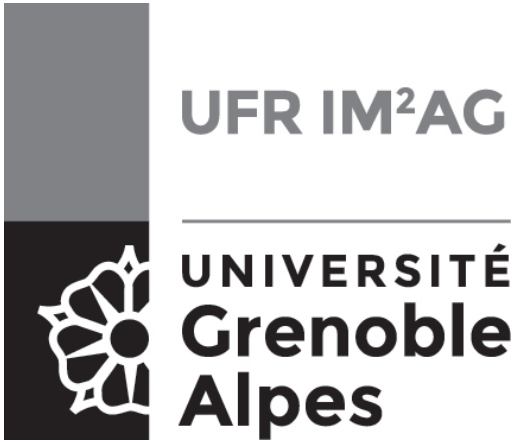


INF362
Conception et structuration de code en java

Suite de la présentation très incomplète du langage Java mettant l'accent sur les notions importantes au sein de l'UE via l'utilisation d'exemples

Surcharge

Une méthode est identifiée par sa signature : nom + type des arguments

- plusieurs méthodes peuvent avoir le même nom mais des signatures différentes, on parle alors de surcharge (*overload*)
- le compilateur retrouve la bonne méthode à partir de son nom et du type des arguments effectifs

Aussi valable pour les constructeurs

```
File() {
    buffer = new int[10];
    taille = 0;
}

File(int capacite) {
    buffer = new int[capacite];
    taille = 0;
}
```

Surcharge

Une méthode est identifiée par sa signature : nom + type des arguments

- plusieurs méthodes peuvent avoir le même nom mais des signatures différentes, on parle alors de surcharge (*overload*)
- le compilateur retrouve la bonne méthode à partir de son nom et du type des arguments effectifs

Aussi valable pour les constructeurs

```
File() {
    this(10); // limité à la première ligne
}

File(int capacite) {
    buffer = new int[capacite];
    taille = 0;
}
```

On en profite pour voir un appel entre constructeurs

Surcharge

Une méthode est identifiée par sa signature : nom + type des arguments

- plusieurs méthodes peuvent avoir le même nom mais des signatures différentes, on parle alors de surcharge (*overload*)
- le compilateur retrouve la bonne méthode à partir de son nom et du type des arguments effectifs

Aussi valable pour les constructeurs

```
File() {
    initialise(10);
}

File(int capacite) {
    initialise(capacite);
}

void initialise(int capacite) {
    buffer = new int[capacite];
    taille = 0;
}
```

Ou une construction via une méthode intermédiaire

Héritage

Une classe peut être définie comme héritant d'une autre classe, elle

- possède les méthodes et attributs de sa classe parente (superclasse)
- crée des objets manipulables via des références à la superclasse

Elle spécialise la superclasse en

- surchargeant ou redéfinissant (*override*) ses méthodes
- masquant ses attributs en en déclarant d'autres de même nom
- déclarant de nouvelles méthodes et attributs

Ses objets sont construits

- comme pour toute classe via un appel à un constructeur
- qui appelle explicitement (super) un constructeur de la superclasse
- ou implicitement le constructeur sans arguments de la superclasse

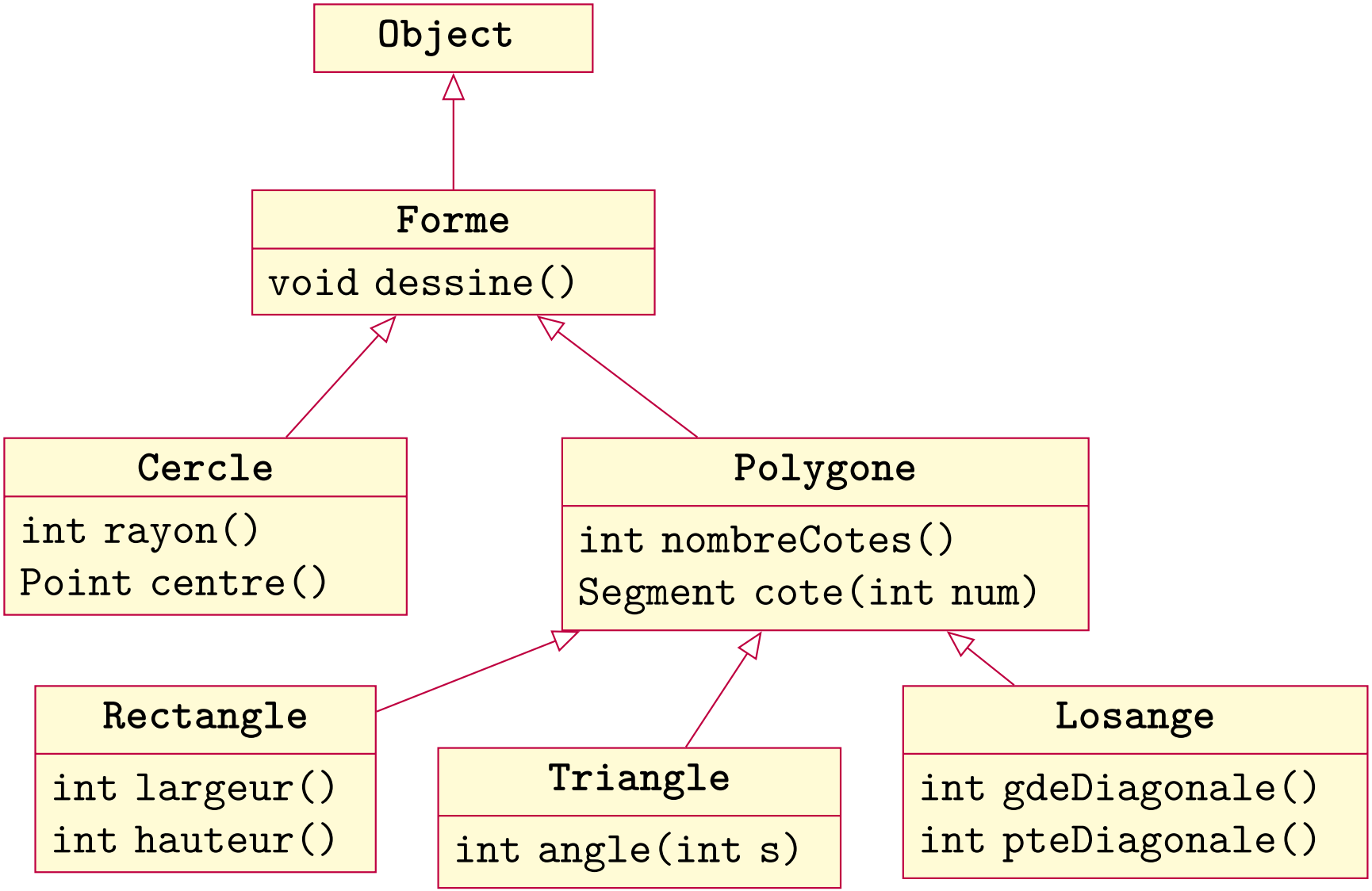
Mise en œuvre

```
class Forme {
    Color c;

    Forme() {
        this(Color.black);
    }

    Forme(Color col) {
        c = col;
    }
}

class Cercle extends Forme {
    Cercle() {
        // appel de super() par défaut
    }

    Cercle(Color col) {
        super(col);
    }
}
```

Redéfinition de méthodes, résolution virtuelle

```
class Forme {
    void dessine() { System.out.println("Forme indéterminée"); }
}

class Cercle extends Forme {
    int rayon;

    @Override                    // Optionnel, aide du compilateur
    void dessine() { System.out.println("Cercle"); }
    int rayon() { return rayon; }
}

class ex_override { public static void main(String [] args) {
        Forme f1 = new Forme();
        Cercle c = new Cercle();
        Forme f2 = c;

        f1.dessine();                          // "Forme indéterminée"
        f2.dessine();                          // "Cercle"
        System.out.println(f2.rayon());        // Erreur de compilation
        c.dessine();                           // "Cercle"
        System.out.println(c.rayon());         // Affiche le rayon
    }
}
```

Arborescence de classes



Polymorphisme

```
// Object est la classe la plus générale (ancêtre commun)
Object [] tab;
tab = new Object[10];

for (int i=0; i<tab.length; i++) {
    switch (userInput()) {
        case 0: tab[i] = new Cercle(x,y,r); break;
        case 1: tab[i] = new Rectangle(x,y,l,h); break;
        case 2: tab[i] = new Triangle(points); break;
        case 3: tab[i] = new Losange(points); break;
        default:
            throw new RuntimeException("Forme inconnue");
    }
}

for (int i=0; i<tab.length; i++) {
    // Conversion de type résolue dynamiquement
    Forme f = (Forme) tab[i];
    // Résolution virtuelle des méthodes
    f.dessine();
}
```

Classes abstraites

Dans l'exemple précédent

- la classe `Forme` ne correspond pas à une forme dessinable
- elle sert d'ancêtre commun à toutes les formes afin de pouvoir manipuler des listes de formes

Créer des objets de la classe n'a donc aucun sens, elle peut être abstraite

```
abstract class Forme {
    Color c;
    ...

    // Méthode abstraite, elle devra être concrétisée dans les
    // classes descendantes
    abstract void dessine();

    // Méthode concrète, sert d'implémentation par défaut
    String toString() {
        return "Forme indéfinie de couleur " + c;
    }
}
```

Classes anonymes

Lorsqu'il n'est pas utile de nommer une classe

- elle spécialise une classe existante
- objets manipulés via des références aux objets de la classe parente
- elle n'est instanciée qu'à un seul endroit

On peut la créer de manière anonyme

```
switch (userInput()) {
    case 0: tab[i] = new Cercle(x,y,r); break;
    case 1: tab[i] = new Rectangle(x,y,l,h); break;
    case 2: tab[i] = new Triangle(points); break;
    case 3: tab[i] = new Losange(points); break;
    default: // Classe anonyme héritant de Forme
        tab[i] = new Forme() {
            void dessine() {
                System.out.println("Forme non supportée,"+
                    "réservée aux extensions futures");
            }
        }
}
```

Cas typique des adaptateurs (interprètes entre deux interfaces)

Retour sur la pile

```
class EssaiPile {
    public static void main(String[] args) {
        Pile p, p2;

        p = new PileListe();
        p.empiler(362);
        p.empiler(42);
        p2 = new PileListe();
        while (!p.est_pile_vide()) {
            p2.empiler(p.depiler());
        }
        System.out.println(p2.depiler());
    }
}
```

Une première fabrique

design pattern permettant d'abstraire la construction

```
class PremiereFabriquePile {
    static Pile nouvelle() {
        return new PileListe();
    }
}

class EssaiPileFabrique {
    public static void main(String[] args) {
        Pile p, p2;

        p = PremiereFabriquePile.nouvelle();
        p.empiler(362);
        p.empiler(42);
        p2 = PremiereFabriquePile.nouvelle();
        while (!p.est_pile_vide()) {
            p2.empiler(p.depiler());
        }
        System.out.println(p2.depiler());
    }
}
```

Une fabrique abstraite

```
abstract class FabriquePile {
    abstract Pile nouvelle();
}

class FabriquePileListe extends FabriquePile {
    @Override
    Pile nouvelle() { return new PileListe(); }
}

class FabriquePileTableau extends FabriquePile {
    @Override
    Pile nouvelle() { return new PileTableau(); }
}
```

...en application

```
class EssaiPileFabriqueAbstraite {
    public static void main(String[] args) {
        switch (Integer.parseInt(args[0])) {
            case 0:
                test(new FabriquePileListe()); break;
            case 1:
                test(new FabriquePileTableau()); break;
            case 2:
                test(new FabriquePileListe());
                test(new FabriquePileTableau()); break;
        }
    }
    public static void test(FabriquePile f) {
        Pile p, p2;
        p = f.nouvelle();
        p.empiler(362);
        p.empiler(42);
        p2 = f.nouvelle();
        while (!p.est_pile_vide()) {
            p2.empiler(p.depiler());
        }
        System.out.println(p2.depiler());
    }
}
```

Généricité

Code paramétré par une classe

```
class MaillonGenerique<E> {
    E element;
    MaillonGenerique<E> suivant;

    MaillonGenerique(E e, MaillonGenerique<E> s) {
        element = e;
        suivant = s;
    }
}
```

Application sur l'interface

```
interface Pile {
    void empiler(int element);
    int depiler();
    boolean est_pile_vide();
}
```

devient

```
interface PileGenerique<E> {
    void empiler(E element);
    E depiler();
    boolean est_pile_vide();
}
```

Généricité (second niveau)

La classe ou interface générique peut elle même être paramètre

```
class PileListeGenerique<E> implements PileGenerique<E> {
    MaillonGenerique<E> sommet;

    public void empiler(E element) {
        MaillonGenerique<E> m;
        m = new MaillonGenerique<>(element, sommet);
        sommet = m;
    }
    public E depiler() {
        E resultat;
        // Exception si sommet == null (pile vide)
        resultat = sommet.element;
        sommet = sommet.suivant;
        return resultat;
    }
    public boolean est_pile_vide() {
        return sommet == null;
    }
}
```

Cas des tableaux génériques

```
class PileTableauGenerique<E> implements PileGenerique<E> {
    // Tableaux génériques interdits par java
    // On peut toujours passer par Object
    Object [] elements;
    int sommet;

    PileTableauGenerique() {
        elements = new Object[10];
        sommet = 0;
    }
    public void empiler(E element) {
        // Exception si sommet >= elements.length
        elements[sommet++] = element;
    }
    public E depiler() {
        // Exception si sommet <= 0
        // On limite la présence des Objects
        // à l'intérieur de la classe
        return (E) elements[--sommet];
    }
    public boolean est_pile_vide() {
        return sommet == 0;
    }
}
```

Et la fabrique ?

Classe générique en paramètre hors de propos, pas d'attributs
$\Rightarrow$ méthode générique

```
abstract class FabriquePileGenerique {
    abstract <E> PileGenerique<E> nouvelle();
}

class FabriquePileListeGenerique extends FabriquePileGenerique {
    @Override
    <E> PileGenerique<E> nouvelle() {
        // type de pile inféré grâce au type de retour
        return new PileListeGenerique<>();
    }
}

class FabriquePileTableauGenerique extends FabriquePileGenerique {
    @Override
    <E> PileGenerique<E> nouvelle() {
        return new PileTableauGenerique<>();
    }
}
```

Application (finale)

```
class EssaiPileFabriqueAbstraiteGenerique {
    public static void main(String[] args) {
        switch (Integer.parseInt(args[0])) {
            case 0:
                test(new FabriquePileListeGenerique()); break;
            case 1:
                test(new FabriquePileTableauGenerique()); break;
            case 2:
                test(new FabriquePileListeGenerique());
                test(new FabriquePileTableauGenerique()); break;
        }
    }
    public static void test(FabriquePileGenerique f) {
        PileGenerique<Integer> p, p2;
        p = f.nouvelle();
        p.empiler(362);
        p.empiler(42);
        p2 = f.nouvelle();
        while (!p.est_pile_vide()) {
            p2.empiler(p.depiler());
        }
        System.out.println(p2.depiler());
    }
}
```

Généricité (en général)

Avantages

- code plus simple qu'avec polymorphisme + cast
- "largement" présent dans la bibliothèque standard

Quelques inconvénients

- pas de tableaux génériques (vieille casserole de java 1.4)
- mécanisme complexe de contraintes sur les classes génériques
- exclut les types de base (pas de classe générique efficace ?)

Paquetages

Un paquetage est un regroupement de classes et interfaces

- espace de nommage distinct
- seule une partie du contenu du paquetage est visible en dehors
- correspond à la notion de module logiciel

On définit un module `Toto` en

- plaçant tous les fichiers qui le constituent dans un répertoire Toto situé dans l'un des répertoires contenus dans le CLASSPATH
- commençant tous ces fichiers par la ligne :
  `package Toto;`

Attention, si `Toto` est dans le répertoire courant

- `javac Toto/TotoPrincipal.java` est correct
- `cd Toto; javac TotoPrincipal.java` est incorrect
  le `.` du CLASSPATH ne contient plus le `Toto` indiqué par `package Toto` dans `TotoPrincipal.java`.

Utilisation de paquetages

Deux possibilités pour utiliser un paquetage

- nommage complet (chemin vers le nom utilisé)

```
MesPiles.Pile f;

f = new MesPiles.PileListe();
f.empiler(42);
```

- import dans l'espace de nommage principal (attention à la pollution)

```
import MesPiles.Pile;          // import des classes et
import MesPiles.PileListe; // interfaces utilisées
// ou
import MesPiles.*;             // import de toute la partie
...                            // publique du paquetage
    Pile f;
    f = new PileListe();
    f.empiler(42);
```

Utilisation de paquetages de la bibliothèque standard

Deux possibilités pour utiliser un paquetage

- nommage complet (chemin vers le nom utilisé)

```
java.util.Random r;

r = new java.util.Random(graine);
System.out.println("Entier dans [0;9] : " +
    r.nextInt(10));
```

- import dans l'espace de nommage principal (attention à la pollution)

```
import java.util.Random; // import de la classe Random
// ou
import java.util.*;      // import de tout java.util
...
    Random r;

    r = new Random(graine);
    System.out.println("Entier dans [0;9] : " +
        r.nextInt(10));
```

Attention, organisation hiérarchique mais import à plat
`import java.util.*` n'inclut pas `import java.util.regex.*`

Visibilité

Qualificatif définissant où une classe/interface est accessible

- `public`, partout
- par défaut, dans le paquetage englobant

Qualificatif définissant où une méthode/attribut est accessible

- `public`, partout
- `protected`, dans le paquetage englobant et les classes descendantes
- par défaut, dans le paquetage englobant
- `private`, dans la classe

Avec les règles par défaut

- tout est caché dans le paquetage de définition (modularité)
- interface du paquetage avec l'extérieur explicitement `public`

Autres attributs

Qualificatifs associables aux méthodes

- `static` : la méthode est liée à une classe
  - appel via le nom de la classe
  - pas de `this`
- `final` : méthode non redéfinissable

Qualificatifs associables aux attributs

- `static` : l'attribut est lié à une classe
  - accès via le nom de la classe
  - instance unique, commune à tous les objets de la classe
- `final` : attribut constant