



Programare orientată pe obiecte

1. Recapitulare
2. Informații despre examen

Computer Science



Ce este o clasă?

- O clasă este *în primul rând* o descriere a obiectelor (instanțelor), clasei respective
 - Clasa conține unul sau mai mulți constructori pentru a crea obiecte
 - O clasă este un *tip*
 - Un *tip* definește un set de valori posibile și operațiile peste acele valori
 - Tipul unui obiect este clasa din care a fost creat



Ce este o clasă?

- Dar o clasă poate conține informații despre sine
 - Orice este declarat *static* aparține clasei în sine
 - Variabilele statice conțin informații despre clasă, nu despre instanțele sale
 - Metodele statice sunt executate de către clasă, nu de instanțele clasei
 - Orice care *nu* este declarat *static* *nu* este parte a clasei și nu poate fi folosit direct de către clasă
 - Cu toate acestea, o metodă statică *poate* crea (sau ei i se pot transmite) obiecte și poate trimite mesaje obiectelor



Clase

- `class MyClass extends ThatClass implements SomeInterface, SomeOtherInterface {...}`
 - La o clasă de nivel sus se poate avea acces *public* sau în pachet (implicit)
 - O clasă poate fi *final*, aceasta însemnând că nu poate fi subclasată
 - O clasă subclasează exact o singură altă clasă (implicit: clasa *Object*)
 - O clasă poate implementa oricâte interfețe



Clase

- `abstract class MyClass extends ThatClass implements SomeInterface, SomeOtherInterface {...}`
 - Aceleași reguli ca mai înainte, cu excepția faptului că o clasă abstractă *nu poate* fi final
 - O clasă *trebuie* declarată abstract dacă:
 - Conține metode abstracte
 - Implementează o interfață, dar nu definește toate metodele din interfața respectivă
 - Orice clasă *poate* fi declarată abstract
 - O clasă abstractă poate avea (și are) constructori
 - Nu se poate instanția o clasă abstractă



De ce moștenire?

- Java furnizează o bibliotecă imensă de clase anticipate
 - Câteodată aceste clasă sunt exact ceea ce este necesar
 - Alteori sunt *aproape* ce e nevoie
 - Este ușor să se subclaseze o clasă și să se suprascrive metodele al căror comportament dorim să fie diferit
- Moștenirea este o modalitate de a furniza un comportament similar diferitelor feluri de obiecte, fără a duplica codul



De ce moștenire?

- O clasă ar trebui extinsă (și să se moștenească din ea) *numai* dacă:
 - Noua clasă *este cu adevărat* un fel mai specific de superclasă **și**
 - Dorim ca noua clasă să aibă *majoritatea sau întreaga* funcționalitate a clasei pe care o extinde **și**
 - E nevoie să adăugăm sau să modificăm capacitățile superclasei
- *NU ar trebui* extinsă o clasă doar pentru a folosi doar *unele* dintre caracteristicile sale
 - În acest caz compoziția este o soluție mai bună



La ce folosesc clasele abstracte?

- Clasele abstracte sunt potrivite atunci când în mod rezonabil se poate implementa o parte, dar nu tot, din comportamentul subclaselor
- Exemplu: Avem un joc cu o tablă pe care se pot mișca diferite feluri de animale
 - Toate animalele pot `move()`, `eat()`, `drink()`, `hide()` etc.
 - Cum aceste sunt identice sau asemănătoare are sens să avem o metodă implicită `move()`, o metodă implicită `drink()` etc.



La ce folosesc clasele abstracte?

- Exemplu (continuare)
 - Dacă avem o metodă `draw()` implicită, ce ar trebui ea să deseneze?
 - Cum probabil nu dorim să avem un obiect `Animal`, ci doar anumite animale (`Dog`, `Cat`, `Mouse` etc.), nu e nevoie să putem instanția clasa `Animal`
 - Facem `Animal` abstract și cu o metodă `abstract void draw()`



Interfețe

- `interface MyInterface extends SomeOtherInterface {...}`
 - O interfață poate avea acces `public` sau în pachet
 - O interfață nu poate fi declarată `final`
 - O clasă poate implementa oricâte interfețe
 - O interfață poate *declara* (nu poate *defini*) metode
 - Toate metodele declarate sunt implicit `public` și `abstract`
 - O interfață poate defini câmpuri, clase și interfețe
 - Câmpurile sunt implicit `static`, `final` și `public`
 - Clasele sunt implicit `static` și `public`
 - O interfață *nu poate* declara constructori
 - Este OK (dar nu e necesar) să se specifice explicit atributele implicite



Declarații și atribuiri

- Presupunem că `class Cat extends Animal implements Pet {...}` și că `class Persian extends Cat {...}` și că `Cat puff = new Cat();`
- Atunci ceea ce urmează este adevărat:
 - `puff instanceof Cat`, `puff instanceof Animal`, `puff instanceof Pet`
- Următoarea *nu este* adevărată: `puff instanceof Persian`
 - Pentru a forma un test negativ, scriem `!(puff instanceof Persian)`
- Următoarele declarații și atribuiri sunt legale:
 - `Animal thatAnimal = puff;`
 - `Animal thatAnimal = (Animal)puff; // ca mai sus, dar cu specificare de tip explicita`
 - `Pet myPet = puff; // o variabila poate fi de un tip interfață`
 - `Persian myFancyCat = (Persian)puff; // face verificarea la execuție`
- Următoarea este și ea legală:
 - `void feed(Pet p, Food f) {...} // tip interfață ca parametru`



La ce folosesc interfețele?

- Moștenirea ne permite să garantăm că obiectele subclasei au aceleași metode ca obiectele superclasei sale
- Interfețele ne permit să garantăm că obiecte neînrudite au aceleași metode
 - Problemă: o GUI are o zonă în care are nevoie să *deseneze* un anumit obiect, dar nu știm încă ce fel de obiect va fi acela



La ce folosesc interfețele?

- **Soluție:**
 - Definim o interfață `Drawable`, care să aibă o metodă `draw()`
 - Facem ca tabelele, grafurile, desenele etc. să implementeze `Drawable`
 - În GUI, apelăm metoda `draw()` a obiectului (lucru legal pentru orice obiect `Drawable`)
- Dacă nu am fi avut interfețe, iată ce ar fi trebuit să facem:
 - ```
if (obj instanceof Table) ((Table)obj).draw();
else if (obj instanceof Graph) ((Graph)obj).draw();
else if (obj instanceof LineDrawing) ((LineDrawing)obj).draw();
// etc.
```
  - Mai rău, pentru a adăuga un nou tip de obiect ar trebui să schimbăm o mulțime de cod



## Clase interne

- **Clasele interne** sunt clase declarate în interiorul altor clase
- O **clasă membru** este definită imediat în interiorul altei clase
  - O clasă membru poate fi **static**
  - O clasă membru poate fi **abstract** sau **final** (dar nu amândouă)
  - O clasă membru poate fi **public**, **protected**, în pachet, sau **private**



## Clase interne

- O **clasă locală** se declară într-un constructor, o metodă, sau într-un bloc de inițializare
  - O clasă locală poate fi **abstract** sau **final** (dar nu amândouă)
  - O clasă locală poate accesa doar variabilele **final** din clasa care o conține
  - O **clasă anonimă** este un caz particular de clasă locală



## Clase interne

- O clasă internă anonimă este un fel de clasă locală
  - O clasă internă anonimă are una dintre următoarele forme:
    - `new NumeleSuperClasei(parametri) { corpul clasei }`
    - `new NumeleInterfetei() { corpul clasei }`
  - Clasele interne anonime nu pot avea constructori expliți
- O **clasă membru statică** este conținută într-o altă clasă, dar nu este de fapt o clasă internă
  - O clasă membru statică nu are acces special la numele din clasa care o conține
  - Pentru a referi o clasă internă statică, folosim sintaxa **`NumeleClaseiExterioare.NumeleClaseiInterne`**
  - O clasă membru statică poate conține câmpuri și metode statice



## La ce folosesc clasele interne?

- Uneori o clasă este necesară unei singure alte clase
  - Exemplu: o clasă pentru tratarea unui eveniment, cum este apăsarea unui buton, este probabil necesară doar în clasa GUI
  - A avea disponibilă o asemenea clasă la nivelul de sus, unde nu este nevoie de ea, doar adaugă confuzie
  - E cel mai bine să "ascundem" asemenea clase de altele pe care nu le interesează



## La ce folosesc clasele interne?

- Uneori o clasă are nevoie de acces la multe dintre variabilele și metodele altei clase
  - Iarși, tratarea evenimentelor constituie un bun exemplu
  - Făcând tratarea o clasă internă îi oferim accesul complet
- Uneori o clasă este necesară o singură dată, pentru un obiect, într-un loc anume
  - Majoritatea handler-elor de evenimente sunt astfel
  - O clasă internă anonimă este foarte convenabilă în acest scop



## Enumerări

- O enumerare, sau "enum," este pur și simplu un set de constante pentru reprezentarea diferitelor valori
- Iată modalitatea veche pentru aceasta
  - `public final int SPRING = 0;`
  - `public final int SUMMER = 1;`
  - `public final int FALL = 2;`
  - `public final int WINTER = 3;`
- Acest mod este neconvenabil și susceptibil la erori
- Iată cum se face:
  - `enum Season { WINTER, SPRING, SUMMER, FALL }`



## enums sunt clase

- O `enum` este de fapt un tip nou de clasă
  - Ea se poate declara ca fiind clasă internă sau exterioară
  - Putem declara variabile de tip `enum` și să obținem siguranța tipurilor și verificarea la compilare
    - Fiecare valoare declarată este o instanță a clasei `enum`
    - `Enums` sunt implicit `public`, `static` și `final`
    - `enums` pot fi comparate fie cu `equals` fie cu `==`



## enums sunt clase

- `enums` extind `java.lang.Enum` și implementează `java.lang.Comparable`
  - De aceea, `enums` pot fi sortate
- `Enums` suprascriu `toString()` și furnizează `valueOf()`
- Exemplu:
  - `Season season = Season.WINTER;`
  - `System.out.println(season);` // tiparește `WINTER`
  - `season = Season.valueOf("SPRING");` // setează variabila `season` la valoarea `Season.SPRING`



## Enums sunt cu adevărat clase

```
public enum Coin {
 // enums pot avea variabile instanță
 private final int value;
 // O enum poate avea constructor,
 // dar acesta nu este public
 Coin(int value) { this.value = value; }
 // Fiecare valoare din enum apelează de fapt un
 // constructor
 PENNY(1), NICKEL(5), DIME(10), QUARTER(25);
 // și, desigur, clasele au metode
 public int value() { return value; }
}
```



## Alte caracteristici ale enums

- `values()` returnează un tablou de valori `enum`
  - `Season[] seasonValues = Season.values();`
- Instrucțiunile `switch` pot lucra cu `enums`
  - `switch (thisSeason) { case SUMMER: ...; default: ... }`
  - Trebuie scris `case SUMMER;`, nu `case Season.SUMMER;`
  - Este o idee bună să includem un caz implicit (default)
- E posibil să definim corpuri de clasă dependente de valoare, astfel că fiecare valoare are propriile metode
  - Sintaxa este destul de ciudată, așa că nu o tratăm



## Clase generice

- ```
public class Box<T> {
    private List<T> contents;
    public Box() {
        contents = new ArrayList<T>();
    }
    public void add(T thing) { contents.add(thing); }
    public T grab() {
        if (contents.size() > 0) return contents.remove(0);
        else return null;
    }
}
```
- Recomandarea celor de la Sun este să se folosească o singură literă mare (cum este `T`) pentru tipuri
- Mulți ignoră această recomandare



Accesul

- Există patru tipuri de acces:
 - public** înseamnă accesibil de oriunde
 - Un câmp făcut **public** poate fi schimbat de oriunde, fără nici o protecție
 - Metodele ar trebui să fie **public** numai dacă este de dorit să poată fi apelate din afara clasei
 - protected** înseamnă accesibil din toate clasele din același director și accesibil din toate subclasele de oriunde



Accesul

- în **pachet** (implicit; fără cuvânt cheie) înseamnă accesibil din toate clasele din acest același director
- private** înseamnă accesibil doar din această clasă
 - Notă: un câmp făcut **private** nu este ascuns față de obiectele de aceeași clasă!
- În general, cel mai bine este să facem toate variabilele cât mai private cu putință și să facem metodele destul de publice pentru a fi folosite unde este nevoie



Folosirea corespunzătoare a câmpurilor

- Un obiect poate avea câmpuri și metode
 - La crearea unui obiect,
 - Acesta este creat cu toate câmpurile **ne-static** definite în clasă
 - O dată creat, el poate executa toate metodele de instanță definite de clasa sa
 - Înăuntrul unei metode de instanță, cuvântul cheie **this** se referă la obiectul care execută metoda
 - Câmpurile unui obiect trebuie să descrie **starea** obiectului
 - Toate câmpurile trebuie să spună ceva semnificativ despre obiect
 - Variabilele care nu descriu obiectul trebuie să fie variabile locale; ele pot fi transmise ca parametri unei alte metode



Folosirea corespunzătoare a câmpurilor

- Câmpurile unui obiect ar trebui să fie imposibil de modificat din exterior
 - Aceasta localizează erorile într-un obiect față de greșelile de programare din clasa sa
 - De aceea, câmpurile trebuie să fie cât mai private cu putință
 - Toate câmpurile **public** trebuie documentate cu Javadoc
 - Se pot folosi metode "getter" și "setter" pentru a verifica validitatea oricăror schimbări
 - Dacă o clasă este menită a fi subclasată, câmpurile la care subclasa necesită acces sunt în mod tipic marcate **protected**



Compoziția și moștenirea

- Compoziția** are loc atunci când un obiect dintr-o clasă *folosește* un obiect dintr-o altă clasă
 - `class MyClass { String s; ... }`
 - `MyClass` are controlul complet asupra metodelor sale
- Moștenirea** are loc atunci când o clasă *extinde* o altă clasă
 - `class MyClass extends Superclass { ... }`
 - `MyClass` primește toate variabilele statice, variabilele instanță, metodele statice și metodele de instanță ale `Superclass`, fie că dorește fie că nu
 - Constructorii *nu* se moștenesc
 - Moștenirea trebuie folosită doar atunci când putem spune cinstit că un obiect `MyClass` **este un** obiect `Superclass`
 - Bine: `class Secretar extends Angajat`
 - Rău: `class Secretar extends SystemContabil`



Constructorii

- Constructorul este *singura* modalitate pentru a crea instanțe ale unei clase
- Iată ce face un constructor:
 - Mai întâi, apelează constructorul superclasei sale:
 - `public MyClass() { super(); ... } // apel implicit (invisible)`
 - Observați că apelează constructorul *fără* argumente
 - Dar putem apela un constructor al superclasei diferit: `public MyClass(int size) { super(size); ... } // apel explicit`
 - Sau putem apela un constructor diferit al acestei clase: `public MyClass() { this(0); ... } // apel explicit`



Constructori

- Apoi, acesta adaugă câmpurile instanță declarate în această clasă (și eventual le inițializează)
 - `class MyClass { int x; double y = 3.5; ... } // în clasă, nu în constructor`
- Apoi, execută codul din constructor:
 - `public MyClass() { super(); next = 0; doThis(); doThat(); ... }`
- În sfârșit, returnează obiectul rezultat
 - Putem scrie `return`; dar nu putem spune ce să returneze



Înlănțuirea constructorilor

- *Fiecare clasă are întotdeauna un constructor*
 - Dacă nu-l scriem, Java furnizează un **constructor implicit** fără argumente
 - Dacă *scriem* un constructor, Java *nu mai* furnizează un constructor implicit
- Primul lucru pe care îl face orice constructor (cu excepția constructorului pentru **Object**) este să apeleze constructorul pentru superclasa sa
 - Aceasta creează un *lanț* de apeluri de constructori în sus pe ierarhie până la **Object**
 - Constructorul implicit apelează constructorul implicit al superclasei sale



Înlănțuirea constructorilor

- De aceea, dacă scriem o clasă cu constructor explicit cu argumente și apoi scriem subclase ale clasei respective,
 - Fiecare constructor de subclasă, în mod implicit, va apela constructorul fără argumente al superclasei (care poate să nu mai existe)
- Soluții: Fie
 - Furnizăm un constructor fără argumente în superclasă, fie
 - Apelăm explicit un anumit constructor al superclasei folosind `super(args)`



Folosirea corespunzătoare a constructorilor

- Un constructor ar trebui *întotdeauna* să creeze obiectele într-o stare *validă*
 - Constructorii nu trebuie să facă nimic altceva decât să creeze obiecte
 - Dacă un constructor nu poate garanta că obiectul construit este valid, atunci ar trebui să fie **private** și accesat prin intermediul unei metode de fabricare
 - Notă: în general, termenul de metodă de fabricare este folosit pentru a se referi la orice metodă al cărei scop principal este crearea de obiecte



Folosirea corespunzătoare a constructorilor

- O metodă de fabricare (**factory method**) este o metodă **statică** care apelează un constructor
 - Constructorul este de obicei **private**
 - Metoda de fabricare poate determina dacă să invoce sau nu constructorul
 - Metoda de fabricare poate arunca o **excepție**, sau poate face altceva potrivit în cazul în care i se dau argumente ilegale sau nu poate crea un obiect valid

```
public Person create(int age) { //exemplu: metodă de fabricare
    if (age < 0) throw new IllegalArgumentException("Too young!");
    else return new Person(n);
}
```



Referințele

- La declararea unei variabile de un tip primitiv, acesteia i se și alocă spațiu pentru a păstra valoarea
 - `int x; double y; boolean b;`
 - Dacă este declarată ca fiind câmp al unei clase, atunci valoarea ei este inițial zero (**false**)
 - Dacă este declarată ca variabilă locală, poate avea o valoare reziduală
 - La asignarea acestei valori la altă variabilă, se *copiază* valoarea



Referințele

- La declararea unui obiect, acestuia i se alocă spațiu pentru păstrarea unei *referințe* la un obiect
 - `String s; int[] counts; Person p;`
 - Dacă este declarată ca fiind câmp, valoarea sa este inițial `null`
 - Dacă este declarată ca variabilă locală, ea poate avea o valoare reziduală (ce s-a nimerit să fie acolo)
 - La asignarea acestei valori unei alte variabile i se *copiază* valoarea
 - ...dar, în acest caz, valoarea este doar o *referință* la un obiect
 - Definim* variabila asignându-i un obiect real (creat cu `new`)



Metodele

- O metodă poate:
 - fi `public`, `protected`, în pachet, sau `private`
 - fi *statică* sau de instanță
 - metodele `static` nu se pot referi la obiectul care le execută (`this`), deoarece ele sunt executate de clasa însăși, nu de către obiect
 - fi `final` sau nefinale
 - returna o valoare sau să fie `void`
 - arunca excepții
- Semnătura unei metode constă din numele și numărul și tipurile (în ordine) parametrilor săi formali



Metodele

- Supraîncărcăm* o metodă scriind o altă metodă cu același nume, dar cu semnătură diferită
- Suprascrim* o metodă *moștenită* prin scrierea unei alte metode cu aceeași semnătură
 - La suprascierea unei metode:
 - Nu o putem face mai puțin publică (`public` > `protected` > `package` > `private`)
 - Nu putem arunca excepții suplimentare (dar putem arunca mai puține)
 - Tipurile returnate trebuie să fie compatibile



Metodele

- O metodă declara *parametri formali* și este "apelată" cu *parametri actuali*
 - `void feed(int amount) { hunger -= amount; } // amount este formal`
 - `myPet.feed(5); // 5 este actual`
- Dar nu "apelăm" o metodă, ci *trimitem un mesaj unui obiect*
 - S-ar putea să nu știm ce fel de obiect este `myPet`
 - D.e. un câine poate mânca diferit de un papagal



Metodele

- La trimiterea unui mesaj, valorile parametrilor actuali înlocuiesc parametrii formali
 - Dacă parametrii sunt tipuri de obiecte, "valorile" lor sunt referințe
 - Metoda poate accesa obiectul real și, eventual, îl poate modifica
- La revenirea dintr-o metodă, parametrii formali *nu* sunt recopiați
 - Totuși, modificările făcute asupra obiectelor referite vor persista



Metodele

- Parametrii sunt transmiși prin atribuire și, de aici:
 - Dacă un parametru formal este `double`, putem apela cu un `int`
 - ...cu excepția cazului în care metoda este supraîncărcată de o alta care are un parametru `int`
 - Dacă un parametru formal este un tip clasă, putem face apelul cu un tip subclassă a clasei respective
- În cadrul unei metode de *instanță*, cuvântul cheie `this` acționează ca parametru suplimentar (specifică obiectul care execută metoda)



Metodele

- Variabilele locale nu sunt neapărat inițializate la zero (sau **false** sau **null**)
 - Compilatorul *încearcă* să vă ferească de folosirea variabilelor neinițializate
- Variabilele locale, inclusiv parametrii, sunt eliminate la revenirea dintr-o metodă
- Oricare metodă, indiferent de tipul său de retur, poate fi folosită ca instrucțiune



Metode generice

- Metodă care ia o List de Strings:


```
private void printListOfStrings(List<String> list) {
    for (Iterator<String> i = list.iterator(); i.hasNext(); ) {
        System.out.println(i.next());
    }
}
```
- Același lucru, dar cu caractere de nume global (wildcard):


```
private void printListOfStrings(List<?> list) {
    for (Iterator<?> i = list.iterator(); i.hasNext(); ) {
        System.out.println(i.next());
    }
}
```



Folosirea corespunzătoare a metodelor

- Metodele menite a fi folosite de alte feluri de obiecte trebuie să fie **public**
 - Toate metodele **public** trebuie documentate folosind Javadoc
 - Metodele **public** care pot eșua sau afecta obiectul dacă sunt invocate incorect, trebuie să arunce o excepție corespunzătoare **Exception**
- Metodele destinate folosirii interne trebuie făcute **private**
 - Metodele **private** pot folosi instrucțiuni **assert** în loc să arunce **Exceptions**
- Metodele pentru uzul intern a unei clase sau al subclaselor sale, trebuie să fie **protected**
- Metodele care nu folosesc nici o variabilă instanță sau metode de instanță trebuie să fie **static**
 - De ce să ceri un obiect când nu ai nevoie de el?



Folosirea corespunzătoare a metodelor

- Ideal, o metodă ar trebui să facă un singur lucru
 - Trebuie să descriem ce face într-o propoziție simplă
 - Numele metodei trebuie să sugereze intenția fundamentală
 - **Trebuie să fie verb**
 - Propoziția ar trebui să menționeze fiecare sursă de intrare (parametri, câmpuri etc.) și fiecare rezultat
 - Nu există noțiunea de metodă "prea mică"
- Metodele **nu** ar trebui de obicei să efectueze I/E
 - Cu excepția, desigur, a cazului în care acesta este scopul principal al metodei
 - Excepție: instrucțiuni print temporare pentru depanare
- Metodele ar trebui să facă verificări de validitate asupra intrărilor lor
 - Metodele disponibile public ar trebui să arunce excepții pentru intrările invalide



Folosirea corespunzătoare a polimorfismului

- Metode cu același nume ar trebui să facă același lucru
 - **Supraîncărcarea** ar trebui folosită doar atunci când metodele supraîncărcate fac același lucru (cu parametri diferiți)
 - Clasele care implementează o interfață ar trebui să implementeze metodele corespunzătoare să facă același lucru
 - **Suprascrierea** ar trebui făcută pentru a schimba detaliile a ceea ce face o metodă, fără a schimba ideea de bază



Folosirea corespunzătoare a polimorfismului

- Metodele trebuie să nu dublice codul din alte metode
 - O metodă supraîncărcată are același nume, dar alți parametri
 - O metodă dintr-o subclasă poate apela o metodă suprascrisă **m(args)** din superclasă folosind sintaxa **super.m(args)**
 - Tipic, acest apel ar fi făcut din metoda care suprascrie o alta, pentru a efectua ceea ce se făcea de obicei, apoi metoda care suprascrie, ar face restul



Proiectarea programelor

- Un proiect de program bun își dovedește valoarea atunci când se pune problema scrierii codului
- Proiectarea bună a programelor este mai degrabă o artă decât o știință
- În general, dorim:
 - Cel mai simplu proiect care ar putea funcționa
 - Clase care sunt de sine stătătoare și au sens luate izolat
 - Metode denumite corespunzător care fac un singur lucru și îl fac bine
 - Clase și metode care pot fi testate (cu JUnit)



Ce se întâmplă la aruncarea unei excepții

- Se creează un obiect excepție (*pe heap*)
- "Contextul" curent este stopat/abortat
- Execuția începe într-o zonă de cod pentru tratarea excepțiilor
 - Poate fi în metoda curentă
 - Poate fi extern metodei curente
- Codul de tratare a excepțiilor are acces la obiectul excepție, care poate fi folosit la
 - Accesarea mesajului String conținut în excepție
 - Determinarea tipului de excepție aruncat
 - Tipărirea unei trase de stivă (stack trace)
 - Altele (re-aruncarea excepției, incrementarea unui contor etc.)



Vectori

- Vector este o clasă care furnizează o colecție dinamică, asemănătoare unei liste înlănțuite, unei cozi etc.
- Trebuie instanțiat via "new" pentru a obține o instanță de Vector.
- Elementele vectorului sunt accesate prin intermediul diferitelor metode utilizate

Metode folosite uzual

`size()` returnează numărul curent de elemente.
`elementAt(int index)` returnează o referință la elementul de la indexul specificat.
`insertElementAt(Object obj, int index)` ca inserarea într-o listă înlănțuită (dar mai lentă); nu poate la sfârșit.
`addElement(Object obj)` adaugă la sfârșit.



Interfața utilizator grafică

- Componente, Containere, Aranjări (Layouts)
- Componentă
 - un obiect cu reprezentare grafică care poate fi afișat pe ecran și care poate interacționa cu utilizatorul.
 - d.e. Canvas, JButton, JLabel, JRadioButton, JTextField, JSlider,
- Container
 - public class **Container** extends Component
 - Un obiect container generic Abstract Window Toolkit(AWT) este o componentă care poate conține alte componente AWT.
 - Componentele adăugate unui container sunt păstrate într-o listă.
 - d.e. JFrame, JPanel

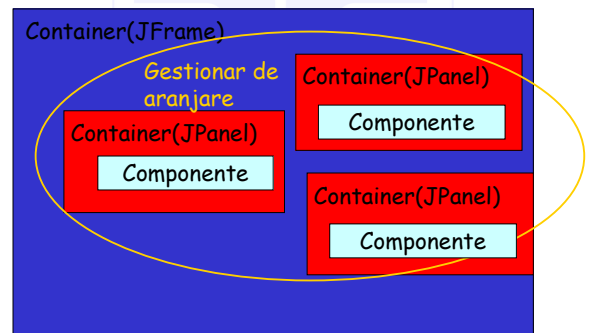


Gestionar de aranjare (LayoutManager)

- public interface **LayoutManager**
 - Definește interfața pentru clase care știu cum să aranjeze Containers.
 - d.e. BorderLayout, FlowLayout, GridLayout



Tabloul de ansamblu





Evenimente GUI

- Ce este programarea bazată pe evenimente?
- Evenimente și ascultători de eveniment
- Cum se scrie un ascultător de eveniment?
- Cum se înregistrează un ascultător de eveniment?



Applet-uri și fire de lucru

- Applet-uri față de aplicații de sine stătătoare
- Metode dintr-un applet
- Limitările applet-urilor
- Fire de lucru
 - creare, start, stop/pauză
 - Animarea applet-urilor folosind fire de lucru



I/E cu fișiere

- Bazate pe fluxuri (Streams)
 - Caracter (cunoscut și ca text)
 - Readers (Intrare) [adică FileReader]
 - Writers (Ieșire) [adică FileWriter]
 - Byte (binare)
 - InputStream (Intrare) [adică FileInputStream]
 - OutputStream (Ieșire) [adică FileOutputStream]
 - Fluxuri de prelucrare
 - Împachetează fluxuri pe Character sau Byte pentru a oferi funcționalitate suplimentară sau pentru a filtra fluxul
 - Cele mai comune: Buffered streams care permit prelucrarea linie cu linie [adică BufferedInputStream, BufferedReader]



Info despre examen

- Fără calculatoare sau telefoane celulare
- Aduceți hârtie și ustensile de scris (pix, stilou)
- Două părți:
 - Din memorie (cca. 40 min)
 - Întrebări despre OO și concepte Java
 - Să fiți capabili să comparați și să exemplificați conceptele
 - Cu documentație (cărți, note de curs, laborator) (cca. 1 oră 40 min)
 - Una sau două probleme de mică întindere de rezolvat pe hârtie
 - Puteți să aduceți ceva documentație (note de curs, de laborator, cărți) ca să vă ajute



Success!