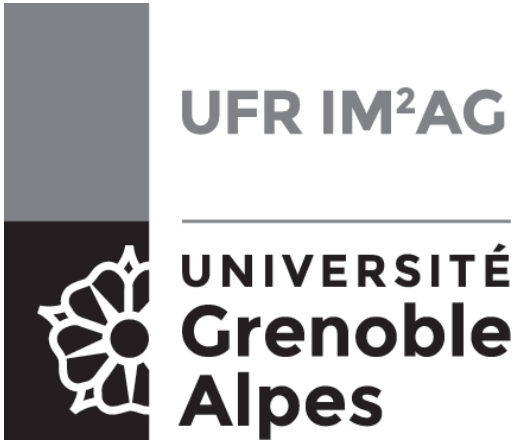


DU-ISN
Un soupçon d'objet

Présentation succinte et concise d'un petit bout du langage ayant pour objectif un passage rapide à la pratique

Notion d'objet

Objet : **combinaison** de données et d'opérations applicables dessus

- **attributs** : données stockées dans l'objet
- **méthodes** : opérations applicables sur les données de l'objet

Exemples

- une fiche de personnel
  - regroupe le nom, le prénom et l'age d'une personne
  - peut être affichée
- un nombre complexe
  - regroupe une partie réelle et une imaginaire
  - peut être additionné, multiplié, etc. avec d'autres nombres complexes
  - peut être affiché

Intérêt : l'**encapsulation**, on regroupe et cache dans l'objet le détail de son contenu et des ses traitements

Notion de classe

Classe : **modèle** permettant de créer des objets

- **attributs** : données stockées dans les objets de la classe
- **méthodes** : opérations applicables sur les objets de la classe

En java, une classe

- est associée à un type de même nom désignant une référence ($\approx$ pointeur) à un objet de cette classe
- peut être instanciée en un objet à l'aide de l'opérateur **`new` qui renvoie une référence à l'objet** ainsi créé

Chaque objet

- contient ses propres attributs selon le modèle indiqué dans sa classe
- est manipulé par une référence à l'aide de laquelle on peut appeler les méthodes de la classe

Remarque : un objet n'a pas besoin d'être détruit (*garbage collector*)

Par rapport aux langages non orienté objet

On retrouve des notions comparables

- la classe est à l'objet ce que le type est à la variable
- les attributs correspondent aux différents champs d'une définition de type structurée
- les méthodes sont similaires à des fonctions ayant un argument implicite, leur objet

De petites différences justifient les termes différents

- attributs forcément attachés à un objet, $\neq$ variables
- une méthode s'appelle à partir d'un objet : argument implicite
- références $\neq$ pointeurs : pas d'arithmétique ni de valeur absolue

Exemple

```
public class Personnel {
    String nom, prenom;
    int age;

    public String toString() {
        return prenom + " " + nom + ", " + age + " ans";
    }
}

class Exemple {
    public static void main(String[] args) {
        Personnel p;
        p = new Personnel();

        p.nom = "Huard";
        p.prenom = "Guillaume";
        p.age = 20;
        System.out.println(p.toString());
    }
}
```

A propos des objets

Construction, `new`

- **alloue la mémoire** nécessaire au stockage de l'objet (attributs, ...)
- **appelle un constructeur** de la classe sur l'objet créé
- est la seule manière de créer un objet (pas d'allocation statique)

Dans une méthode appelée depuis un objet

- attributs accessibles directement (objet implicite)
- peut être explicité avec `this` (référence à l'objet implicite)

Exemple avec constructeur

```
public class Personnel2 {
    String nom, prenom;
    int age;

    Personnel2(String n, String p, int age) {
        nom = n; prenom = p;
        // attribut age masqué récuperré via this
        this.age = age;
    }
    public String toString() {
        return prenom + " " + nom + ", " + age + " ans";
    }
}

class Exemple2 {
    public static void main(String[] args) {
        Personnel2 p;
        p = new Personnel2("Huard", "Guillaume", 20);
        System.out.println(p.toString());
    }
}
```

Parenthèse : 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*)
- java trouve la bonne méthode avec le type des arguments effectifs

```
class Complexe {
    double reelle;
    double imaginaire;

    void additionne(double r, double i) {
        reelle += r;
        imaginaire += i;
    }

    void additionne(Complexe c) {
        reelle += c.reelle;
        imaginaire += c.imaginaire;
    }
}
```

Parenthèse : 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*)
- java trouve la bonne méthode avec le type des arguments effectifs

Aussi valable pour les constructeurs

```
class Complexe {
    double reelle, imaginaire;

    Complexe(double r, double i) {
        reelle = r;
        imaginaire = i;
    }

    Complexe(double r) {
        reelle = r;
        imaginaire = 0;
    }
}
```

Parenthèse : 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*)
- java trouve la bonne méthode avec le type des arguments effectifs

Aussi valable pour les constructeurs

```
class Complexe {
    double reelle, imaginaire;

    Complexe(double r, double i) {
        reelle = r;
        imaginaire = i;
    }

    Complexe(double r) {
        // Appel d'un autre constructeur (1ere ligne)
        this(r, 0);
    }
}
```

Héritage

Une classe peut hériter d'une autre classe

- elle **hérite des méthodes et attributs** de sa superclasse (parente) $\iff$ ils sont implicitement présents dans les objets de la classe dérivée
- ses objets sont aussi manipulables via des références à la superclasse

Elle spécialise la superclasse, on peut

- **redéfinir** ses méthodes (*override*)
- **masquer** ses attributs en en déclarant d'autres de même nom
- **déclarer** de nouvelles méthodes et attributs

Ses objets sont construits de manière incrémentale

- via un appel à un constructeur (comme pour toute classe)
- qui appelle explicitement un constructeur de la superclasse (super)
- 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);
    }
}
```

Polymorphisme

Outre la factorisation de code, l'héritage apporte le polymorphisme :
Interface unique applicable à des entités de **différents types**

- les classes héritant d'une même classe sont distinctes ($\neq$ types)
- les méthodes sont héritées (même interface que la classe parente)
- voire adaptées au nouveau type (redéfinition)

Dans l'exemple précédent

- un cercle est une forme
- autres formes possibles : rectangle, losange, triangle, ...
- on peut définir une méthode de dessin dans la forme
- et la redéfinir dans les formes dérivées pour l'adapter à chaque forme

Toutes les formes spécialisées sont manipulables comme des formes générales mais se dessinent spécifiquement

Redéfinition de méthodes, résolution virtuelle

```
class Forme {
    void dessine() { System.out.println("Indéterminé"); }
}
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();                       // "Indéterminé"
        f2.dessine();                       // "Cercle"
        System.out.println(f2.rayon()); // Erreur compil.
        c.dessine();                        // "Cercle"
        System.out.println(c.rayon());  // Affichage
    }
}
```

Polymorphisme, interface commune et résolution virtuelle

```
Forme [] tab;
tab = new Forme[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;
    }
}

for (int i=0; i<tab.length; i++) {
    // Résolution virtuelle des méthodes (dessin correct)
    tab[i].dessine();
}
```

Relations d'héritage et `Object`

Dans un programme en java

- Une classe ne peut hériter, au plus, que d'une seule autre classe
- La classe `Object`, est parent implicite d'une classe n'ayant pas de relation d'héritage explicite
- `Object` n'hérite d'aucune classe, elle est ancètre de toutes les autres

L'ensemble des classes forme une arborescence

Le type de référence `Object` peut être utilisé comme un type générique

- pour stocker une référence à tout objet

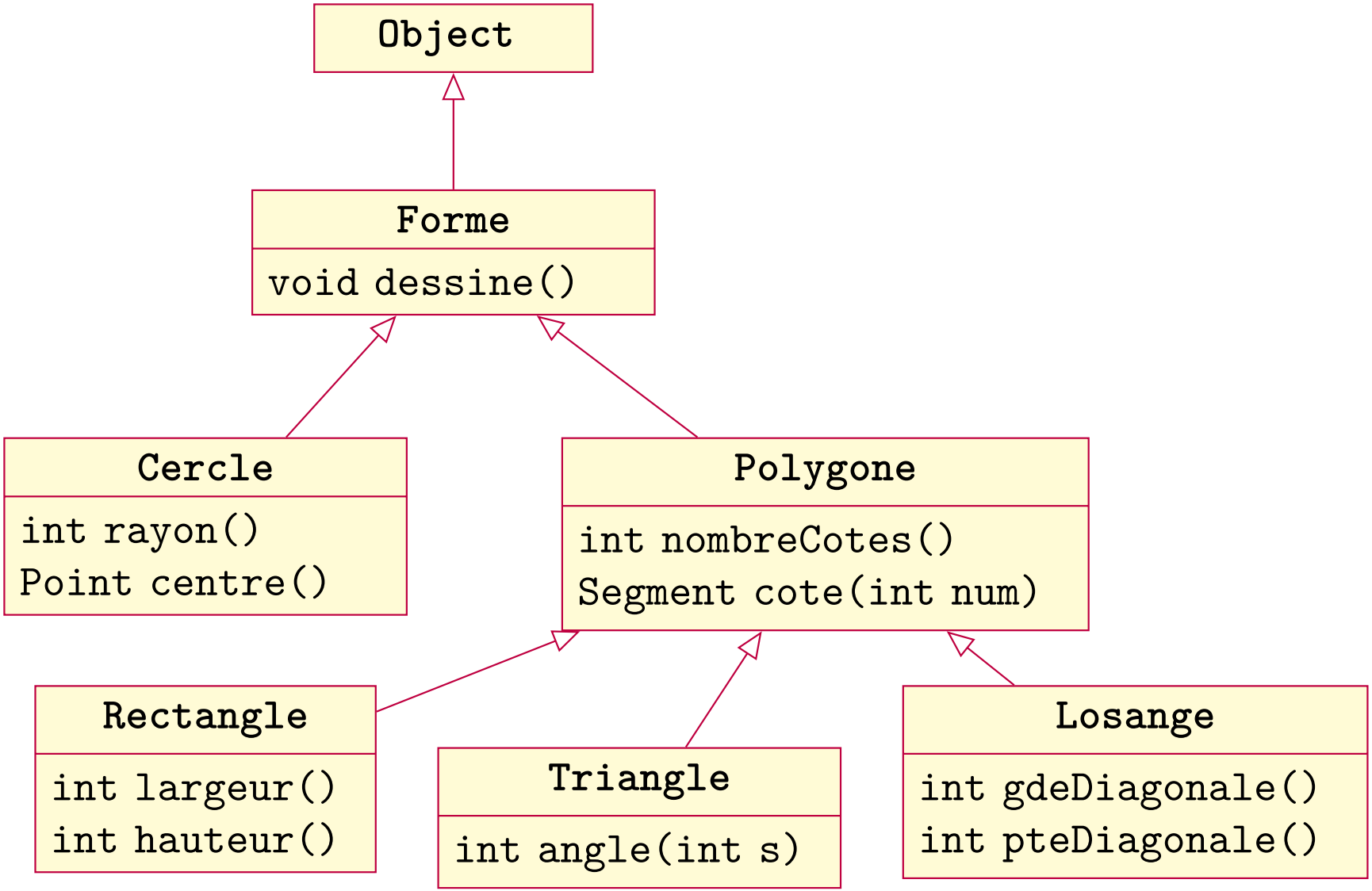
```
Object o = new Forme();
```

- dont on peut tirer dynamiquement une référence d'un autre type

```
Forme f = (Forme) o;
f.dessiner();
```

Si, bien entendu, l'objet référencé est bien du type demandé !

Arborescence de classes



Méthodes d'`Object`

La classe `Object` définit quelques méthodes, dont :

- `toString` : permet d'obtenir une représentation textuelle de l'objet. Utilisée par défaut dans les contextes où une chaîne est attendue (affichage, concaténation, ...)
- `equals` : test d'égalité entre l'objet appelant et celui donné en paramètre

Peuvent être redéfinies pour les spécialiser dans les classes filles

Exemple

```
public class Personnel3 {
    String nom, prenom; int age;

    Personnel3(String n, String p, int age) {
        nom = n; prenom = p;
        this.age = age;
    }
    @Override
    public String toString() {
        return prenom + " " + nom + ", " + age + " ans";
    }
    @Override
    public boolean equals(Object o) {
            Personnel3 p = (Personnel3) o;
            if ((age == p.age) && nom.equals(p.nom)
                && prenom.equals(p.prenom))
                return true;
            else
                return false;
    }
}
```

Exemple

```
class Exemple3 {
    public static void main(String[] args) {
        Personnel3[] p;

        p = lireFichierPersonnel();
        for (int i = 0; i < p.length; i++)
            for (int j = i + 1; j < p.length; j++)
                // Le test d'égalité doit appeler
                // explicitement la fonction
                if (p[i].equals(p[j]))
                    // Seule toString est implicite
                    // pour la conversion en chaine
                    System.out.println("La fiche " +
                                       p[i] + " est en double");
    }
}
```

Exceptions

Objets particuliers servant à gérer les erreurs

- sont d'une classe ayant `Exception` pour ancètre
- peuvent être "lancés" (`throw`, levée d'exception), dans ce cas :
  - l'erreur est gérée par le programmeur (nous verrons plus tard)
  - le programme s'arrête et affiche l'erreur
- le compilateur force la gestion des exception sauf pour les `RuntimeException` et dévrivées

Nous nous contenterons de `RuntimeException` dans un premier temps

Exemple

```
public class Personnel4 {
    String nom, prenom;
    int age;

    Personnel4(String n, String p, int age) {
        nom = n;
        prenom = p;
        if (age >= 0)
            this.age = age;
        else
            throw new RuntimeException("Age invalide");
    }
}
```