;
    }
}
```

Observați faptul că în clasa `Student` nu sunt definite metode de tip `set`, deci este respectată și regula 3!

CLASE DE TIP ÎNREGISTRARE (RECORDS)

- În Java 15 au fost introduse *clasele de tip înregistrare* sau, pe scurt, *înregistrări* (*records*).

- O *înregistrare* este o clasă imutabilă utilizată pentru a manipula o mulțime fixă de valori, denumite *componentele înregistrării*. De obicei, înregistrările sunt utilizate pentru încărcarea unor date dintr-o anumită sursă (de exemplu, un fișier sau o bază de date) și, eventual, transportarea acestora către o anumită destinație, folosind facilitățile limbajului Java pentru programarea în rețea.

- O înregistrare se declară într-un mod foarte concis, precizând doar tipul și numele componentelor sale în descriptorul înregistrării:

```
[modificatori de acces] record Denumire (descriptor) {}
```

Exemplu:

```
public record Student(String nume, int grupa, double medie) {}
```

- Orice înregistrare este în mod implicit o clasă de tip `final` care extinde clasa `java.lang.Record`, deci o înregistrare nu poate fi abstractă, nu poate fi extinsă și nici nu poate extinde alte clase sau alte înregistrări. Totuși, o înregistrare poate implementa una sau mai multe interfețe.

- Pentru o înregistrare, compilatorul va genera automat o clasă de tip `final` având următoarele componente:
  - câte o dată membră privată și finală pentru fiecare componentă;
  - un constructor canonic public care va avea câte un parametru pentru fiecare componentă a înregistrării și va utiliza valoarea parametrului respectiv pentru a inițializa componenta corespunzătoare;
  - câte o metodă de tip `get` pentru fiecare componentă, denumirea unei metode fiind identică cu denumirea componentei asociate;
  - implementarea metodei `equals(Object)` din clasa `Object`;
  - implementarea metodei `hashCode()` din clasa `Object`;
  - implementarea metodei `toString()`din clasa `Object`.

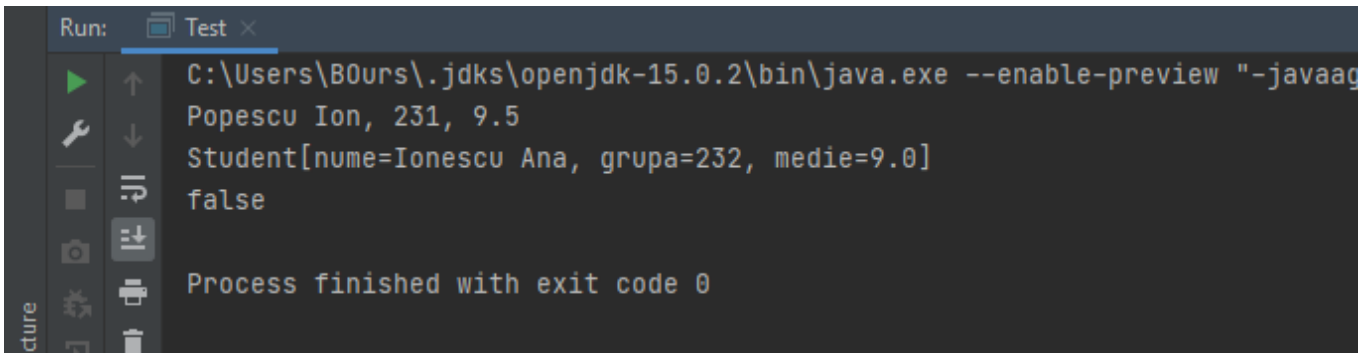
Metodele vor utiliza toate componentele înregistrării!

Exemplu:

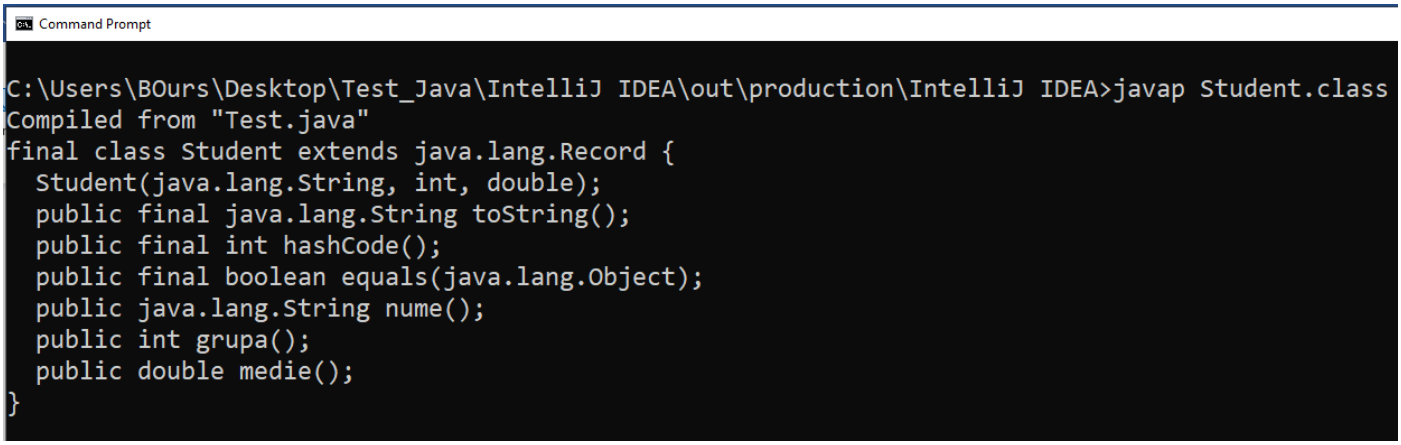
```
record Student(String nume, int grupa, double medie) {}

public class Test
{
    public static void main(String[] args)
    {
        Student stud_1 = new Student("Popescu Ion", 231, 9.50);
        Student stud_2 = new Student("Ionescu Ana", 232, 9.00);

        System.out.println(stud_1.nume() + ", " + stud_1.grupa() + ", "
                                                + stud_1.medie());
        System.out.println(stud_2); //se va utiliza metoda toString()!
        System.out.println(stud_1.equals(stud_2));
    }
}
```



Folosind programul utilitar `javap` putem afișa metodele generate automat de compilatorul Java pentru înregistrarea `Student` definită mai sus:



- În cadrul unei înregistrări se pot adăuga constructori supraîncărcați, dar aceștia trebuie să apeleze explicit constructorul canonic:

```
record Student(String nume, int grupa, double medie) {
    public Student(String nume, int grupa) {
        this(nume, grupa, 0);
    }
}
```

- Constructorul canonic poate fi redefinit, de obicei pentru a realiza prelucrări suplimentare sau validări ale componentelor înregistrării:

```
record Student(String nume, int grupa, double medie) {
    Student(String nume, int grupa, double medie) {
        if(medie < 0)
            medie = -medie;

        this.nume = nume.toUpperCase();
        this.grupa = grupa;
        this.medie = medie;
    }
}
```

- În cadrul unei înregistrări constructorul canonic poate fi accesat și prin intermediul unui *constructor compact*, adică un constructor public care nu are o listă a parametrilor, această fiind dedusă automat din descriptorul înregistrării:

```
record Student(String nume, int grupa, double medie) {
    Student {
        if(medie < 0)
            medie = -medie;
    }
}
```

  În cazul modificării constructorului compact, se vor preciza doar prelucrări suplimentare pe care dorim să le efectuăm înaintea inițializării componentelor înregistrării, deoarece instrucțiunile necesare inițializării componentelor vor fi adăugate automat de compilator!

- O înregistrare este o clasă imutabilă de tip *shallowly immutable*, respectiv datele membre de tip referință vor fi copiate superficial (*shallow copy*), ci nu în adâncime (*deep copy*), ceea ce poate afecta imutabilitatea înregistrării respective! Pentru a evita acest aspect, trebuie să redefinim constructorul canonic și metodele de tip `get` corespunzătoare componentelor mutabile conform regulii 4 prezentate în secțiunea dedicată claselor imutabile (în exemplul următor am considerat clasa mutabilă `Facultate` definită anterior):

```
record Student(String nume, Facultate facultate, int grupa, double medie) {
    Student(String nume, Facultate facultate, int grupa, double medie) {
        this.nume = nume;
        //regula 4a
        this.facultate = new Facultate(facultate);
        this.grupa = grupa;
        this.medie = medie;
    }

    @Override
    public Facultate facultate() {
        //regula 4b
        return new Facultate(facultate);
    }
}
```

- Într-o înregistrare nu se pot declara date membre de instanță, dar se pot declara metode nestatice (dar nu se recomandă acest lucru, deoarece rolul principal al înregistrărilor este acela de a stoca date, ci nu acela de a le prelucra). De asemenea, se pot adăuga date, metode și blocuri de inițializare statice:

```
record Student(String nume, int grupa, double medie) {
    private static String facultate = "Facultatea de Drept";

    public static String getFacultate() {
        return facultate;
    }

    public static void setFacultate(String facultate) {
        Student.facultate = facultate;
    }
}
```

Observați faptul că datele membre statice nu mai sunt implicit de tip final și nu se mai generează automat metode de tip `get` pentru ele!

- În concluzie, înregistrările reprezintă o modalitate simplă de implementare a unor clase imutabile care să permită stocarea și, eventual, transportarea unor date .