

## CLASE ABSTRACTE ȘI INTERFEȚE

### Obiective

Scopul acestui laborator este prezentarea conceptelor de **interfață** și de **clasă abstractă** și utilizarea lor în limbajul Java.

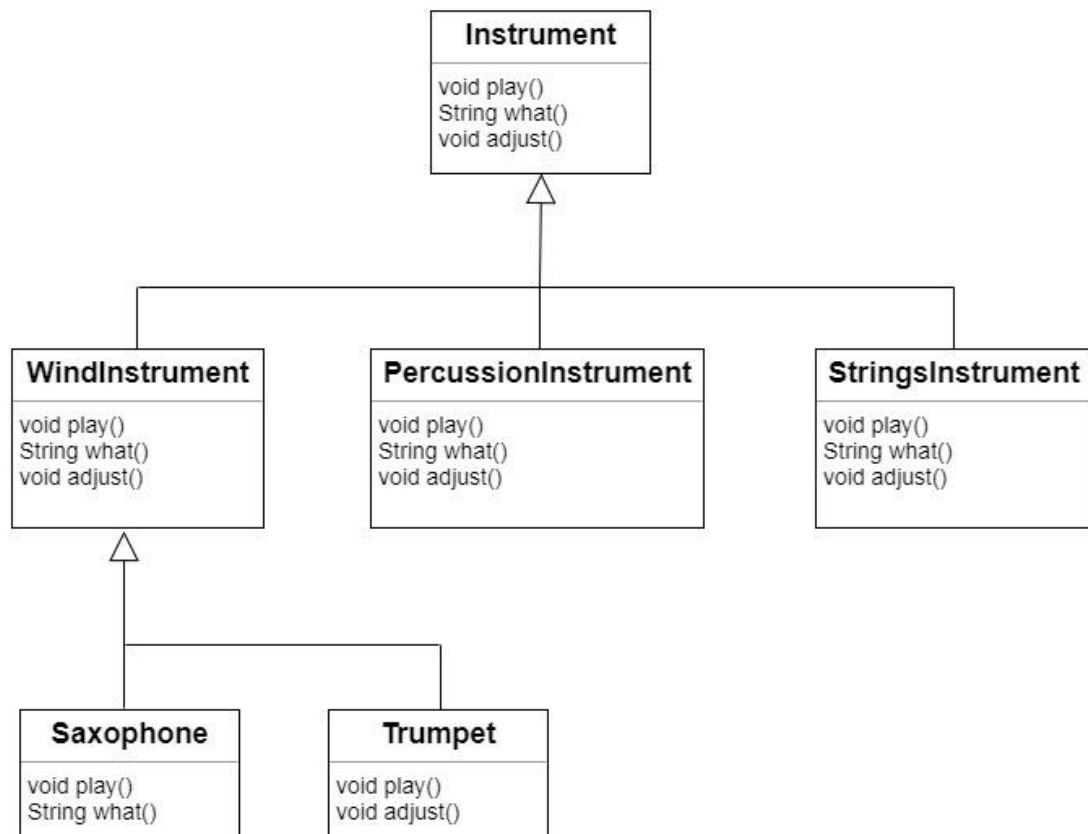
Conceptele de metode și clase abstracte și de interfețe sunt prezente și în alte limbaje OOP, fiecare cu particularitățile lor de sintaxă. E important ca în urma acestui laborator să înțelegeți ce reprezintă și situațiile în care să le folosiți.

Aspectele urmărite sunt:

- Prezentarea interfețelor și claselor abstracte
- Moștenirea în cazul interfețelor și claselor abstracte
- Diferențele dintre interfețe și clase abstracte

### Introducere

Fie următorul exemplu (Thinking in Java) care propune o ierarhie de clase pentru a descrie o suită de instrumente muzicale, cu scopul demonstrării polimorfismului:



Clasa **Instrument** nu este instanțiată niciodată pentru că scopul său este de a stabili o interfață comună pentru toate clasele derivate. În același sens, metodele clasei de bază nu vor fi apelate niciodată. Apelarea lor este ceva greșit din punct de vedere conceptual.

### Clase abstracte vs Interfețe

Folosim o **clasă abstractă** atunci când vrem:

- să implementăm doar unele din metodele din clasă
- ca respectiva clasă să nu poată fi instanțiată

Folosim o **interfață** atunci când vrem:

- să avem doar o descriere a structurii, fără implementări
- ca interfața în cauză să fie folosită împreună cu alte interfețe în contextul moștenirii

## Clase abstracte

Dorim să stabilim interfața comună pentru a putea crea funcționalitate diferită pentru fiecare subtip și pentru a ști ce anume au clasele derivate în comun. O clasă cu caracteristicile enumerate mai sus se numește **abstractă**. Creăm o clasă abstractă atunci când dorim să:

- manipulăm un set de clase printr-o **interfață comună**
- **reutilizăm** o serie de metode și membrii din această clasă în clasele derivate.

Metodele suprascrise în clasele derivate vor fi apelate folosind **dynamic binding** (late binding). Acesta este un mecanism prin care compilatorul, în momentul în care nu poate determina implementarea unei metode în avans, lasă la latitudinea JVM-ului (mașinii virtuale) alegerea implementării potrivite, în funcție de tipul real al obiectului. Această legare a implementării de numele metodei la **momentul execuției** stă la baza polimorfismului. **Nu există instanțe** ale unei clase abstracte, aceasta exprimând doar un punct de pornire pentru definirea unor instrumente reale. De aceea, crearea unui obiect al unei clase abstracte este o eroare, compilatorul Java semnalând acest fapt.

## Metode abstracte

Pentru a exprima faptul că o metodă este abstractă (adică se declară doar interfața ei, nu și implementarea), Java folosește cuvântul cheie **abstract**:

```
abstract void f();
```

O clasă care conține **metode abstracte** este numită **clasă abstractă**. Dacă o clasă are una sau mai multe metode abstracte atunci ea trebuie să conțină în definiție cuvântul **abstract**.

**Exemplu:**

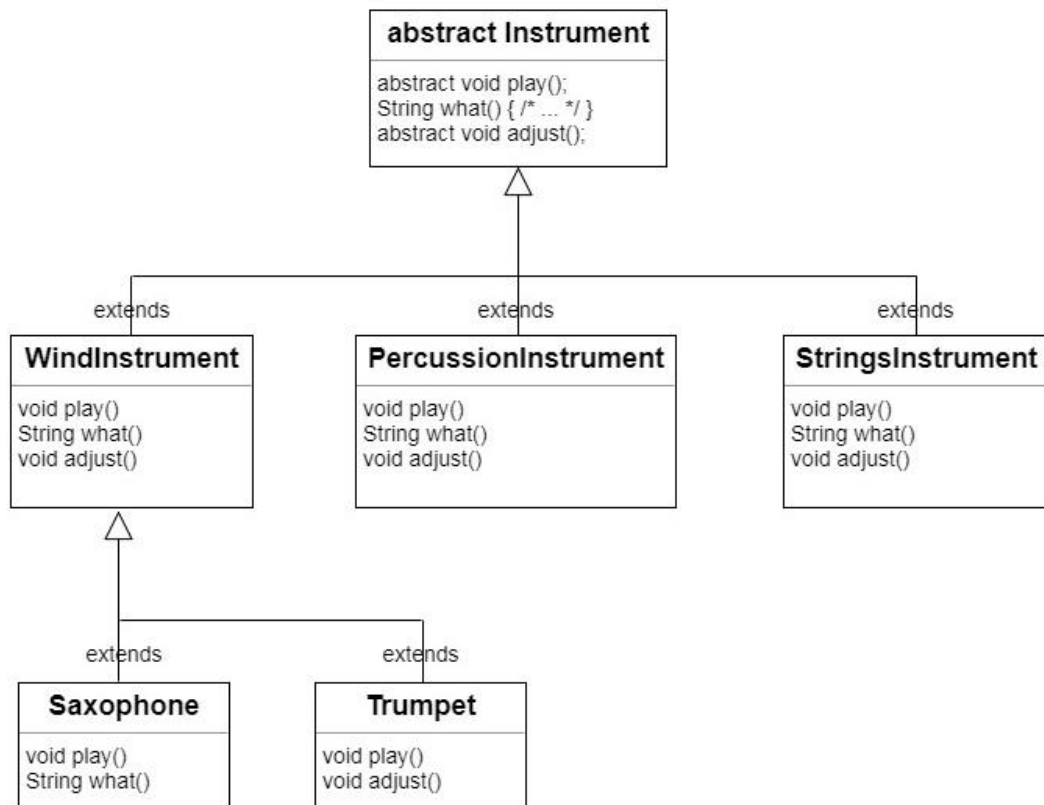
```
abstract class Instrument {  
...  
}
```

Deoarece o clasă abstractă este incompletă (există metode care nu sunt definite), crearea unui obiect de tipul clasei este împiedicată de compilator.

## Clase abstracte în contextul moștenirii

O clasă care moștenește o clasă abstractă este ea însăși abstractă dacă nu implementează **toate** metodele abstracte ale clasei de bază. Putem defini clase abstracte care moștenesc alte clase abstracte ș.a.m.d. O clasă care poate fi instanțiată (adică nu este abstractă) și care moștenește o clasă abstractă trebuie să implementeze (definiească) toate metodele abstracte pe lanțul moștenirii (ale tuturor claselor abstracte care îi sunt „părinți”). Este posibil să declarăm o **clasă abstractă fără** ca ea să aibă **metode abstracte**. Acest lucru este folositor când declarăm o clasă pentru care nu dorim instanțe (nu este corect conceptual să avem obiecte de tipul acelei clase, chiar dacă definiția ei este completă).

Iată cum putem să modificăm exemplul instrument cu metode abstracte:



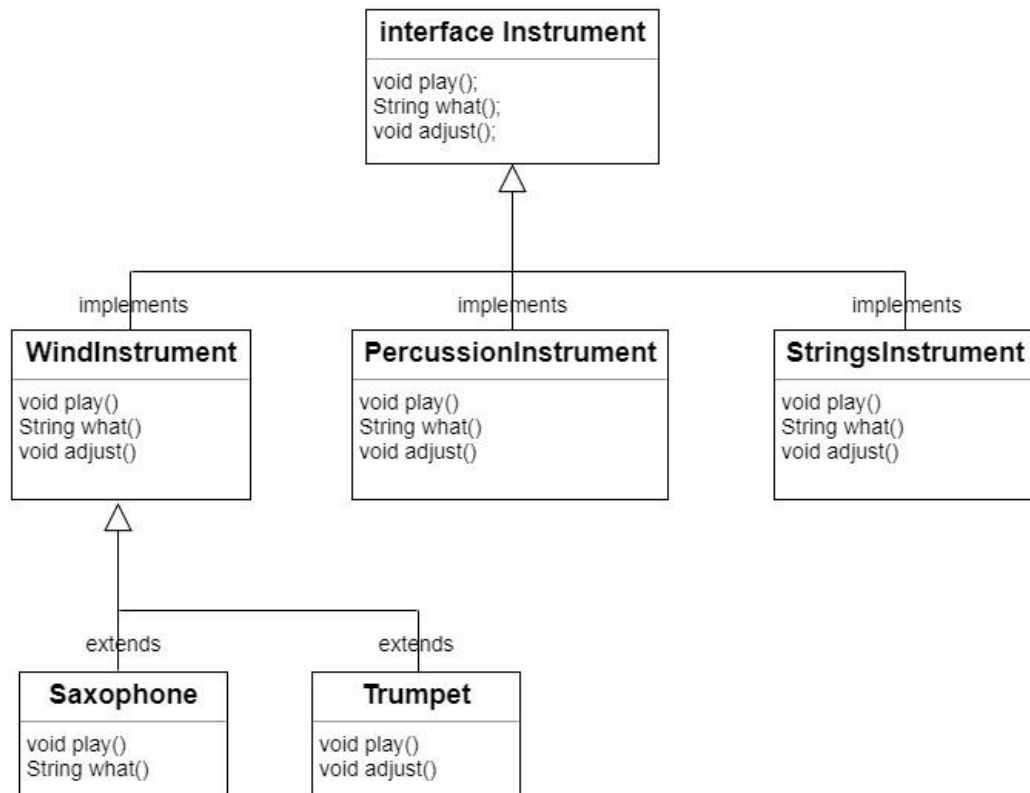
## Interfețe

Interfețele duc conceptul abstract un pas mai departe. Se poate considera că o interfață este o **clasă abstractă pură**: permite programatorului să stabilească o „formă” pentru o clasă (numele metodelor, lista de argumente, valori întoarse), dar fără **nici o implementare**. O interfață poate conține câmpuri, dar acestea sunt în mod implicit **static** și **final**. Metodele declarate în interfață sunt în mod implicit **public**.

Interfața este folosită pentru a descrie un **contract** între clase: o clasă care implementează o interfață va implementa metodele definite în interfață. Astfel orice cod care folosește o anumită interfață știe ce metode pot fi apelate pentru aceasta.

Pentru a crea o interfață folosim cuvântul cheie **interface** în loc de **class**. La fel ca în cazul claselor, interfața poate fi declarată public doar dacă este definită într-un fișier cu același nume ca cel pe care îl dăm acesteia. Dacă o interfață nu este declarată public atunci specificatorul ei de acces este `package-private`. Pentru a defini o **clasă** care este conformă cu o interfață anume folosim cuvântul cheie **implements**. Această relație este asemănătoare cu moștenirea, cu diferența că nu se moștenește comportament, ci doar „interfața”. Pentru a defini o **interfață** care moștenește altă interfață folosim cuvântul cheie **extends**. După ce o interfață a fost implementată, acea implementare devine o clasă obișnuită care poate fi extinsă prin moștenire.

Iată exemplul dat mai sus, modificat pentru a folosi interfețe:



Codul arată astfel:

```
interface Instrument {

    // Compile-time constant:
    int FIELD = 5; // static & final

    // Cannot have method definitions:
    void play(); // Automatically public

    String what();

    void adjust();
}

class WindInstrument implements Instrument {

    public void play() {
        System.out.println("WindInstrument.play()");
    }

    public String what() {
        return "WindInstrument";
    }

    public void adjust() {
    }
}

class Trumpet extends WindInstrument {
```

```

public void play() {
    System.out.println("Trumpet.play()");
}

public void adjust() {
    System.out.println("Trumpet.adjust()");
}
}

```

Un exemplu pentru o interfață care extinde mai multe interfețe:

```

interface A {
    void a1();
    void a2();
}

interface B {
    int x = 0;
    void b();
}

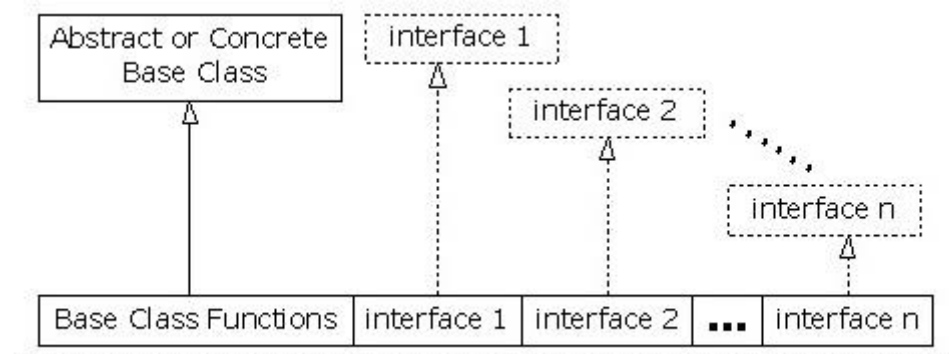
interface C extends A, B {
    // this interface will expose
    // * all the methods declared in A and B (a1, a2 and b)
    // * all the fields declared in A and B (x)
}

```

Implicit, specificatorul de acces pentru membrii unei interfețe este **public**. Atunci când implementăm o interfață trebuie să specificăm că funcțiile sunt **public** chiar dacă în interfață ele nu au fost specificate explicit astfel. Acest lucru este necesar deoarece specificatorul de acces implicit în clase este package-private, care este **mai restrictiv** decât **public**.

## Moștenire multiplă în Java

Interfața nu este doar o formă „pură” a unei clase abstracte, ci are un scop mult mai înalt. Deoarece o interfață nu specifică niciun fel de implementare (nu există nici un fel de spațiu de stocare pentru o interfață) este normal să „combinăm” mai multe interfețe. Acest lucru este folositor atunci când dorim să afirmăm că „X este un A, un B și un C”. Acest deziderat este moștenirea multiplă și, datorită faptului că o singură entitate (A, B sau C) are implementare, nu apar problemele moștenirii multiple din C++.



```

interface CanFight {

```

```
    void fight();
}

interface CanSwim {
    void swim();
}

interface CanFly {
    void fly();
}

class ActionCharacter {
    public void fight() {
    }
}

class Hero extends ActionCharacter implements CanFight, CanSwim, CanFly {

    public void swim() {
    }

    public void fly() {
    }
}

public class Adventure {

    static void t(CanFight x) {
        x.fight();
    }

    static void u(CanSwim x) {
        x.swim();
    }

    static void v(CanFly x) {
        x.fly();
    }

    static void w(ActionCharacter x) {
        x.fight();
    }

    public static void main(final String[] args) {
        Hero hero = new Hero();

        t(hero); // Treat it as a CanFight

        u(hero); // Treat it as a CanSwim

        v(hero); // Treat it as a CanFly
    }
}
```

```
w(hero); // Treat it as an ActionCharacter
    }
}
```

Se observă că *Hero* combină clasa *ActionCharacter* cu interfețele *CanSwim* etc. Acest lucru se realizează specificând prima dată clasa concretă (sau abstractă) (**extends**) și abia apoi **implements**. Metodele clasei *Adventure* au drept parametri interfețele *CanSwim* etc. și clasa *ActionCharacter*. La fiecare apel de metodă din *Adventure* se face **upcast** de la obiectul *Hero* la clasa sau interfața dorită de metoda respectivă.

## Coliziuni de nume la combinarea interfețelor

Combinarea unor interfețe care conțin o metodă cu același nume este posibilă doar dacă metodele nu au tipuri întoarse diferite și aceeași listă de argumente. Totuși este preferabil ca în interfețe diferite care trebuie combinate să nu existe metode cu același nume deoarece acest lucru poate duce la confuzii evidente (sunt amestecate în acest mod 3 concepte: *overloading*, *overriding* și *implementation*).

## Extinderea interfețelor

Se pot adăuga cu ușurință metode noi unei interfețe prin extinderea ei într-o altă interfață:

**Exemplu:**

```
interface Monster {
    void menace();
}

interface DangerousMonster extends Monster {
    void destroy();
}
```

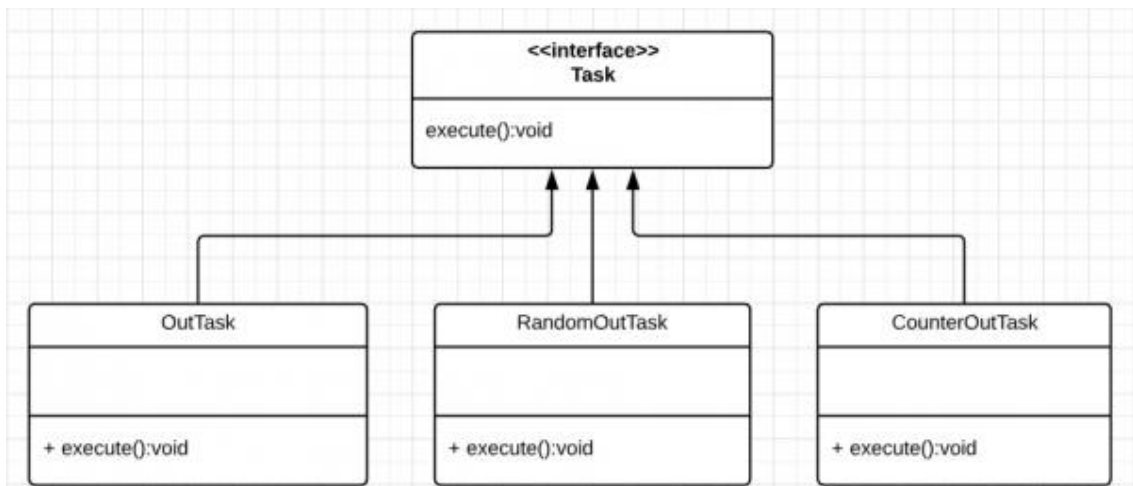
Deoarece câmpurile unei interfețe sunt automat **static** și **final**, interfețele sunt un mod convenabil de a crea grupuri de constante, asemănătoare cu **enum**-urile din C , C++.

## Inițializarea câmpurilor în interfețe

În interfețe **nu** pot exista **blank finals** (câmpuri final neinițializate) însă pot fi inițializate cu **valori neconstante**. Câmpurile fiind statice, ele vor fi inițializate prima oară când clasa este inițializată.

## Exerciții

1. (2p) Implementați interfața *Task* (din pachetul *first*) în cele 3 moduri de mai jos.
  - Un task care să afișeze un mesaj la output. Mesajul este specificat în constructor. (OutTask.java)
  - Un task care generează un număr aleator și afișează un mesaj cu numărul generat la output. Generarea se face în constructor. (RandomOutTask.java)
  - Un task care incrementează un contor global și afișează valoarea contorului după fiecare incrementare (CounterOutTask.java). **Notă:** Acesta este un exemplu simplu pentru [Command Pattern](#)



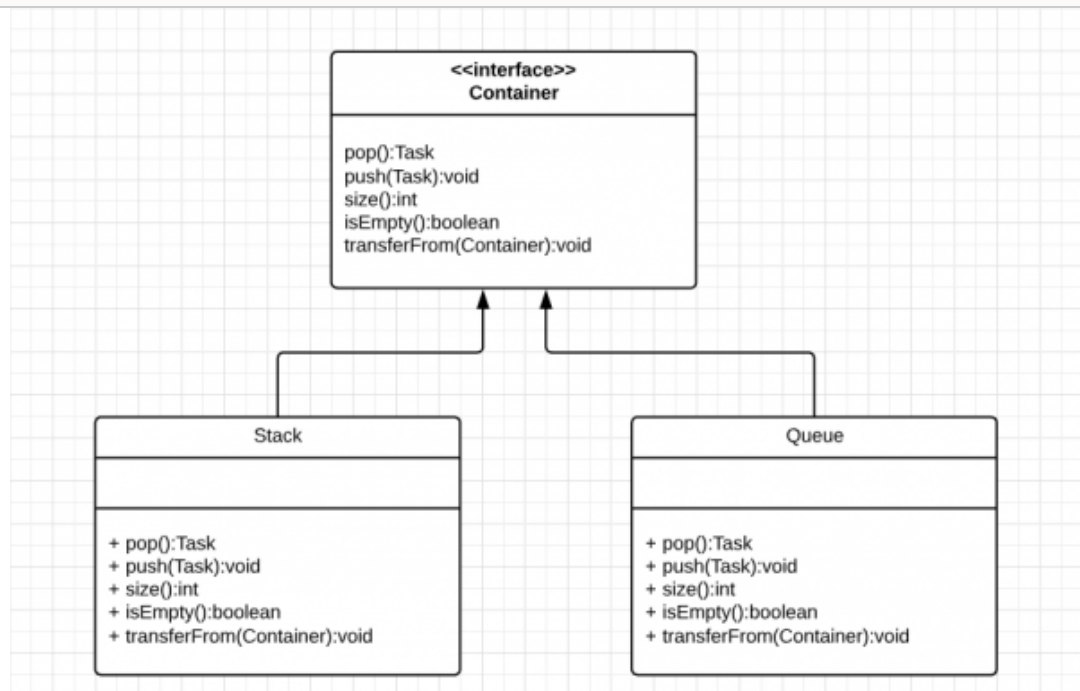
2. (3p) Interfața *Container* (din pachetul *second*) specifică interfața comună pentru colecții de obiecte *Task*, în care se pot adăuga și din care se pot elimina elemente. Creați două tipuri de containere care implementează această clasă:

- (1.5p) *Stack* - care implementează o strategie de tip **LIFO**
- (1.5p) *Queue* - care implementează o strategie de tip **FIFO**

**Evitați codul similar** în locuri diferite!

**Hint:** Puteți reține intern colecția de obiecte, utilizând clasa **ArrayList** din SDK-ul Java. Iată un exemplu de inițializare pentru șiruri:

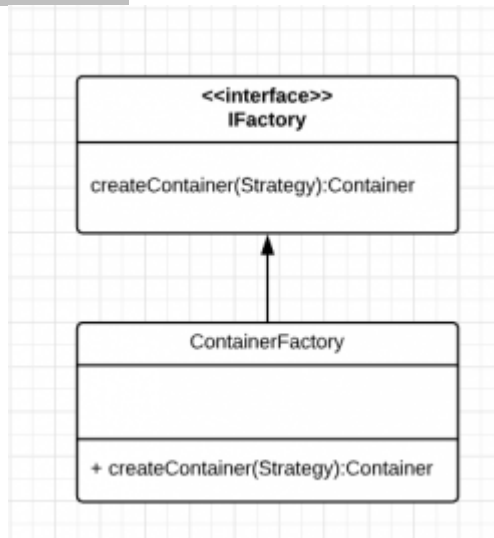
```
ArrayList<String> list = new ArrayList<String>();
```



3. (2p) Implementați interfața *IFactory* (clasa *ContainerFactory*, din pachetul *third*) care conține o metodă ce primește ca parametru o strategie (**enum** *Strategy*) și care întoarce un container asociat acelei strategii. Din acest punct înainte, în programul vostru veți crea containere folosind doar această clasă (nu puteți crea direct obiecte de tip *Stack* sau *Queue*). **Evitați** instanțierea clasei *Factory* implementate de voi la fiecare creare a unui container! **Notă:** Acest mod de a crea obiecte de tip *Container* elimină problema care apare în momentul în care decidem să folosim o implementare diferită pentru un anumit tip de strategie și nu vrem să facem modificări și în restul programului. De asemenea o astfel de abordare este utilă când avem implementări care urmăresc scopuri diferite (putem avea un *Factory* care să creeze containere optimizate pentru viteză sau un

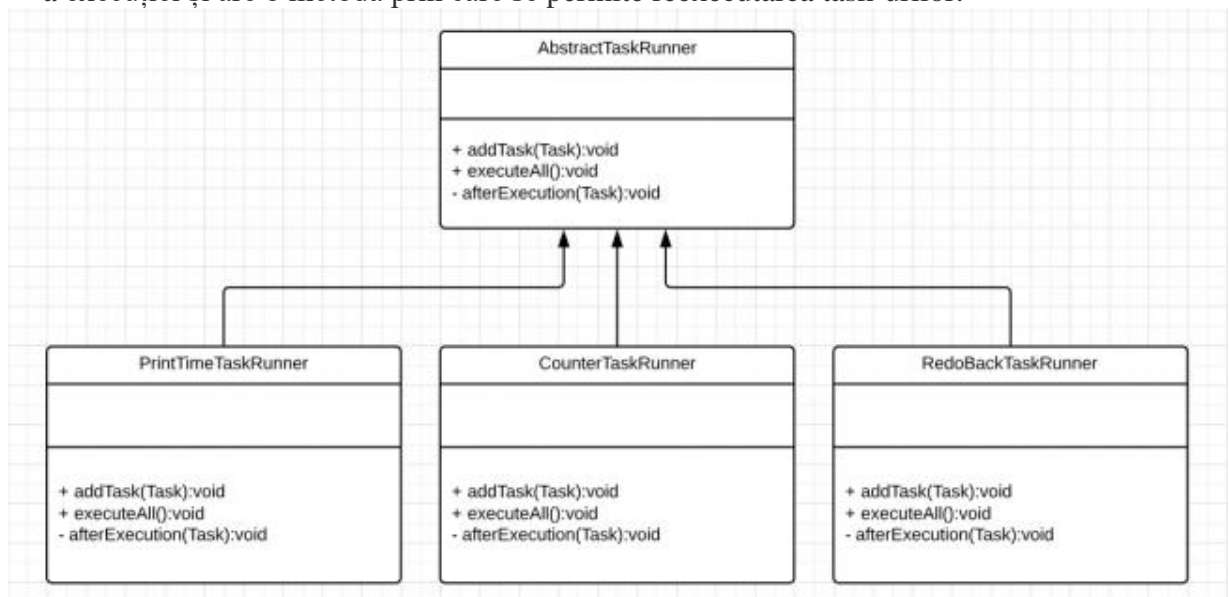


Factory cu containere ce folosesc minimum de memorie). Șablonul acesta de programare poartă denumirea de [Factory Method Pattern](#).



4. (3p) Extindeți clasa *AbstractTaskRunner* (din pachetul *fourth*) în 3 moduri:

- (1p) *PrintTimeTaskRunner* - care afișează un mesaj după execuția unui task în care se specifică ora la care s-a executat task-ul (vedeți clasa [Calendar](#)).
- (1p) *CounterTaskRunner* - incrementează un contor local care ține minte câte task-uri s-au executat.
- (1p) *RedoBackTaskRunner* - salvează fiecare task executat într-un container în ordinea inversă a execuției și are o metodă prin care se permite reexecutarea task-urilor.



### Întrebări de control

1. Ce este o clasă abstractă?
2. Ce este o interfață?
3. Care cuvinte-cheie se folosesc pentru a declara o clasă abstractă, dar o interfață?
4. Deosebirile și asemănările între o clasă abstractă și o interfață.
5. Ce înseamnă o metodă abstractă?
6. Când se folosește o clasă abstractă?
7. Care sunt cerințele obligatorii față de o clasă non-abstractă care moștenește o clasă abstractă?
8. Poate o clasă abstractă să conțină metode cu funcțional definit?

9. Când se folosește o interfață?
10. Poate oare o clasă să extindă o interfață?
11. Cum o clasă folosește o interfață?
12. Există oare o moștenire multiplă în Java și cum se reflectă? Aduceți un exemplu.
13. În ce scop o interfață poate să extindă pe o altă interfață?
14. Ce se întâmplă când mai multe interfețe implementate conțin o metodă cu același nume?