

*Eric Anquetil (eric.anquetil@irisa.fr)
Dépt. Informatique
Insa Rennes*

Initiation à la Programmation Orientée Objets en Java

Version 2.0
Module de Pre-Spécialisation

Sommaire

❖	CHAP. 1 : Préambule	
—	Une définition de la programmation	6
—	Langage et Programmation	7
❖	CHAP. 2 : Introduction	
—	Portabilité de Java	9
—	Java et Internet	10
—	Java Virtuelle Machine (JVM) / sécurité	11
—	Programmes Java de 2 types	12
—	Plate-forme de développement (JDK)	13
—	Bibliothèques Java	14
—	Environnement intégré de développement : IDE	15
❖	CHAP. 3 : Introduction à la Programmation Orientée Objet	
—	Introduction à la programmation orientée objet	17
—	Notion d'objet	18
—	Notion de classe : abstraction	19
—	1er diagramme de classes UML	20
—	Notion d'agrégation	21
—	Notion d'héritage	22
—	Notion d'héritage et de Polymorphisme	23
❖	CHAP. 4 : Objets et Classes	
—	Les objets Java	25
—	Les types primitifs	26
—	Exemple : la classe Point	27
—	Invocation des méthodes et attributs / this	28
—	Surcharge de méthodes	29
—	Paramètres des méthodes	30
—	Paramètres des méthodes (bis)	31
—	Encapsulation et contrôle d'accès	32
—	Définition d'une classe : résumé	33
❖	CHAP. 5 : Unité de compilation	
—	Unités de compilation □ Fichier	35
	— Exemple d'unité de compilation / main	36
❖	CHAP. 6 : Packages	
—	Utilisation des Packages : espace de nommage	38
—	Définition de Package - contrôle d'accès	39
❖	CHAP. 7 : Egalités d'objets (1ère approche)	
—	Égalité d'objets (surcharge)	41
❖	CHAP. 8 : Membre statique	
—	Membres statiques : static	43
❖	CHAP. 9 : Donnée constante	
—	Données constantes : final	45
❖	CHAP. 10 : Agrégation	
—	Agrégation : classe Point / Rectangle	47
—	Agrégation : exemple de la classe Rectangle	48
❖	CHAP. 11 : Recopie	
—	Référence : copie superficielle	50
—	Référence : notion de copie en profondeur	51
❖	CHAP. 12 : Exercice : Classe Polygone	
—	Définition de la classe Polygone	53
—	Diagramme de classes : Point / Polygone	54
—	Polygone : 1ère version	55
—	Polygone : translation d'un polygone fermé	56
—	Polygone : problème de référence	57
—	Polygone : notion de recopie	58
—	Nouveau diagramme de classes : Point / Polygone	59
❖	CHAP. 13 : Gestion mémoire	
—	Durée de vie des objets	61
—	Le ramasse miette ou « garbage collector »	62
—	La méthode finalize()	63
❖	CHAP. 14 : Héritage	
—	Introduction	65
—	Héritage : diagramme de classe (Personne /employe)	66

—	Héritage : la classe Personne	67	—	Collection Java : interfaces / classes abstraites	98
—	Héritage : constructeur / extends	68	—	Collection Java : diagramme de classes du framework	99
—	Héritage : des attributs et des méthodes	69	—	Collection Java : les iterateurs	100
—	Héritage : enrichissement d'une méthode	70	—	Collection Java : un petit exemple	101
—	Héritage : redéfinition d'une méthode	71	❖	CHAP. 21 : Exception	
—	Héritage : redéfinition et attachement dynamique	72	—	Notion d'erreur / Exception	103
—	Héritage : exemple / attachement dynamique	73	—	Traitement des cas d'erreurs exceptionnelles	104
—	Héritage : accès aux membres propres de l'objet dérivé	74	—	Exemple : mécanisme d'exception / finally	105
—	Héritage : instanceof	75	—	Réception / traitement d'une exception	106
❖	CHAP. 15 : Contrôles d'accès		—	Lancement d'exceptions / throw	107
—	Synthèse des différents contrôles d'accès en Java	77	—	Capture et traitement d'exceptions / try ... catch	108
❖	CHAP. 16 : Classe Object		—	Retransmission d'une exception	109
—	Définition de java.lang.Object	79	—	Exception standard en Java	110
❖	CHAP. 17 : Egalités d'objets (redéfinition)		—	Créer vos propres classes d'exceptions : Throwable	111
—	Égalité d'objets (surcharge de equals)	81	—	Créer vos propres classes d'exceptions : MonException	112
—	Égalité d'objets (redéfinition de equals)	82	—	Exemple complet	113
❖	CHAP. 18 : Classe abstraite		❖	CHAP. 22 : Clonage	
—	Classe et méthode abstraites : abstract	84	—	Notion de copie en profondeur	115
—	Exemple de Classe et méthode abstraites	85	—	La méthode clone() de la classe Object	116
❖	CHAP. 19 : Interface		—	Mise en œuvre de la méthode clone()	117
—	Interface Java : implements	87	—	Mise en œuvre de la méthode clone()	118
—	Exemple d'utilisation d'interface 1/2	88	—	Clonage et héritage	119
—	Exemple d'utilisation d'interface 2/2	89	—	Remarques sur le clonage en Java	120
—	Interface : Diagramme de Classes	90	❖	CHAP. 23 : Classe interne	
—	Interface et héritage	91	—	Classe interne : inner class	122
—	Interface Versus Classe abstraite	92	—	Classe interne : membre d'une classe	123
❖	CHAP. 20 : Collection Java		—	Classe interne : locale	124
—	Les collections Java : Principe	94	—	Classe interne : anonyme	125
—	Les collections Java : Vector, List, ...	95	—	Classe interne : anonyme	126
—	Collection Java : l'interface collection	96	❖	CHAP. 24 : IHM : AWT	
—	Collection Java : les interfaces	97	—	2 mots sur l'AWT et JFC	128

—	Hiérarchie de composants et conteneurs	129	—	Les Generics – Définition de “generics”	160
—	Exemple 1 : une JFrame	130	—	Les Generics – Définition de “generics”	161
—	Exemple 2 : JFrame / JLabel	131	—	Les Generics – Héritage	162
—	Exemple 3 : JFrame / JPanel	132	—	Les Generics – Wildcards et Bounded Wildcards	163
—	Exemple 4 : JFrame / MenuBar	133	—	Les méthodes Generic	164
—	Principe de la gestion des événements (JDK 1.1 et +)	134	❖	CHAP. 28 : Remarques	
—	Évènements et Interfaces "Listener"	135	—	Quelques Remarques 1/2	166
—	Les interfaces "Listener"	136	—	Quelques Remarques 2/2....	167
—	Exemple 5 : Évènements et JFrame	137	❖	CHAP. 29 : Annexe : Java 5 (Tiger)	
—	Utilisation d'un "Listener Adapter"	138	—	Introduction : Java5 (Tiger)	169
—	Exemple 6 : JFrame / JButton	139	—	Autoboxing et Auto-Unboxing des types primitifs	170
—	Objets Sources et Objets délégués	140	—	Itérations sur des collections : nouvelle boucle for	171
❖	CHAP. 25 : Exemple : TP / application graphique		—	Type énuméré : enum	172
—	Présentation de L'IHM	142	—	Nombre d'arguments variable pour une méthode	173
—	Préambule : les figures géométriques de base	143	—	Printf : Sortie standard formatée	174
—	Les figures géométriques de base / Ellipse	144	—	Le scanner : entrée formatée	175
—	L'Architecture : 1 premier diagramme de Classe	145	—	Les imports statiques	176
—	Le Panel des Icones / Constructeur	146			
—	Le Panel des Icones / Paint	147			
—	Le Panel des Icones / Gestion des évènements	148			
—	IHM : ce qu'il reste à faire	149			
—	L'Architecture : diagramme de Classe avec gestion des evts	150			
❖	CHAP. 26 : Introduction aux Applets				
—	« Applet » : utilisation d'un framework simple	152			
—	« Applet » : chargement d'une Applet	153			
—	« Applet » : démonstration	154			
—	« Applet » : un exemple / code	155			
—	« Applet » : invocation de l'Applet / html	156			
—	« Applet » : action du paint	157			
❖	CHAP. 27 : Introduction aux génériques				
—	Les Generics – Introduction	159			

A decorative graphic consisting of several concentric, overlapping arcs in a light blue color, creating a sense of motion or a stylized 'C' shape, positioned behind the chapter title.

— Chapitre 1

Préambule

- ❖ **Premier aspect : la notion de programmation dite « impérative »**
>> succession d'instructions qui produisent le résultat souhaité
 - **Notion de calcul / illustration : une recette**

1. Mettre 300g de farine
 2. De la levure de boulanger
 3. Verser l'eau jusqu'à obtenir une pâte homogène
 4. Etc.

Faire la pâte
- ❖ **Second aspect : organisation, coopération et coordination de plusieurs entités**
 - **Notion d'interaction / illustration : un restaurant :**
 - Ensemble de personnes (*entités*) qui ont des *tâches* à réaliser (*séquences d'instructions*) en *interaction mutuelle* avec leur milieu extérieur (*communication*).
- ❖ **Ces 2 aspects sont omniprésents en programmation : base de données, interface homme-Machine, réseau Internet, ...**

❖ Langage de programmation :

- Définition d'une syntaxe précise, de structures de contrôle, structures de données, de séquences d'instructions prédéfinies (« haut niveau »), etc.

❖ Langage et paradigme de programmation

- Un langage est "souvent" associé à un paradigme de programmation, cad à une approche pour définir, organiser, utiliser la connaissance.
 - Programmation fonctionnelle : scheme, camel, ...
 - Centrée sur une décomposition en fonctions qui rendent des résultats
 - Programmation impérative : Pascal, C, ...
 - Centrée sur une structuration « séquentielle » des traitements
 - Programmation objet : C++, Java, Eiffel, ...
 - Centrée sur les données / associées à leurs traitements spécifiques

❖ Apprendre un langage et savoir programmer

- ce n'est pas seulement apprendre une syntaxe et des mots clés,
- c'est surtout apprendre à aborder et résoudre un problème par rapport au paradigme sous tendu par le langage

A decorative graphic consisting of several concentric, overlapping circular arcs in a light blue color, creating a sense of motion or a stylized 'C' shape, positioned behind the chapter title.

Chapitre 2

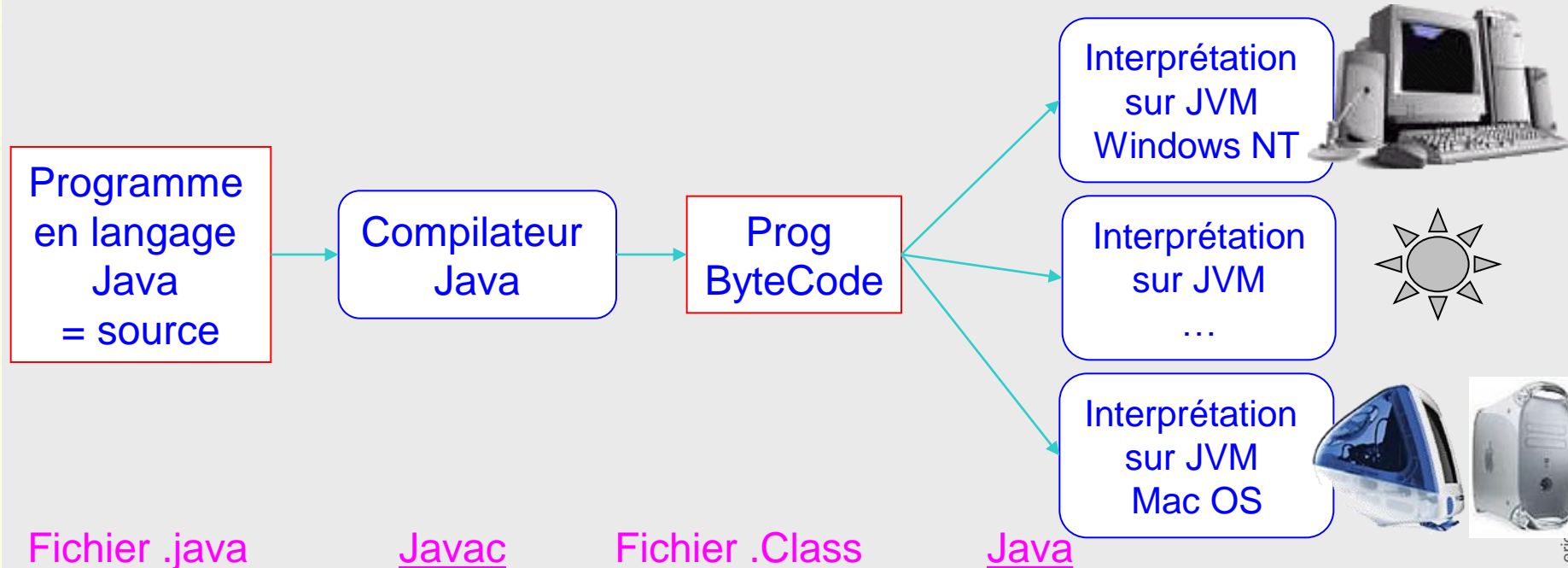
Introduction

❖ Java est indépendant du matériel (portable)

- le même programme peut s'exécuter sur n'importe quelle plate-forme (PC, Mac, Linux...)

❖ Principe de la portabilité : Java est interprété

- Le compilateur Java (javac) produit du pseudo-code (*byte code*) indépendant de la plate-forme et interprétable (java) par une machine virtuelle : la *Java Virtual Machine (JVM)*
- Chaque plate-forme possède sa propre machine virtuelle



❖ **Java et Internet**

■ **Objectif**

- Écrire un programme sur une machine, le transmettre sur le réseau et l'exécuter sur des machines totalement différentes, quel que soit leur système d'exploitation (Windows NT, Linux...).

■ **Caractéristiques nécessaires**

- **Robustesse**
 - minimisation des risques de « crash » (beaucoup de vérifications faites par le compilateur Java)
- **Portabilité**
 - indépendance vis-à-vis du système d'exploitation (grâce à la JVM)
- **Sécurité**
 - pas de virus... (la JVM masque la machine)

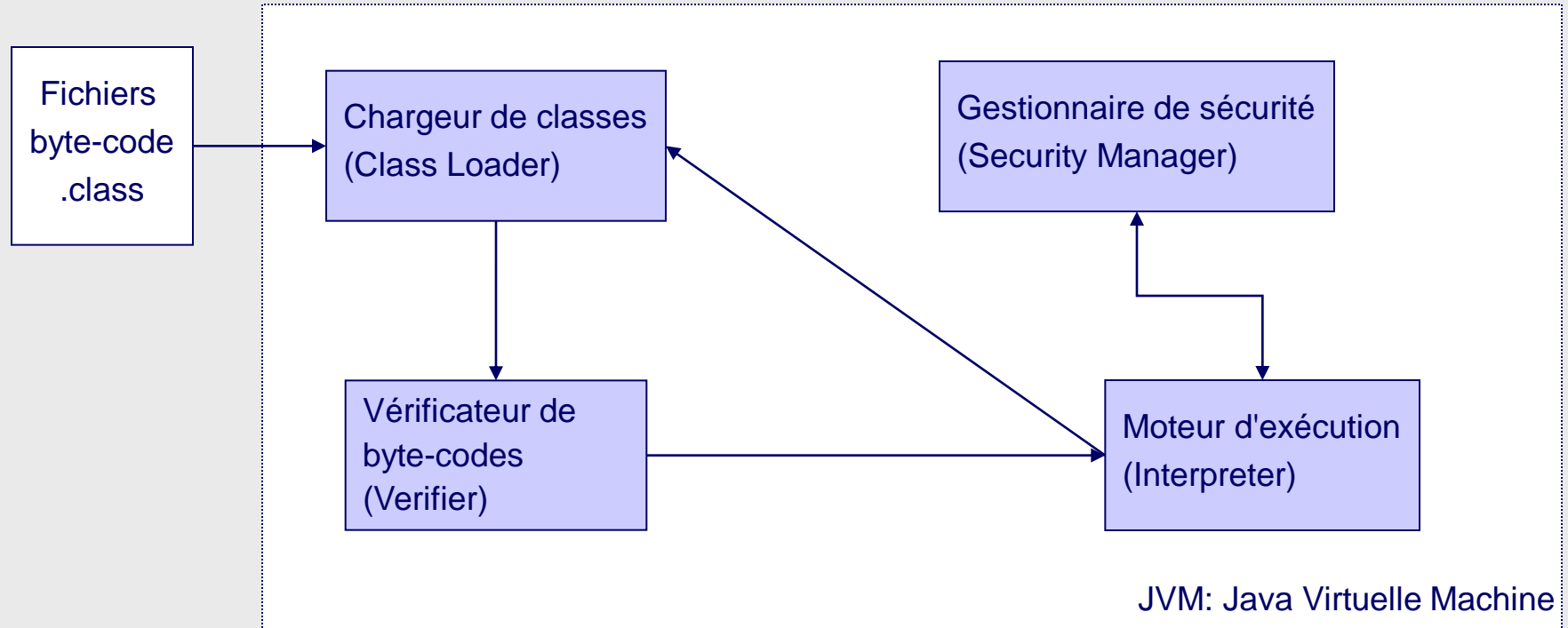
Chargement dynamique
des classes

Copie en mémoire des classes demandées
lors de l'exécution du programme

Protection dynamique

Vérification des accès aux ressources

Modulable selon le contexte d'exécution
(programme, Applet, Applet signées)

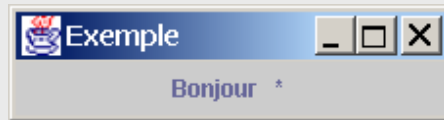


JVM: Java Virtuelle Machine

Vérification statique
du byte-code

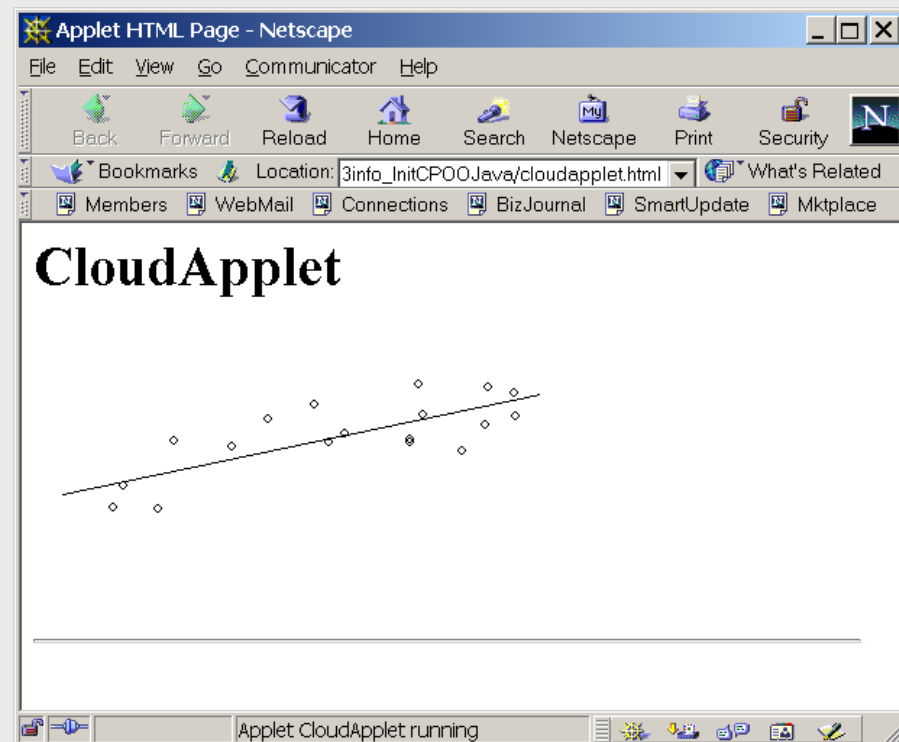
❖ Java permet de développer deux types de programmes selon leur contexte d'exécution

■ Les applications autonomes



■ Les Applets

- Programmes Java destinés à fonctionner via un navigateur Web
- Programmes chargés à travers une page HTML



- ❖ **Outils nécessaires pour développer des programmes Java**
 - Un éditeur de texte (notepad, xemacs, ...) pour écrire le programme
 - Un compilateur Java
 - Un interpréteur de byte code
- ❖ **Le JDK (Java Development Kit) est téléchargeable gratuitement il contient notamment :**
 - **javac : compileurs de sources java**
 - fichier source .java ---> byte-code .class
 - **java : interpréteur de byte code**
 - fichier byte-code .class est interprété
 - **jdb : débogueur**
 - pour détecter les erreurs dans un programme
 - **javadoc : génération de documentation en HTML**
 - **Appletviewer : permet de tester une *applet* (sans passer par un browser Web)**
 - **Les API (Application Programming Interface)**
 - un ensemble complet de modules prédéfinis pour le développement d'applications en Java (graphisme 2D, 3D, communication Internet, ...)

❖ Java offre un ensemble complet de bibliothèques standardisées :

cf. pour une documentation détaillée : <http://java.sun.com>

- java.lang : classes de base du langage (Class, Character, Math, Process, ...)
- java.util : structures de données (Collections, HashMap, Vector, ...)
- java.io : entrées - sorties classiques (File, InputStream, OutputStream, ...)
- java.awt : interface Homme-Machine (Button, Canvas, Graphics2D, Image, ...)
- java.net : communication internet (URL, Socket, ...)
- java.applet : gestion des programmes destinés à fonctionner via un browser Web
- ...

❖ D'autres bibliothèques

- JGL (extension des structures de données), JFC (extension de l'AWT), ...

❖ IDE : Environnement intégré de développement

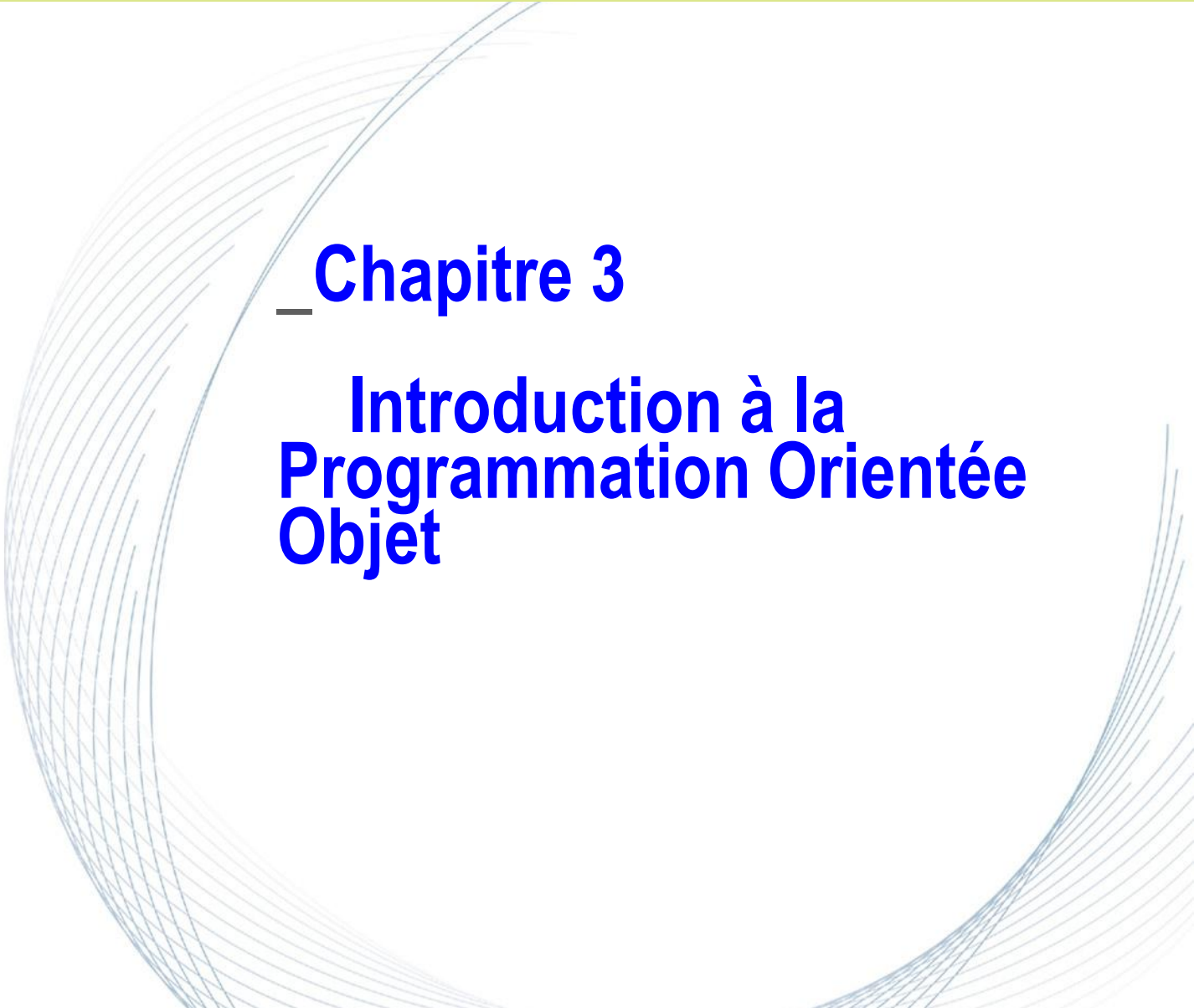
- Objectif : faciliter au maximum le travail de développement
- ex : Eclipse (Ibm), Jbuilder (Borland), Visual Café (symantec), ...

❖ Ils intègrent notamment sous un environnement commun :

- un éditeur
- une interface pour compiler, exécuter, générer et visualiser la documentation...
- des outils de développement visuels pour :
 - la création d'interfaces graphiques
 - la détection d'erreurs, le parcours du code, ...
- des outils de gestion de projets, de travail en groupe

❖ Exemple : L'EDI Eclipse

- Plate-forme modulaire (plug in) pour les développements informatiques
 - Java, UML, C++, etc.
- IBM en est à l'origine : aujourd'hui sous licence publique
 - cf. <http://www.eclipse.org/>

A decorative graphic consisting of several concentric, overlapping circular arcs in a light blue color, creating a sense of motion or a stylized 'C' shape, positioned behind the main title text.

Chapitre 3

Introduction à la Programmation Orientée Objet

❖ La programmation orientée objet est axée

- 1/ sur les données / encapsulées autour d'un concept appelé : « objet »
- 2/ sur les traitements qui y sont associés.
 - la complexité des programmes est alors répartie sur chaque « objet ».
 - les langages objets (C++, smaltalk, java, ...) facilitent grandement cette approche de programmation.

❖ Le modèle objet est utilisable aussi bien pour

- l'analyse, la conception, le codage.
 - cohérence pendant le développement
 - facilite la réalisation de programme : développement et exploitation de gros logiciels

❖ Principes de base

- abstraction, encapsulation, agrégation, héritage, polymorphisme

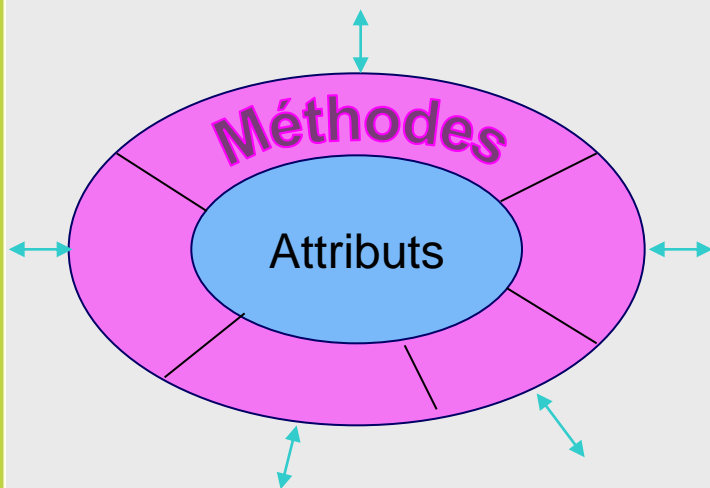
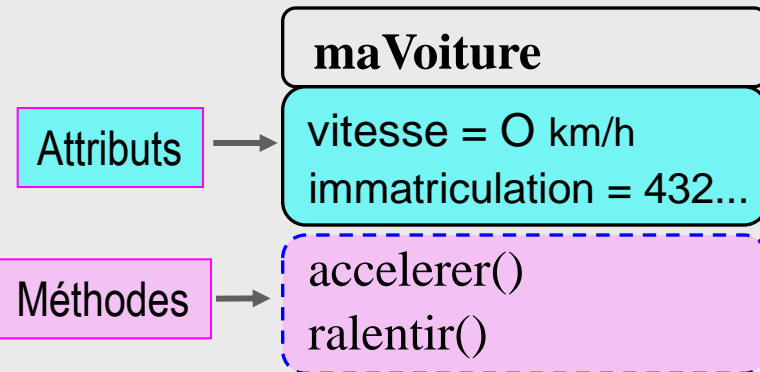
❖ Les objets / Encapsulation :

■ Un objet est constitué :

- d'une collection **d'attributs** (définissant **l'état** de l'objet)
- et de **méthodes** associées (définissant son **comportement**)

■ Modularité et masquage de l'information

- les méthodes constituent l'interface de communication.
- la manipulation d'un objet peut s'opérer indépendamment de sa mise en œuvre (structure interne) ;



Les attributs sont « encapsulés » dans l'objet et par conséquent protégeables contre des accès non autorisés

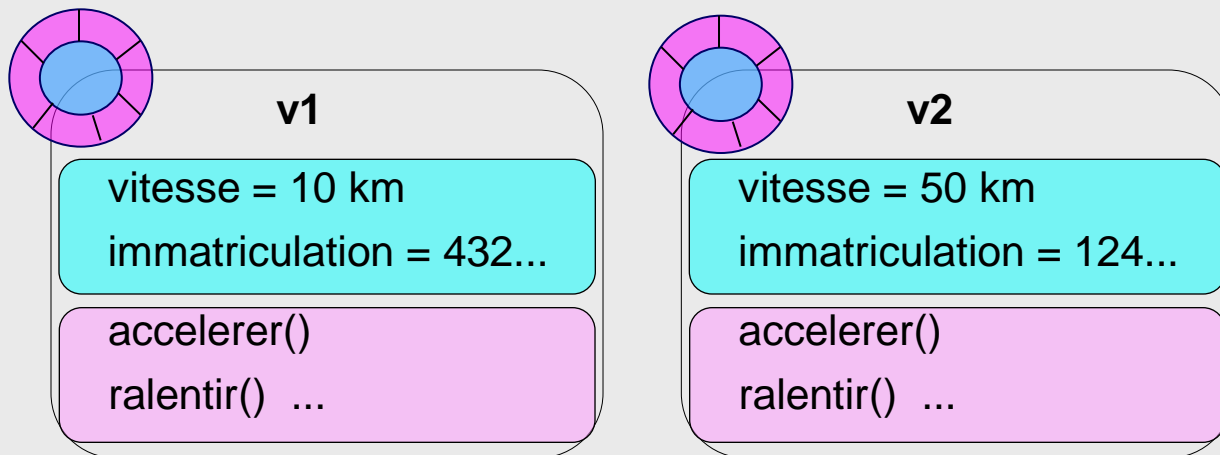
❖ Les classes (abstraction de données) :

- « *Une classe est un prototype (moule) qui définit la nature des attributs et les méthodes communs à tous les objets d'un certain type.* »

Voiture
immatriculation: String # vitesse: String
+ accelerer() + ralentir()

❖ Un objet est l'instance d'une classe

- Une classe permet par instantiation de construire des objets
 - exemples :
v1 et v2 sont deux instances (objets) de la classe Voiture



❖ 1er pas vers un diagramme de classes UML

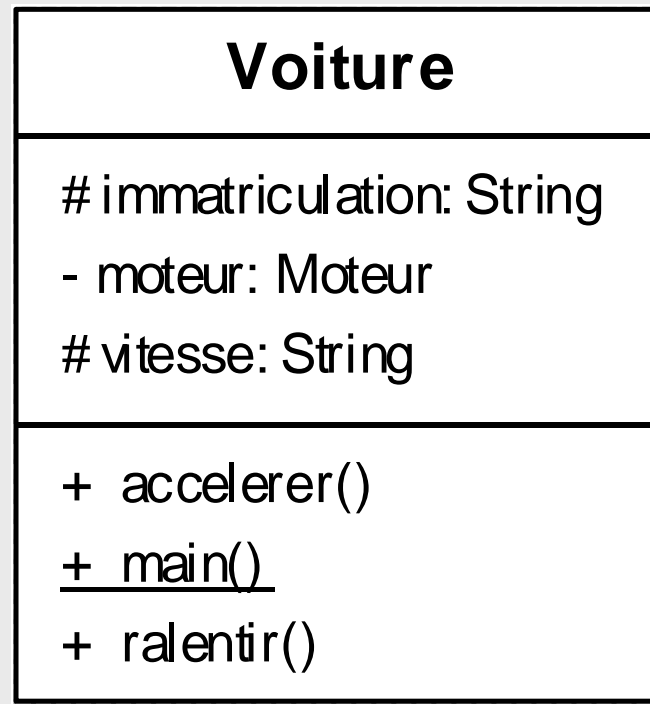
■ Notation

- contrôle d'accès - : privé + : publique # : protégé
- membre static : _____

Classe

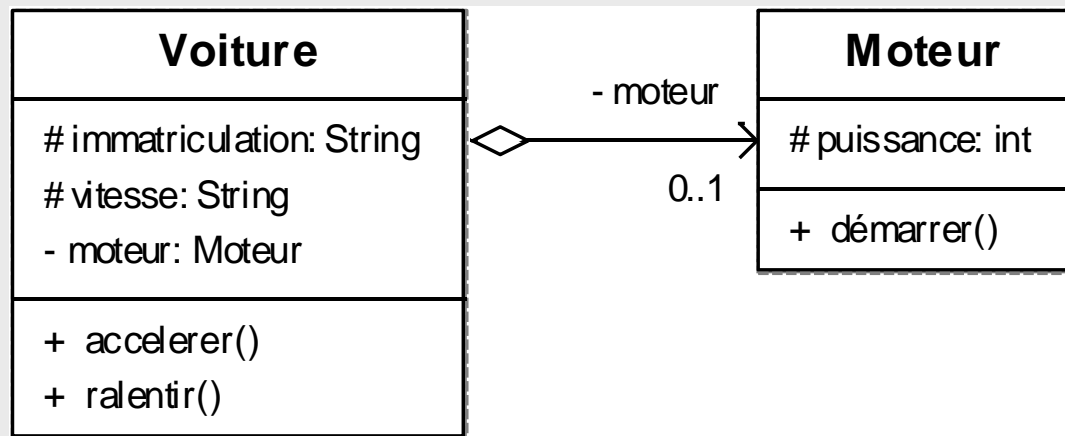
Attributs

Méthodes



❖ Association entre deux classes de type : ensemble / élément

- Exploitation d'un concept pour définir un autre concept
- Une voiture *possède* un moteur



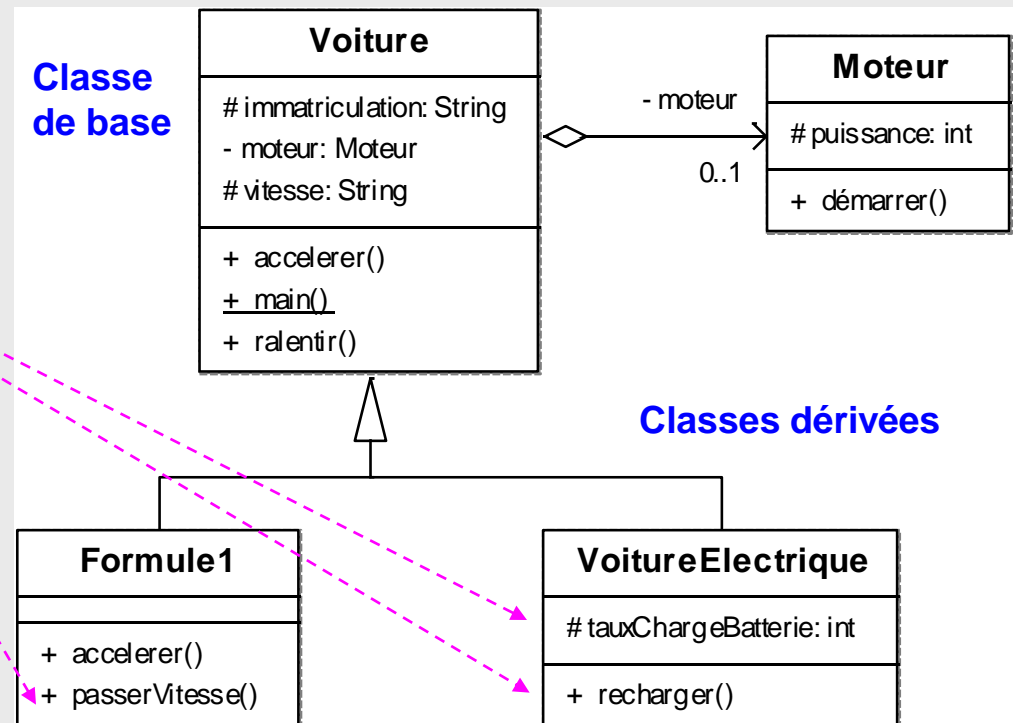
```
public class Voiture {
    protected String immatriculation;
    protected String vitesse;
    private Moteur moteur;
    public void accelerer() {System.out.println("accelererVoiture");};
    public void ralentir() {System.out.println("ralentirVoiture");};
}
```

❖ Notion d'héritage

« On peut raffiner la notion d'abstraction de données en utilisant la notion d'héritage »

- Une classe (*classe dérivée*) peut être spécialisée à partir d'une autre classe (*classe de base*)
- Chaque sous classe (ou classe dérivée) hérite de toutes les caractéristiques de la classe de base (attributs et méthodes)

- Une classe dérivée peut *ajouter* des attributs et des méthodes
- Une classe dérivée peut aussi *redéfinir* les méthodes de la classe de base.



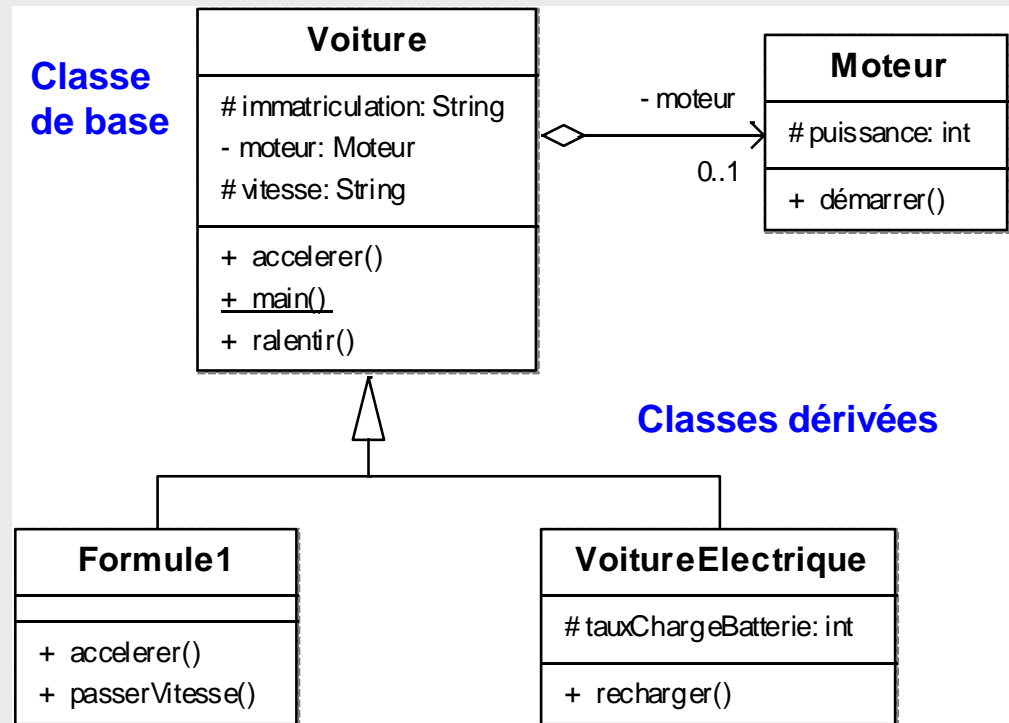
❖ Notion d'héritage et de Polymorphisme

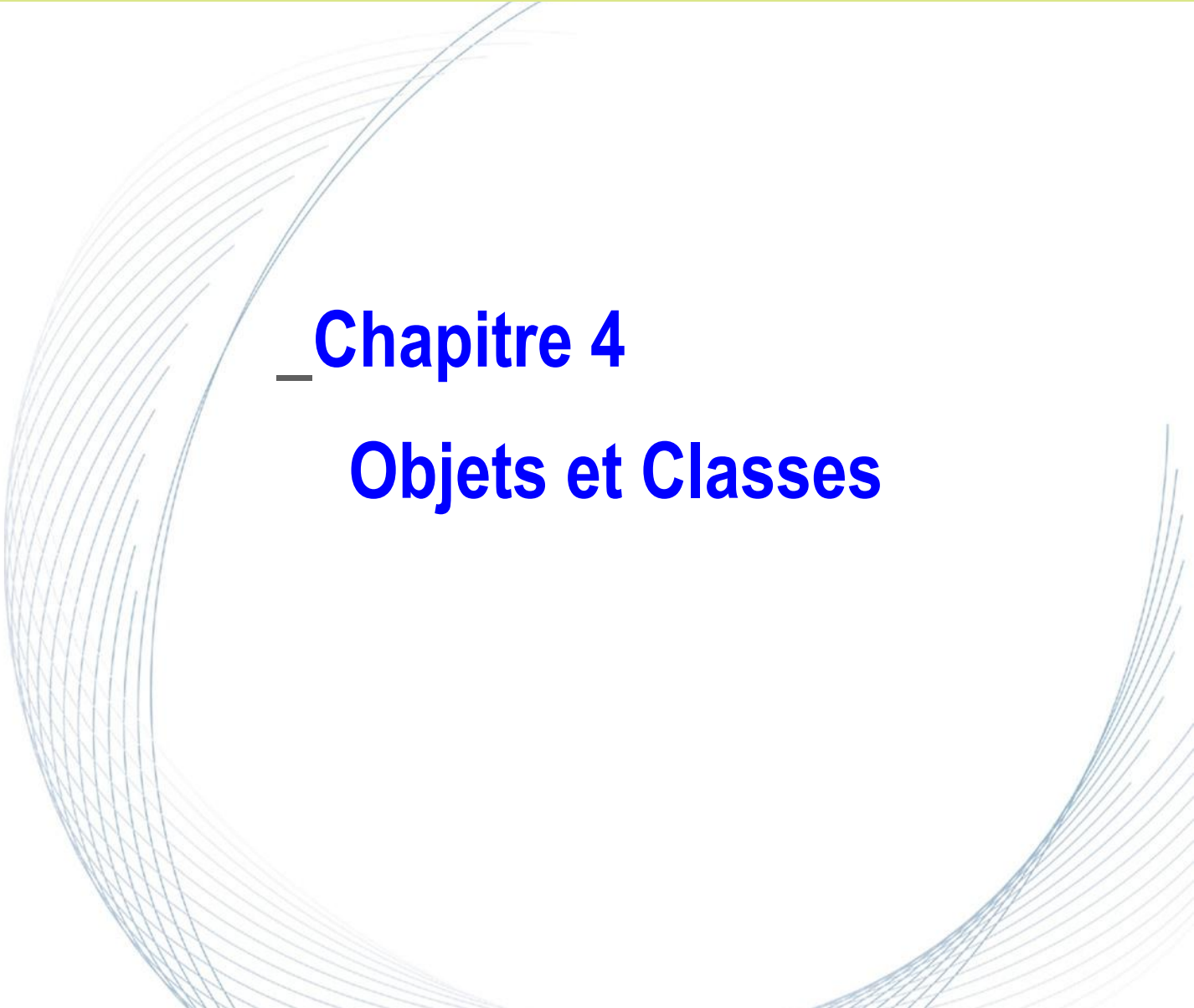
- Polymorphisme : mécanisme qui autorise l'appel d'une méthode qui à été redéfinie sur différents objets en provoquant des actions différentes selon la nature de l'objet.

```
public static void main(String [] args) {
    Formule1 f1=new Formule1();
    VoitureElectrique vElec=
        new VoitureElectrique();

    Voiture[] tabVoit={f1,vElec};
    for (int i=0; i< tabVoit.length; i++) {
        tabVoit[i].accelerer();
    }
}
```

accelererFormule1
accelererVoiture



A decorative graphic consisting of several concentric, overlapping circular arcs in a light blue color, creating a sense of motion or a stylized 'C' shape, positioned behind the chapter title.

Chapitre 4

Objets et Classes

❖ Tout objet doit être créé « NEW »

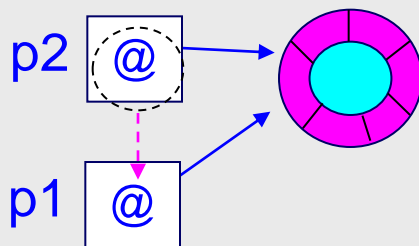
- Tous les objets sont alloués « new » dans le Tas (mémoire dynamique)

Point p = **new** Point(10,10) ; *// appel au constructeur de la classe*

❖ L'identificateur d'un objet correspond à une "référence" sur l'objet

Point p1 ; *// déclaration uniquement d'un identificateur ~pointeur*
 // - NULL si p1 est un champ d'une classe
 // - non initialisé si c'est une variable locale

Point p2 = new Point() ; *// java : créer explicitement un nouvel objet*



p1 = p2 ; *// p1 et p2 correspondent à 2 identificateurs du même objet*
 // ~ "pointent ou référencent le même objet"

Tous les éléments manipulés en Java sont des *objets* sauf les types dit *numériques* ou *primitifs*

❖ Les types numériques ou primitifs

■ Définition

- byte, boolean, char, short, int, long, float, double;

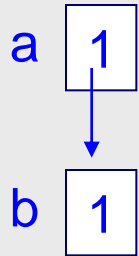
■ La manipulation de type numérique : // valeur

- On manipule des valeurs et non plus des identificateurs

```
int a = 1 ;
```

```
int b = a ;      // ok, manipulation standard b=1; a=1
```

```
b = 10 ;        // ok, manipulation standard b=10; a=1
```



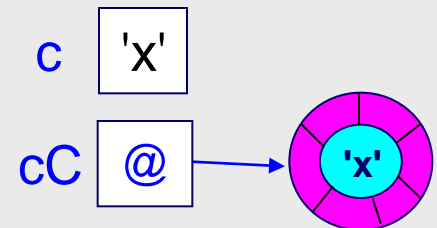
■ Les types numériques possèdent des classes associées (type objet)

```
char c = 'x';
```

```
Character cC = new Character('x');
```

```
Integer Inte=new Integer(10);
```

```
int i = Inte.intValue();
```



```

public class Point {
    private int _x;
    private int _y;

    /** fonction d'accès
     * abscisse du point ...
     */
    public int x() {
        return _x;
    }

    /** fonction d'accès
     * ordonnée du point
     */
    public int y() {
        return _y;
    }

    /** constructeur de point à partir
     * de ses coordonnées x et y
     */
    public Point(int x,int y) {
        _x = x;
        _y = y;
    }
}

```

```

/** déplacement du point par rapport aux coordonnées
 * du vecteur (x,y) passé en paramètre
 */

```

```

public void deplace(int dx,int dy) {
    _x += dx;
    _y += dy;
}

```

```

public static void main(String[] args) {
    Point p1=new Point(10,10);
    Point p2=new Point(20,20);
    System.out.println("p1 : (" +p1.x()+"," +p1.y()+")");
    System.out.println("p2 : (" +p2.x()+"," +p2.y()+")");
    p2.deplace(1,1);
    System.out.println("p2 apres déplacement de (1,1)
                        : (" +p2.x()+"," +p2.y()+")");
}

```

p1 : (10,10)

p2 : (20,20)

p2 apres déplacement de (1,1): (21,21)

❖ Dans la même classe : sur l'objet implicite (this)

```
class Point {  
    ...  
    public void initOrigine() {           // positionne le point courant à l'origine  
        _x=0; _y=0;                       // ⇔ this._x=0 ; this._y=0;  
    }  
}
```

❖ Dans une classe différente : sur un objet explicite

```
class Figure {  
    ...  
    private Point ref;                     // point de référence de la figure  
    ...  
    public void translate (int x, int y) {  
        ref.deplace(x,y);                 // manipulation explicite de l'objet réf  
    }                                     // par une méthode publique  
                                         // ⇔ this.ref.deplace(x,y);  
}
```

❖ Surcharge

- On surcharge une méthode en gardant le même nom et le même type de retour mais en changeant les paramètres (type et/ou nombre \Leftrightarrow signature)

```
class Point {  
    private int _x;  
    private int _y;  
  
    public int x()    { return _x; }  
    public int y()    { return _y; }  
  
    public void deplace (int dx, int dy) {  
        _x += dx; _y += dy; // ok  
    }  
  
    public void deplace (int d) {  
        _x += d; _y += d; // ok  
    }  
  
    ...  
}
```

Le compilateur choisit la méthode en fonction de sa signature

```
Point    p1 = new Point() ;  
p1.deplace(10 , 15 );  
p1.deplace(10 );
```



```

class Base {
    public void chg1(double a) {
        // les types primitifs sont passés par VALEUR
        a = 11.1;
    }    // pas d'effet en dehors de la procédure

    public void chg2(Point p) { // les objets sont modifiables
        // par l'intermédiaire de leur référence
        p.deplace(2,2);
    }    // effet en dehors de la procédure

    public static void main (String[] args) {
        double a=2.0;
        Base b1 = new Base();
        Point p1=new Point (10,10);

        Point p2=p1;
        b1.chg1(a);
        b1.chg2(p1);

        // a (valeur) ne sera pas modifié
        // Point p1 a été modifié (déplacement)
        // ce qui engendre la mise à jour de p2
    }
}

```

a 2.0

recopie
des valeurs

a 2.0

p @

recopie des références

p1 @

recopie des références

p2 @

❖ En java, tous les paramètres sont passés par VALEUR

```
class Base {  
    public void swap(double s) {  
        double temp = _val;    _val = s;    s = temp;  
    }    // s = temp;    pas d'effet en dehors de la procédure  
    double _val;  
}
```

❖ NB : les valeurs (identificateurs) sont assimilables à des pointeurs

```
class Base {  
    public void swap(Base b) {  
        double temp = _val; _val = b._val;    b._val = temp;  
        // b._val = temp; effet en dehors de la procédure  
        b = new Base();    // ok mais sans influence sur l'extérieur  
    }  
    double _val;  
}
```

Identificateur sur un objet de type base

- on peut modifier l'objet associé à l'identificateur passé en paramètre
- on ne peut cependant pas redéfinir les identificateurs pour l'extérieur !

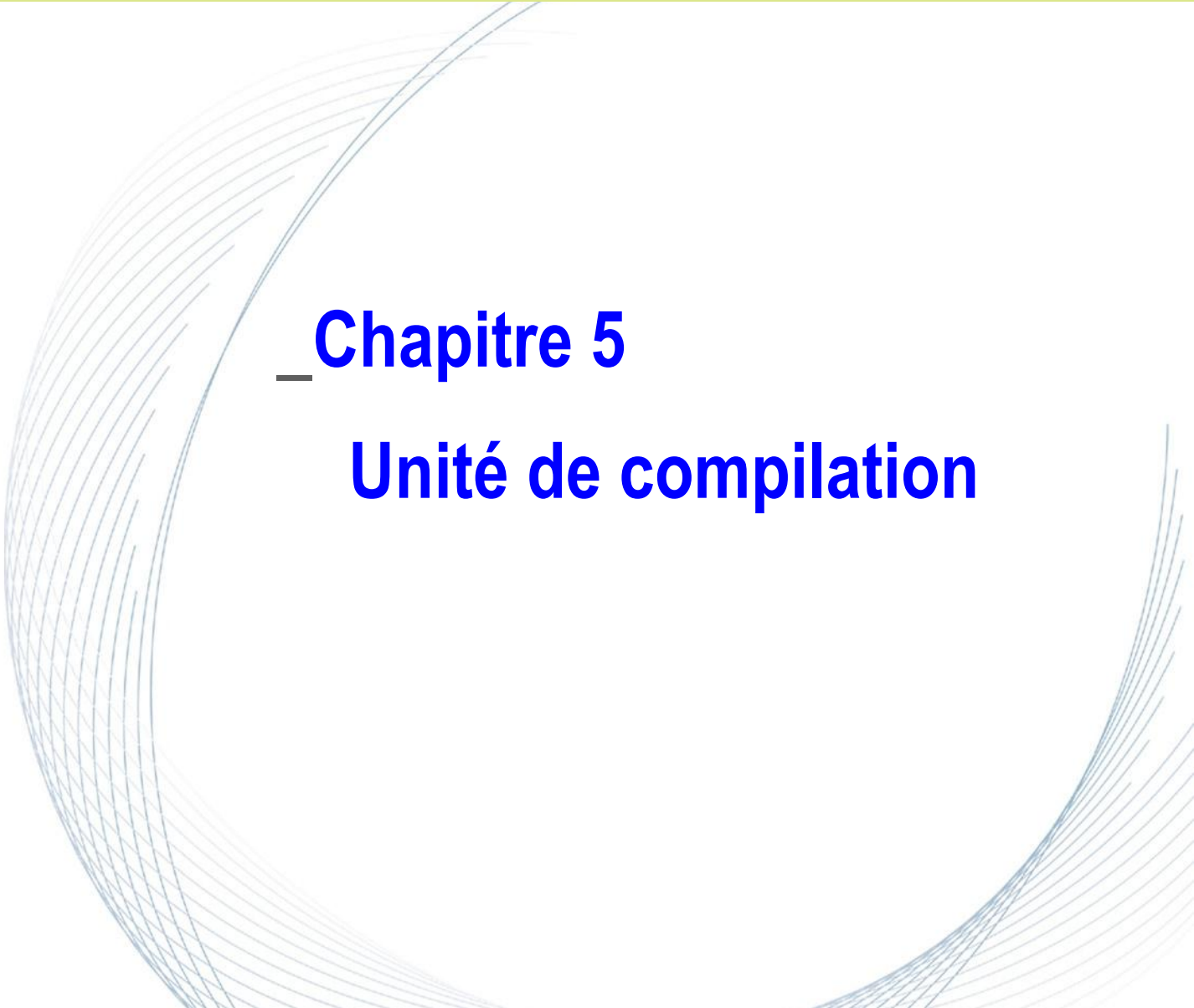
- ❖ **2 premiers niveaux de protection pour les membres d'une classe :**
 - **private :**
 - les membres privés ne sont pas accessibles à l'extérieur de la classe
 - **public :**
 - les membres publics sont accessibles partout où la classe est accessible

```
class Point {  
    private int _x;  
    private int _y;  
  
    public int x()    { return _x; }  
    public int y()    { return _y; }  
    public void deplace (int dx, int dy) {  
        _x += dx;  _y += dy; // ok  
    }  
    ...  
}
```

```
Point    p1 = new Point() ;  
Point    p2 = new Point(10,10);  
p1._x =10; // erreur  
p1.deplace(p2._x , p2._y ) ; // erreur  
p1.deplace(p2.x() , p2.y() ) ; // ok
```

- Les attributs publics et privés sont spécifiés pour chaque membre
- La définition des fonctions membres est effectuée dans la définition de la classe
- Les données membres sont initialisées par défaut
- Remarques : il n'y a pas de point virgule à la fin de la définition de classe

```
class Point {  
    public Point (int x, int y) {  
        _x = x;  
        _y = y;  
    }  
    public void deplace (int dx, int dy){  
        _x += dx;  
        _y += dy;  
    }  
    public int x() { return _x; }  
    public int y() { return _y; }  
  
    private int _x;  
    private int _y;  
}
```

A decorative graphic consisting of several concentric, overlapping circular arcs in a light blue color, creating a sense of motion or a stylized 'C' shape, positioned behind the chapter title.

Chapitre 5

Unité de compilation

❖ **Unité de compilation ⇔ Fichier : *MaClasse.java***

- une seule classe peut être publique dans un fichier
 - = interface externe
- cette classe possède alors obligatoirement le même nom que le fichier
 - `public class MaClasse { }`

❖ **Compilation en Java : `javac`**

- une unité de compilation `.java` → des fichiers *bytecode*
(un `.class` par classe)

```
javac MaClasse.java
```

❖ **Interprétation : `java`**

- fichiers **bytecode** : instructions destinées à la machine virtuelle

```
java MaClasse
```

Fichier : MaClasse.java

```
class Auxiliaire {  
    ...  
    public static void main (String[ ] args) {  
        Auxiliaire x = new Auxiliaire ();  
        ...  
    }  
}
```

```
public class MaClasse {  
    ...  
    public static void main (String[ ] args) {  
        MaClasse x = new MaClasse();  
        ...  
    }  
}
```

Test unitaire des classes

```
java MaClasse           //   appel à MaClasse.main()  
java Auxiliaire         //   appel à Auxiliaire.main() (test unitaire!)
```


A decorative graphic consisting of several concentric, overlapping circular arcs in a light blue color, creating a sense of motion or a stylized 'C' shape, positioned behind the chapter title.

Chapitre 6

Packages

❖ Le code java est partitionné en différents *Packages*

- ex : java.util // regroupe différentes classes utilitaires
 java.applet // regroupe les classes utiles à la programmation
 d'applet (programme java pour Internet)

❖ Accès aux classes des Packages

- Pour accéder aux classes des packages il faut accéder à leur nom

- soit en utilisant le nom étendu

```
java.util.Vector v = new java.util.Vector( );
```

- soit en important le nommage de la classe du package

```
import java.util.Vector;
```

```
Vector v = new Vector ( );
```

- soit en important l'espace de nommage du package entier

```
import java.util.*;
```

```
Vector v = new Vector ( );
```

❖ Définition d'un package // Répertoire

- Si aucun Package n'est mentionné, toutes les classes définies dans le même répertoire appartiennent à un même package implicite
- Package explicite / ex : fichier MaClass.java

```
package    malibrairie;      // première instruction du fichier
public class MaClass { ... } // même nom que le fichier
```

❖ Visibilité des classes

- La classe "Public" (unique / fichier) : accessible à l'extérieur du package
- Les autres classes ne sont utilisables que dans le package

Package P1

```
package    P1;
public class CA {
    CB ob; // ok accès dans
           // le même package
}
```

```
package    P1;
class CB {
    CA oA; // ok accès
           // classe publique
}
```

Package P2

```
package    P2;
import P1.*;
public class CC {
    CA obA; // ok accès
           // classe publique
    CB obB; // ERREUR !
           // package différent
}
```

Chapitre 7

Egalités d'objets (1ère approche)

❖ Opérateur ==

- Teste l'égalité des références (identificateurs) des objets
- Ne considère jamais l'égalité du contenu d'un objet

```
String s = "coucou";
```

```
if (s == "coucou") ...           // comparaison des références !!!
```

❖ Opérateur equals

- L'opérateur *equals* par défaut (de la classe de base *Object*) opère une comparaison sur les références.
- Il a été redéfini pour beaucoup de classe de la bibliothèque

```
if (s.equals("coucou")) ...      // ok en Java
```

❖ Ecriture d'un opérateur equals

- Normalement : redéfinir l'opérateur equals de la classe « *Object* »
 - Cf. redéfinition/equals
- En attendant :
possibilité de le surcharger

```
Class Point {  
    public boolean equals (Point p)  
        // ici surcharge <> redéfinition  
        { return _x== p._x && _y == p._y ; }  
    ...  
}
```

A decorative graphic consisting of several concentric, overlapping circular arcs in a light blue color, creating a sense of motion or a stylized 'C' shape, positioned behind the chapter title.

Chapitre 8

Membre statique

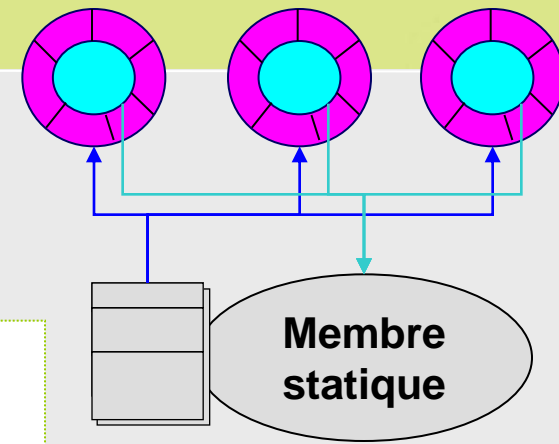
- Membres d'une classe (attributs ou méthodes) partagés par l'ensemble des instances de la classe
- Invocation sur une classe et non sur un objet

```
public class Point {
    private int _x;
    private int _y;
    private static int nbPoint=0;

    public static void nbPointC() {
        System.out.println("nb d'objets Point créés : " + nbPoint);
    }
    ...
    public Point(int x,int y) {
        _x = x; _y = y;
        nbPoint++;
    }
    public Point() { // constructeur par défaut
        _x = 0 ; _y = 0;
        nbPoint++;
    }
    ...
}
```

```
public static void main(String args[]) {
    Point p1= new Point(10,10);
    Point p2= new Point();
    Point p3= p2;
    Point.nbPointC();
}
```

nb d'objets Point créés : 2



A decorative graphic consisting of several concentric, overlapping circular arcs in a light blue color, creating a sense of motion or a stylized 'C' shape, positioned behind the chapter title.

Chapitre 9

Donnée constante

Les constantes sont des composants de classe déclarés : *final*

❖ **La valeur d'une constante n'est attribuée qu'une fois au cours de l'exécution :**

- soit à l'endroit de sa déclaration
- soit dans les constructeurs de la classe où elle est définie
... permet d'initialiser la constante en fonction de l'objet instancié ...

```
class A {  
    public static final int UN=1;    // constante et statique  
                                    // initialisée une fois pour toute  
  
    public final int Xorig;  
    public final A copain;          // l'objet sera modifiable mais pas sa référence  
    ...  
    A(int x, A c) {  
        Xorig=x;                   // initialisé une fois pour toute / objet créé  
        copain=c;                  // idem au niveau de la référence  
        // NB: l'objet référencé peut être modifié  
    }  
    ...  
}
```

A decorative background graphic consisting of several concentric, overlapping circular arcs in a light blue color, creating a sense of motion or a stylized 'C' shape.

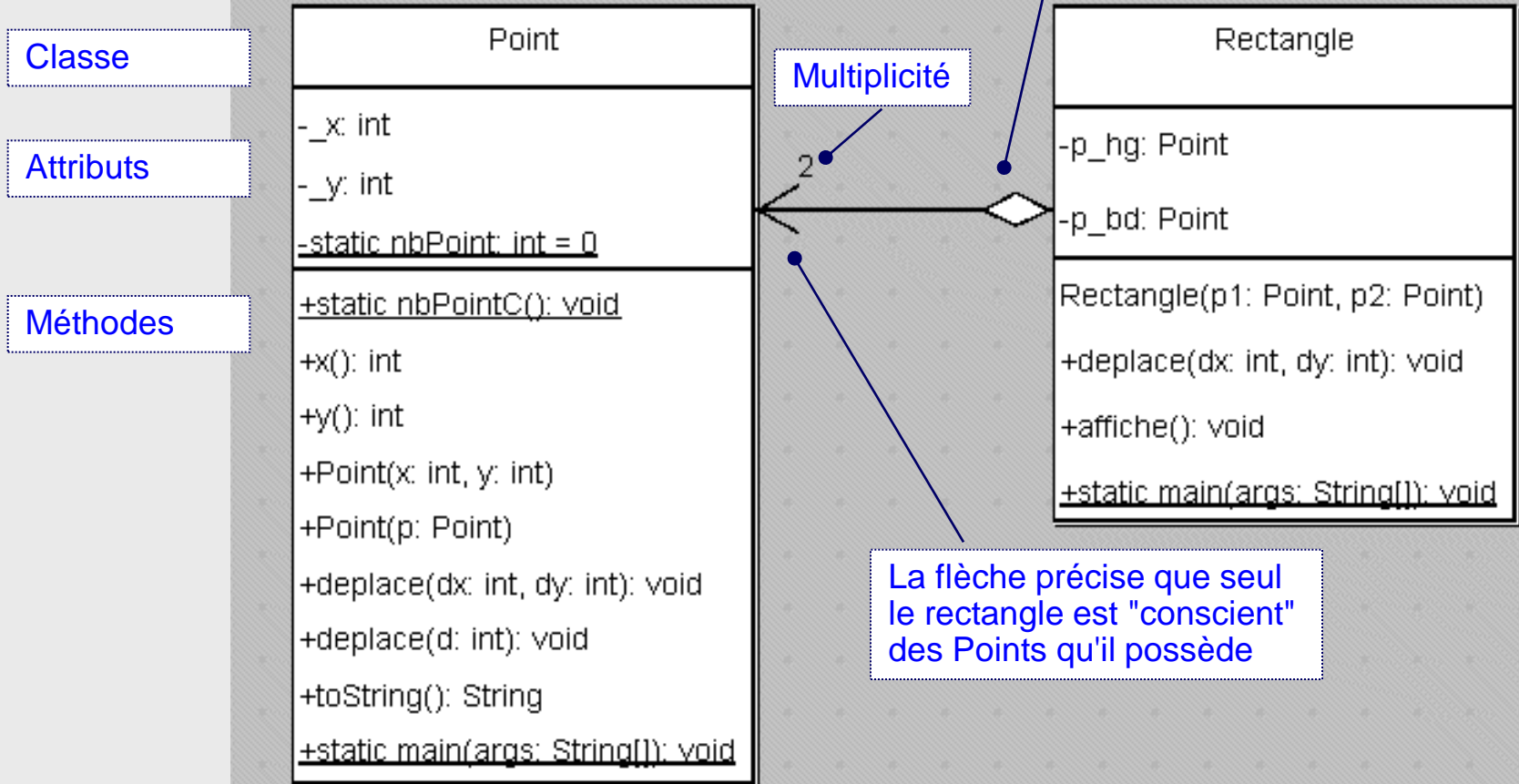
Chapitre 10

Agrégation

❖ 1er pas vers un diagramme de classes UML

- **Notation (membres) :** - : privé + : publique _ : static
- **Association :** connexion sémantique bidirectionnelle entre deux classes
- **Agrégation (faible) :** association non symétrique qui exprime une relation de type : ensemble / élément

Association de type agrégation " ensemble / élément " (losange blanc) :
Cad : un rectangle contient 2 Points"



- ❖ **Agrégation : abstraction et réutilisation d'un concept**
 - La classe Rectangle utilise la classe point dans sa définition

```
public class Rectangle {
    private Point p_hg;
    private Point p_bd;

    Rectangle(Point p1, Point p2) {
        p_hg = p1 ;
        p_bd = p2 ;
    }

    public void deplace(int dx, int dy) {
        p_hg.deplace(dx,dy);
        p_bd.deplace(dx,dy);
    }

    public void affiche() {
        System.out.println("Rectangle : (" +
            p_hg.x()+" "+p_hg.y() +
            ") (" + p_bd.x()+" "+p_bd.y()+"");
    } ...
}
```

```
public static void main(String args[]) {
    Point p1=new Point(5,5);
    Rectangle r1=new Rectangle (p1,new Point(10,10));
    r1.affiche();
    r1.deplace(1,1);
    System.out.println("deplacement");
    r1.affiche();
    System.out.println("Point : " + p1.x()+" "+p1.y()); // NB
}
```

```
Rectangle : (5,5) (10,10)
deplacement
Rectangle : (6,6) (11,11)
Point : 6,6
```

A decorative graphic consisting of several concentric, overlapping circular arcs in a light blue color, creating a sense of motion or a stylized 'C' shape, positioned in the background of the slide.

— Chapitre 11

Recopie

❖ Problème : création d'un carré à partir d'un point

```

public class Rectangle {
    private Point p_hg;
    private Point p_bd;

    Rectangle (Point p1, Point p2) {
        p_hg = p1 ; p_bd = p2 ; // ici pas de copie
    }

    public void affiche(){
        System.out.println("Rectangle : (" +
            p_hg.x()+"," + p_hg.y()+
            ") (" + p_bd.x()+"," + p_bd.y()+")");
    }

    public static void main(String args[]) {
        Point p1=new Point(5,5);
        Rectangle r1 = p1.faireRectangle(10);
        r1.affiche();
    }
}

```

Rectangle : (5,5) (5,5)

```

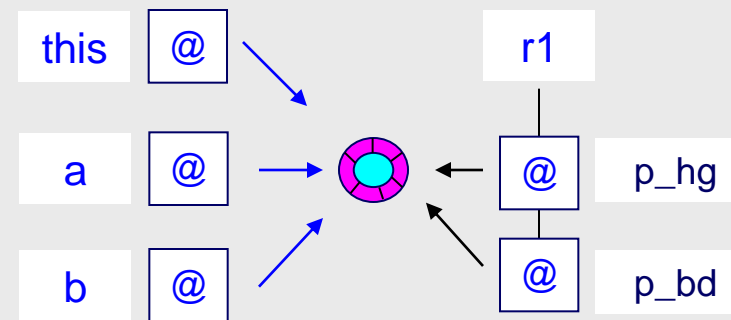
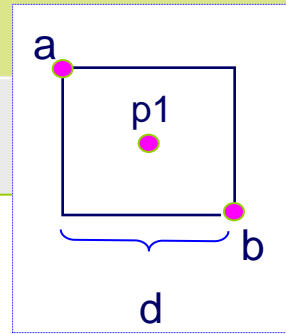
public class Point
{
    ...

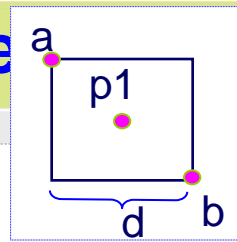
```

```

    public Rectangle faireRectangle (int d) {
        Point a= this ; // pas de copie
        a.deplace(-d/2,-d/2);
        Point b= this ; // pas de copie
        b.deplace(d/2,d/2);
        Rectangle r=new Rectangle(a,b);
        return r;
    }
}

```





❖ Solution : recopie des points

- Avec un "constructeur par recopie"
- Avec la notion de clonage (cf. après)

```
public class Rectangle { // idem
    private Point p_hg;
    private Point p_bd;

    Rectangle (Point p1, Point p2) {
        p_hg = p1 ; p_bd = p2 ;
    } // ici pas de recopie

    ...

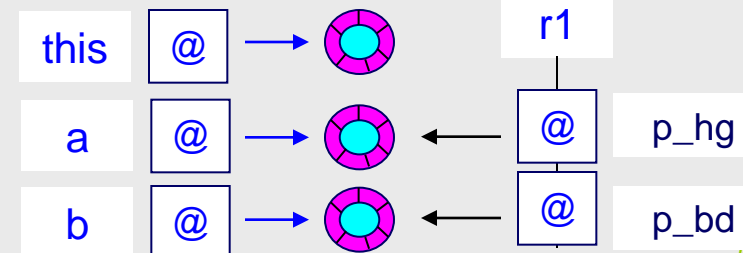
    public static void main(String args[]) {
        Point p1=new Point(5,5);
        Rectangle r1 = p1.faireRectangle(10);
        r1.affiche();
    }
}
```

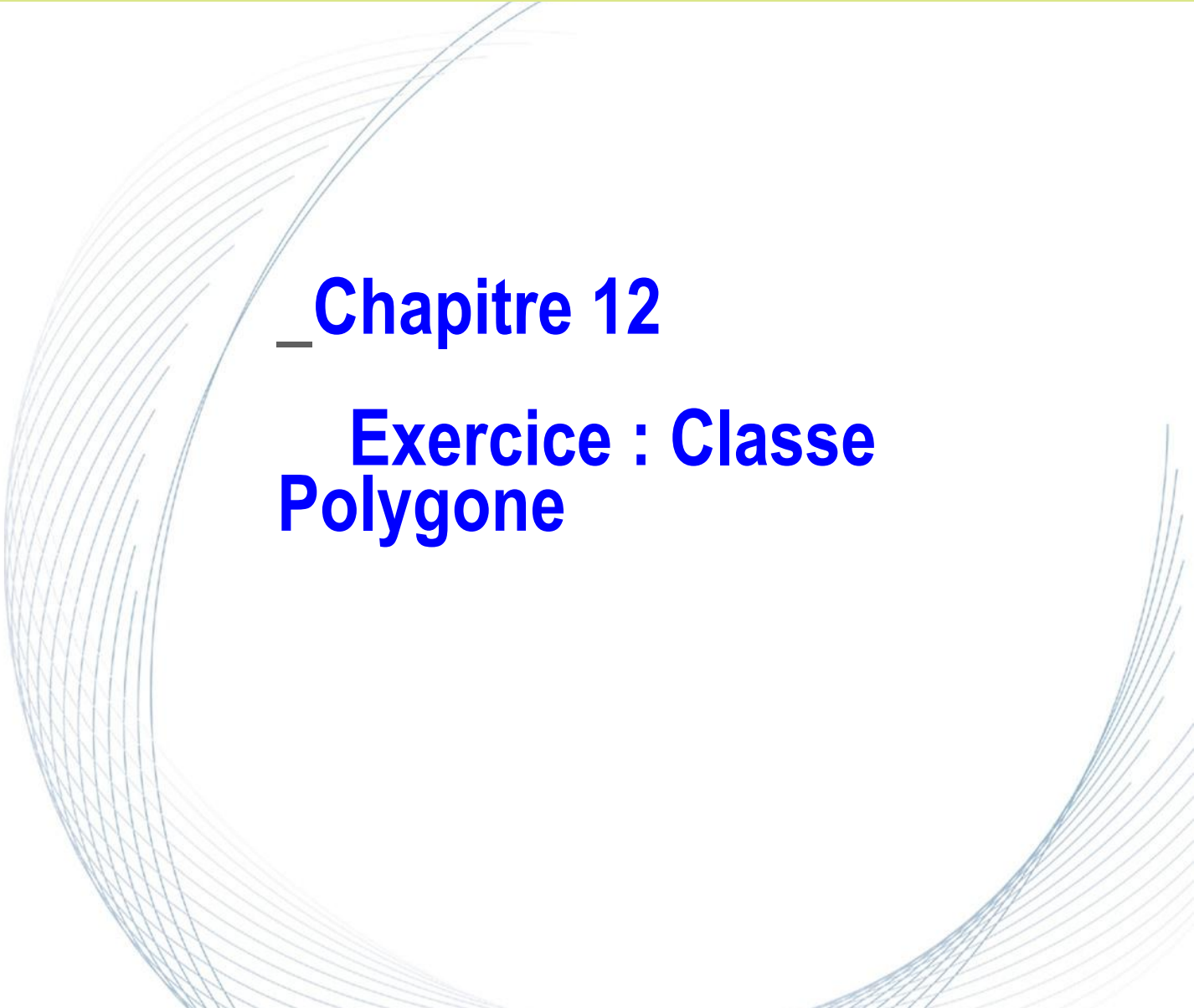
Rectangle : (0,0) (10,10)

```
public class Point {
    /** constructeur par recopie
     * permet de construire un point à partir
     * d'un autre point passé en paramètre
     */

    public Point(Point p) {
        _x = p.x();
        _y = p.y();
        nbPoint++;
    } ...

    public Rectangle faireRectangle (int d) {
        Point a= new Point(this) ; // recopie
        a.deplace(-d/2,-d/2);
        Point b= new Point(this) ; // recopie
        b.deplace(d/2,d/2);
        Rectangle r=new Rectangle(a,b);
        return r; }
}
```



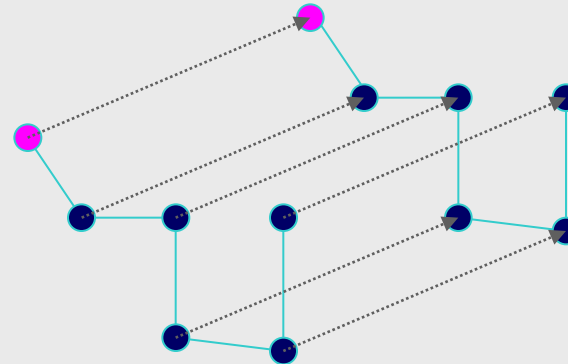
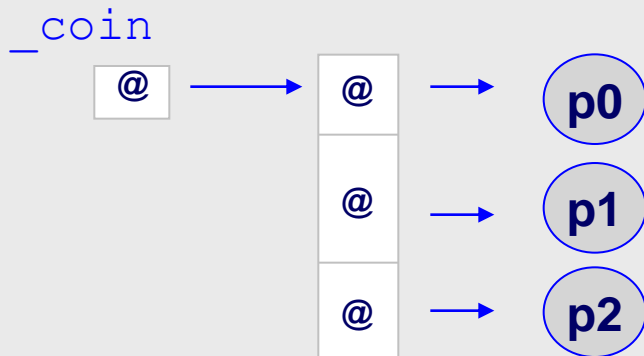
A decorative graphic consisting of several concentric, overlapping circular arcs in a light blue color, creating a sense of motion or a stylized 'C' shape, positioned behind the main text.

Chapitre 12

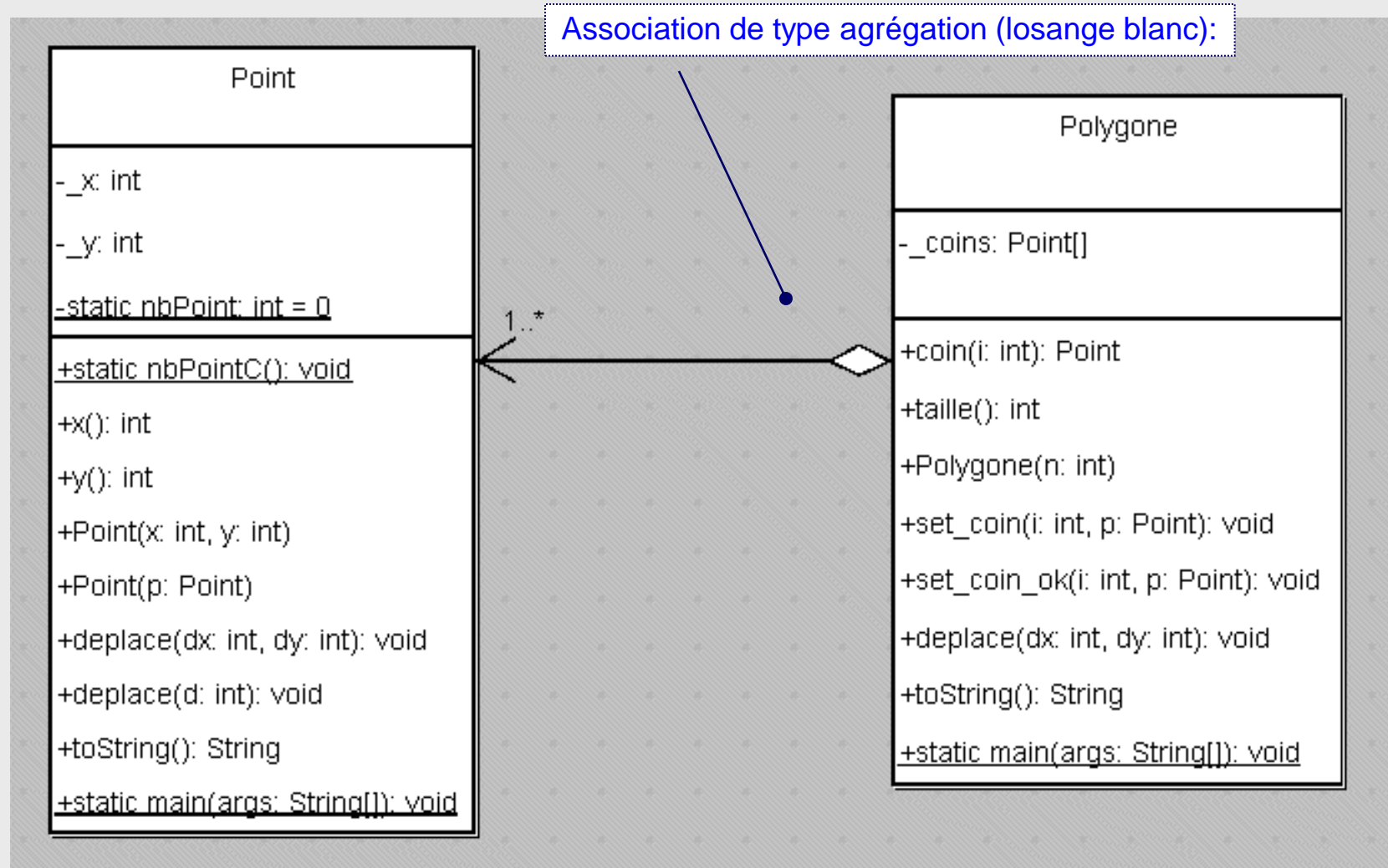
Exercice : Classe Polygone

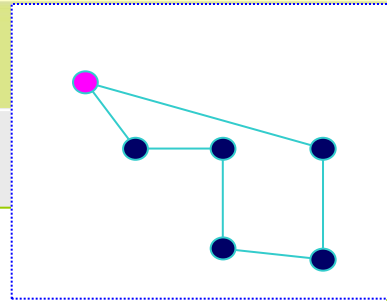
❖ Définition : classe polygone :

- un polygone sera représenté par un tableau d'objets Point
- la taille du polygone sera définie lors de sa construction
- les points du polygone sont ensuite définis un par un à l'aide d'une méthode
- une méthode permettra de déplacer (translater) un polygone



❖ Diagramme de classes





```
public class Polygone
```

```
{
    private Point[] _coins;
    public Point coin(int i){ return _coins[i];}
    public int taille() { return _coins.length;}
    public Polygone(int n) {
        _coins = new Point[n];
    }

    /** ajout d'un point dans le polygone
     * sans recopie */
    public void set_coin(int i, Point p) {
        if(0 <= i && i < _coins.length)
            _coins[i] = p;
    }
    public void deplace(int dx, int dy){
        for(int i = 0; i < _coins.length; i++)
            _coins[i].deplace(dx, dy);
    }
}
```

```
public String toString() {
    String S="";
    for (int i=0; i<taille(); i++){
        S+= "("+ coin(i)+" ";
    }
    return S;
}
```

```
public static void main(String args[]) {
    Point p0=new Point(10,10);
    Point p1=new Point(20,20);
    System.out.println("p0 :"+p0);
    System.out.println("p1 :"+p1);
    Polygone Poly=new Polygone(3);
    Poly.set_coin(0,p0);
    Poly.set_coin(1,p1);
    Poly.set_coin(2,p0);

    System.out.println(Poly);
}
```

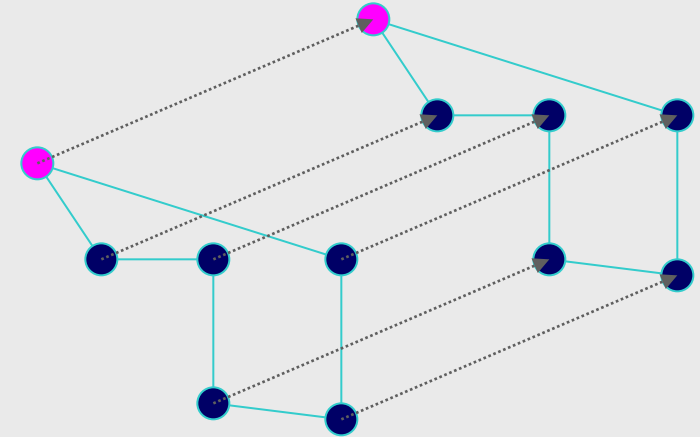
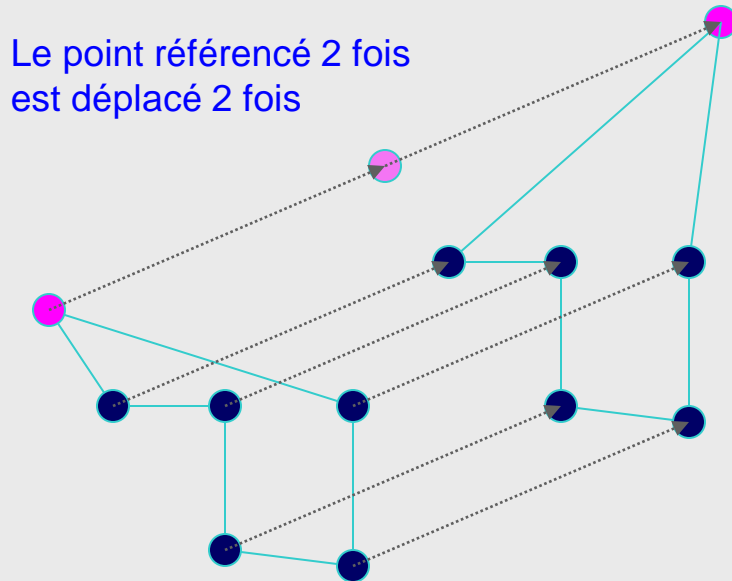
p0 :(10,10)

p1 :(20,20)

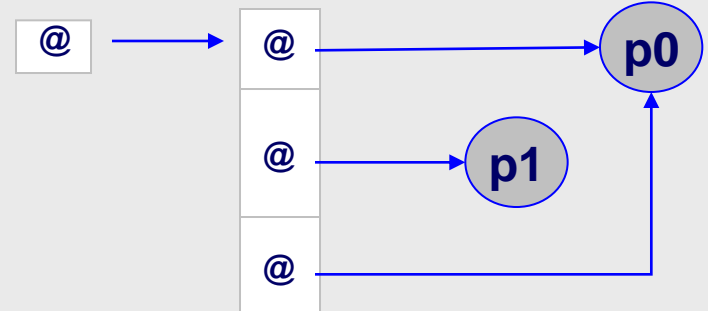
((10,10)) ((20,20)) ((10,10))

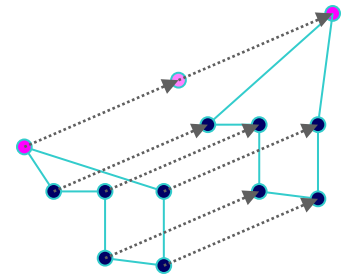
❖ On peut définir un polygone fermé en insérant en début et en fin du Polygone le même Point

- Que se passe t'il si l'on déplace ce polygone
- Proposer une solution générale pour remédier à ce Problème
- Modifier en conséquent les classes Point et Polygone



`_coin`





```
public class Polygone
```

```
{
```

```
    private Point[] _coins;
```

```
    public Point coin(int i){ return _coins[i];}
```

```
    public int taille() { return _coins.length;}
```

```
    public Polygone(int n) {
        _coins = new Point[n];
    }
```

```
    /** ajout d'un point dans le polygone
        sans recopie
```

```
    */
```

```
    public void set_coin(int i, Point p) {
        if(0 <= i && i < _coins.length)
            _coins[i] = p;
    } ...
```

```
p0 :(10,10)
```

```
p1 :(20,20)
```

```
((10,10)) ((20,20)) ((10,10))
```

```
-- apres déplacement de (1,1) de Poly --
```

```
((12,12)) ((21,21)) ((12,12))
```

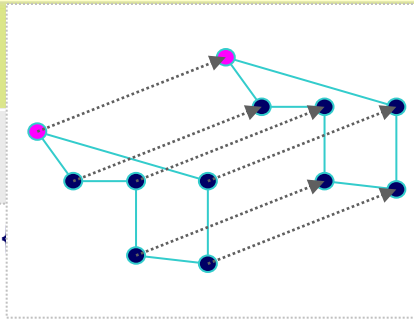
```
    public String toString() {
        String S="";
        for (int i=0; i<taille(); i++){
            S+= "("+ coin(i)+" ";
        }
        return S;
    }
```

```
    public static void main(String args[]) {
        Point p0=new Point(10,10);
        Point p1=new Point(20,20);
        System.out.println("p0 :"+p0);
        System.out.println("p1 :"+p1);
        Polygone Poly=new Polygone(3);
        Poly.set_coin(0,p0);
        Poly.set_coin(1,p1);
        Poly.set_coin(2,p0);
```

```
        System.out.println(Poly);
        System.out.println("-- apres déplacement
                               de (1,1) de Poly --");
```

```
        Poly.deplace(1,1);
        System.out.println(Poly);
```

```
    }
```



```
public class Polygone
```

```
{
```

```
    private Point[] _coins;
```

```
    public Point coin(int i) { return _coins[i];}
```

```
    public int taille() { return _coins.length;}
```

```
    public Polygone(int n) {
        _coins = new Point[n];
    }
```

```
    /** Ajout d'un nouveau point :
     *  ce point est une recopie du point
     *  passé en paramètre. Utilisation du
     *  constructeur par recopie.
     */
```

```
    public void set_coin_ok(int i, Point p) {
        if(0 <= i && i < _coins.length)
            _coins[i] = new Point(p);
    }
    ...
```

```
    public String toString() {
        String S="";
        for (int i=0; i<taille(); i++){
            S+= "("+ coin(i)+ " ";
        }
        return S;
    }
```

```
    public static void main(String args[]) {
        p0=new Point(10,10);
        p1=new Point(20,20);
        System.out.println("p0 :"+p0);
        System.out.println("p1 :"+p1);
        Polygone Poly2=new Polygone(3);
        Poly2.set_coin_ok(0,p0);
        Poly2.set_coin_ok(1,p1);
        Poly2.set_coin_ok(2,p0);
```

```
        System.out.println(Poly2);
        Poly2.deplace(1,1);
```

```
        System.out.println("-- apres deplacement
                               de (1,1) de Poly2 --");
        System.out.println(Poly2);}
```

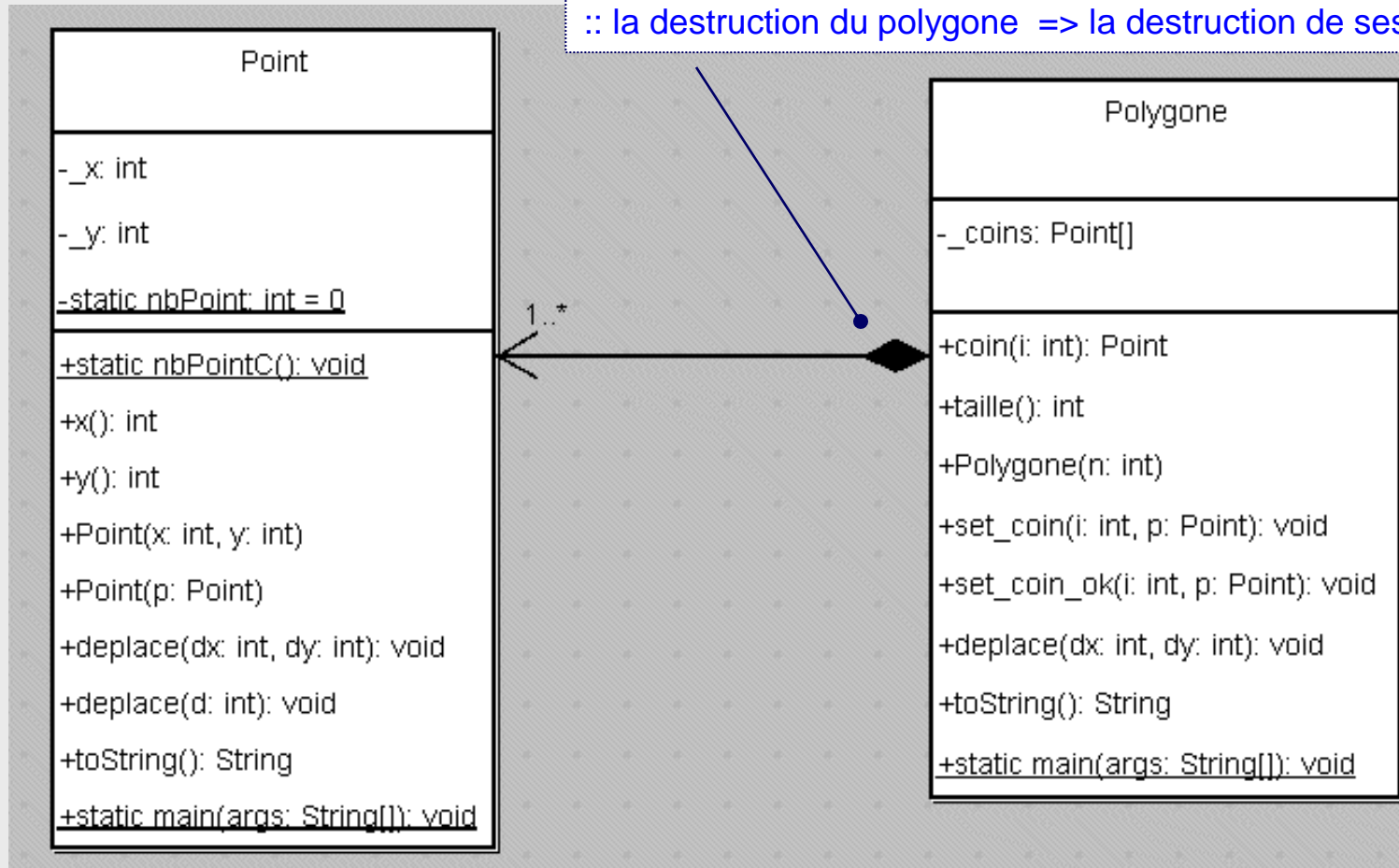
```
p0 :(10,10)
p1 :(20,20)
((10,10)) ((20,20)) ((10,10))
-- apres deplacement de (1,1) de Poly2 --
((11,11)) ((21,21)) ((11,11))
```

❖ Diagramme de Classes

- **Association** : connexion sémantique bidirectionnelle entre deux classes
- **Agrégation faible** : association non symétrique qui exprime une relation de type : ensemble / élément
- **Agrégation forte (Composition)** : le polygone possède des Points qui lui sont propres

Agrégation forte : composition (losange noir) :

:: la destruction du polygone => la destruction de ses Points



A decorative background graphic consisting of several concentric, overlapping circular arcs in a light blue color, creating a sense of motion or a stylized 'C' shape.

Chapitre 13

Gestion mémoire

La portée des variables est liée aux { ... } = durée de vie

❖ Portée au niveau des types primitifs

```
class Test {
    void methodeA {
        int a, b;
        a=1; b=2;
        ...
    }
    void methodeB {
        int a, b;
        ...
        if (a < b) {
            int f;
            f = a ;...
        }
        f = 0;
    }
}
```

```
{ int a=12;
  {
    int a= 10;    // erreur en JAVA
  }
}
```

// ok, on n'est pas dans le même bloc

// les variables du bloc supérieur
// sont connues dans les blocs inférieurs

// la réciproque est fausse
// erreur, f n'est pas connue dans ce bloc

❖ Portée des objets java

```
{
    String s = new String("toto");
}
```

// l'identificateur est hors de portée **et** on ne peut plus accéder à l'objet
// il sera alors détruit automatiquement par le « ramasse miette »

« Assure une gestion plus sûre de la mémoire »

❖ **Durée de vie : le ramasse miette ou « garbage collector »**

- **Java dispose d'un système de récupération de mémoire automatique quand il « estime » que l'espace occupé par un objet peut être restitué (\Leftrightarrow plus de référence sur cet objet).**

- attention un objet peut être référencé à plusieurs endroits et tant qu'il existe une référence sur un objet il ne sera pas détruit

```
String[] T= new String[10];
```

```
{  
    String s ="toto";  
    T[0] = s;  
}
```

```
// la référence s n'existe plus  
// mais la chaîne "toto" est accessible par T[0]  
// donc la mémoire n'est pas libérée
```

- **Le « ramasse miette » fonctionne en arrière plan par défaut. La destruction est donc asynchrone (gérée par un thread)**
- **La récupération mémoire peut aussi être invoquée explicitement par le programmeur : `System.gc()`**

❖ Méthode finalize()

- Méthode appelée par le ramasse-miettes avant la destruction d'une instance de la mémoire
- Permet de libérer les ressources (fichier, socket, etc.)
- Elle peut être directement appelée

❖ Appel implicite possible si

- l'instance n'est plus référencée
- et que le ramasse-miettes est entré en action

❖ Attention

- **On ne peut pas prédire quand le ramasse-miettes va entrer en action !**
 - Appel explicite possible (en tâche de fond) avec : `System.gc()`
- **La méthode `System.runFinalization()`**
 - Lance le ramasse-miettes et attend que les méthodes `finalize()` soient invoquées avant de rendre la main
- **La fin d'un programme n'implique pas forcément l'appel des `finalize()` !**
 - Pour forcer l'appel à tous les `finalize()`: `System.runFinalization(true)`

```
Class UneClasse {  
    private File fichTemp= new File("ftemp");  
    private FileOutputStream flotfichTemp;  
    // ...  
    public void finalize() throws IOException {  
        flotFichTemp.close() // fermeture  
        fichTemp.delete(); // efface le fichier  
    }  
}
```

gestion d'un fichier temporaire

A decorative background graphic consisting of several concentric, overlapping circular arcs in a light blue color, creating a sense of motion or a stylized 'C' shape.

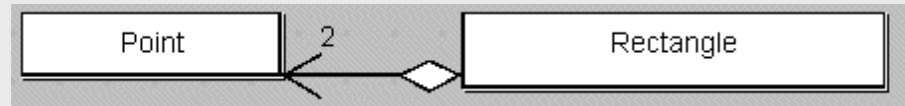
— Chapitre 14

Héritage

❖ La notion d'abstraction, de réutilisation se traduit par :

■ La notion de classe

- "moule" de construction permettant de créer des objets d'un certain type

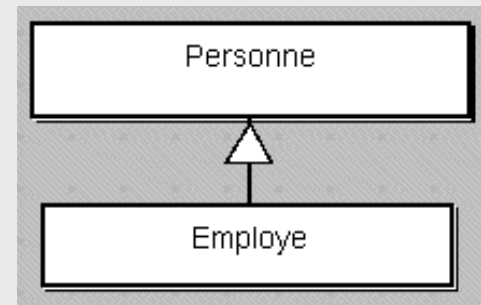


■ La notion d'agrégation

- Exemple : la classe Rectangle utilise le concept de Point dans sa définition
- => définition avec un plus haut niveau d'abstraction

■ La notion d'héritage

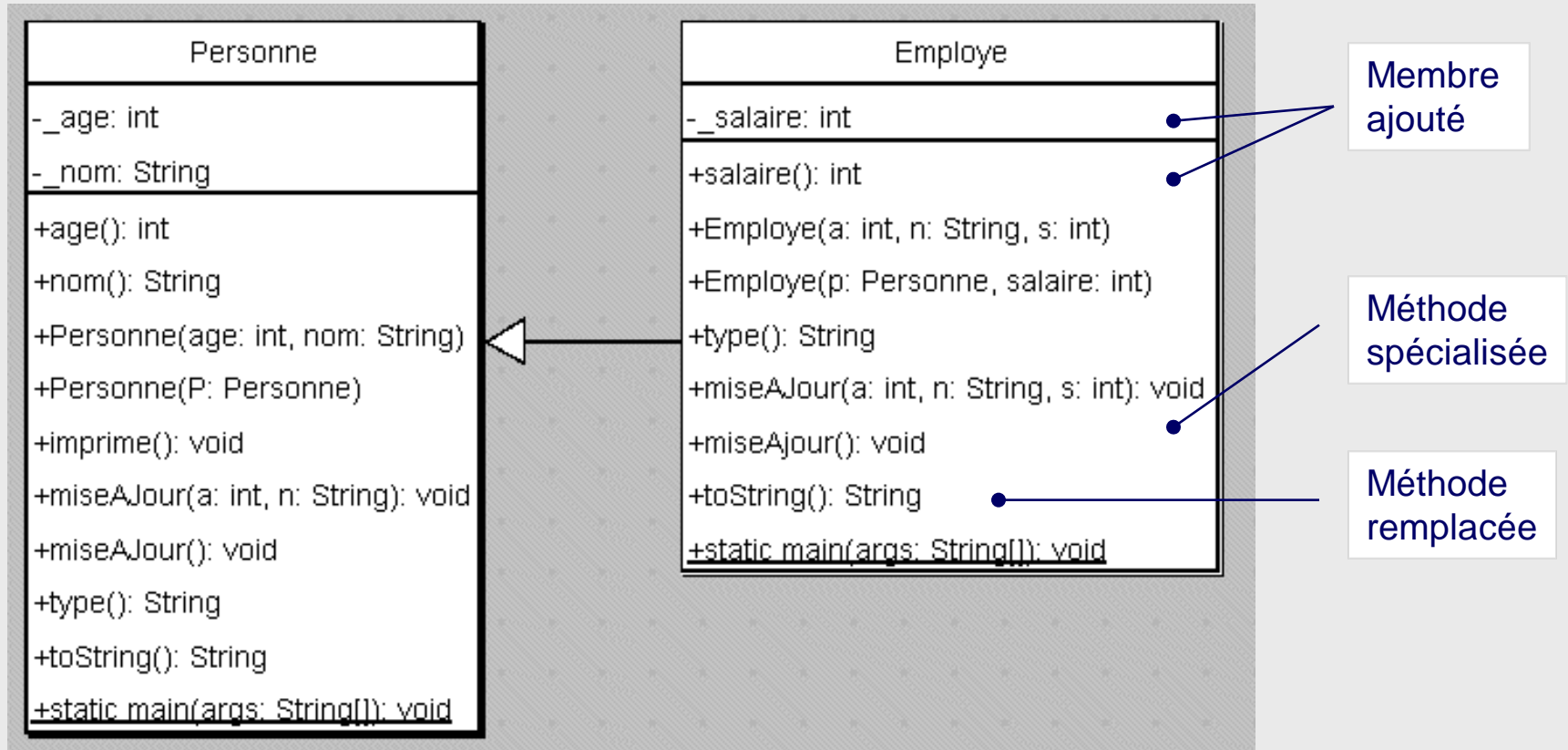
- Spécialisation d'un concept
- Une classe dérivée
 - adopte,
 - spécialise
 - et/ou enrichit



les membres (structure, méthode) d'une classe de base

- Hiérarchisation des concepts

❖ Diagramme de classes



❖ La classe Personne

```
public class Personne {  
    private int _age;  
    private String _nom;  
  
    public int age()      { return _age;}  
    public String nom() { return _nom;}  
  
    public Personne (int a, String n) {  
        _age=a;  
        _nom=n;  
    }  
  
    public Personne (Personne p) {  
        _age=p._age;  
        _nom=p._nom;  
    }  
}
```

```
    public void miseAJour(int a,String n){  
        _nom=n;  
        _age=a;  
    }  
  
    public String toString() {  
        return ("Personne : "+_nom+" : "+  
                _age);  
    }  
    ... }
```

❖ Déclaration de l'héritage

- Le mot clé utilisé pour signifier l'héritage : «**extends**»
- En Java on ne peut hériter que d'une seule classe à la fois

❖ Constructeur

- Le constructeur de la classe de base (*ici **Personne***) est appelé par l'instruction « **SUPER** »
 - «super» doit être la 1ère instruction
- Le(s) constructeur(s) d'une classe dérivée devront faire appel au(x) constructeur(s) de la classe de base :
 - implicitement (utilisation du constructeur par défaut)
 - ou explicitement (super ...)

```
public class Employe extends Personne {  
    private int _salaire;  
    public Employe (int a, String n, int s) {  
        super(a,n);  
        _salaire=s;  
    }  
    public Employe (Personne p, int salaire) {  
        super(p);  
        _salaire=salaire;  
    }  
    ...  
}
```


❖ **La classe dérivée hérite automatiquement des attributs et des méthodes de la classe de base**

```
public class Personne {
    private int _age;
    private String _nom;

    public int age() {return _age;}
    public String nom() {return _nom;}
    ...
    public Personne (Personne p) {
        _age=p._age;
        _nom=p._nom;
    }
    ...
}
```

```
public class Employe extends Personne {
    private int _salaire;
    ...
    public Employe (Personne p, int salaire) {
        super(p);
        _salaire=salaire;
    }
    public Employe (int a, String n, int s) {
        super(a,n);
        _salaire=s;
    }

    public static void main(String[] args){
        Personne p1=new Personne(29, "toto");
        Employe e2=new Employe(p1,4000);
        Employe e3=new Employe(20,"titi",1000);
        System.out.println(e2.nom());
        System.out.println(e3.age());
        ...
    }
}
```



❖ Super

- Permet d'invoquer le constructeur de la classe de base
- Permet d'accéder aux membres de la classe de base

```
public class Personne {  
    private int _age;  
    private String _nom;  
  
    public int age() { return _age; }  
    public String nom() { return _nom; }  
    ...  
    public Personne (Personne p) {  
        _age=p._age;  
        _nom=p._nom;  
    }  
    public void miseAJour(int a,String n) {  
        _nom=n;  
        _age=a;  
    }  
    public void miseAJour() {  
        _nom="*";  
        _age=0;  
    }  
    ...  
}
```

```
public class Employe extends Personne {  
    private int _salaire;  
    ...  
    public Employe (Personne p, int salaire) {  
        super(p);  
        _salaire=salaire;  
    }  
    public void miseAJour(int a,String n,int s) {  
        miseAJour(a,n);           // pas ambiguïté  
        _salaire=s;  
    }  
    public void miseAJour() { // enrichissement  
        super.miseAJour(); // ambiguïté levée  
        _salaire=0;  
    }  
    public static void main(String[] args){  
        Personne p1=new Personne(29, "toto");  
        Employe e2=new Employe(p1,4000);  
        e2.miseAJour(e2.age(),e2.nom(),6000);  
        e2.miseAJour(); System.out.println(e2);  
    }  
    ...  
}
```

❖ Redéfinition (≠ surcharge)

- Une méthode est redéfinie si elle possède la même signature (nom + type des paramètres) et le même type de retour que dans la classe de base

```
public class Personne {
    private int _age;
    private String _nom;

    public int age() { return _age; }
    public String nom() { return _nom; }
    ...
    public Personne (Personne p) {
        _age=p._age;
        _nom=p._nom;
    }
    public String toString() {
        return ("Personne : "+_nom+" : "
                +_age);
    }
    ...
}
```

```
public class Employe extends Personne {
    private int _salaire;
    ...
    public Employe (Personne p, int salaire) {
        super(p);
        _salaire=salaire;
    }
    public String toString() {
        return ("Employe : " +nom()+" : "+
                age()+" / "+ _salaire);
    }
    public static void main(String[] args){
        Personne p1=new Personne(29, "toto");
        System.out.println(p1); // toString de Personne
        Employe e2=new Employe(p1,4000);
        System.out.println(e2); // toString de Employe
    }
    ...
}
```

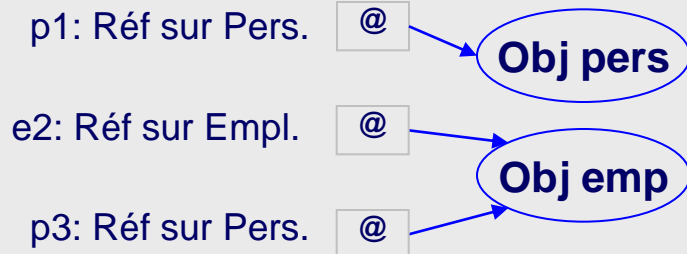


- ❖ La redéfinition est particulièrement intéressante quand elle est utilisée avec la notion d'attachement dynamique / Polymorphisme
 - L'attachement dynamique (mécanisme implicite de Java) permet à la JVM de choisir la bonne méthode en fonction du type réel (de la classe) de l'objet appelant

```
public class Personne {  
    private int _age;  
    private String _nom;  
    ...  
    public Personne (Personne p) {  
        _age=p._age;  
        _nom=p._nom;  
    }  
    public String type() {  
        return "Personne";  
    }  
    ...  
}
```

```
public class Employe extends Personne {  
    private int _salaire;  
    ...  
    public Employe (Personne p, int salaire) {  
        super(p);  
        _salaire=salaire;  
    }  
    public String type() {  
        return "Employe";  
    }  
    public static void main(String[] args){  
        Personne p1=new Personne(29, "toto");  
        Employe e2=new Employe(p1,4000);  
        Personne p3=e2;  
        System.out.println("type de e2 : "+e2.type());  
        System.out.println("type de p1 : "+p1.type());  
        System.out.println("type de p3 : "+p3.type());  
        ...  
    }  
}
```

type de e2 : Employe
type de p1 : Personne
type de p3 : Employe





❖ Utilisation d'un tableau de <Personne>

- Nb: les <Employe> sont aussi des <Personne>
- On met dans le tableau des objets de type <Employe> et <Personne>
- L'attachement dynamique permet de manipuler les éléments du tableau de personnes en fonction de leur classe !

```
public class Personne {  
    private int _age;  
    private String _nom;  
  
    public int age() {return _age;}  
    public String nom() {return _nom;}  
    ...  
    public String type() {  
        return "Personne";  
    }  
    ...  
}
```

```
Personne : toto : 29 / type = Personne  
Personne : tata : 27 / type = Personne  
Employe : titi : 29 / 7000 / type = Employe
```

```
public class Employe extends Personne {  
    private int _salaire;  
    ...  
    public String type() {  
        return "Employe";  
    }  
    public static void main(String[] args){  
        Personne p1=new Personne(29, "toto");  
        Personne p2=new Personne(27, "tata");  
        Employe e1=new Employe(29, "titi",7000);  
        Personne[] tabPers= new Personne[3];  
        tabPers[0]=p1;  
        tabPers[1]=p2;  
        tabPers[2]=e1;  
        for (int i=0; i< tabPers.length; i++) {  
            System.out.println( tabPers[i] + " / type = "  
                                + tabPers[i].type() );  
        }  
    }  
}
```



❖ Accès aux membres propres d'un l'objet dérivé via une référence de type de la classe de base

- Vérifier le type de l'objet : sinon risque d'erreur à l'exécution
- Faire un CAST / sinon erreur à la compilation

```
public class Personne {  
    private int _age;  
    private String _nom;  
    ...  
    public String type() {  
        return "Personne";  
    }  
    ...  
}
```

Méthode spécifique

Méthode redéfinie

Vérification du type de l'objet / Polymorphisme

Cast :
Réf. sur Personne -> Réf. sur Employe

```
Personne : toto : 29 / type = Personne  
Personne : tata : 27 / type = Personne  
Employe : titi : 29 / 7000 / type = Employe  
Employe 2 salaire 7000
```

```
public class Employe extends Personne {  
    private int _salaire;  
    public int salaire() { return _salaire;}  
    public String type() {  
        return "Employe";  
    }  
    public static void main(String[] args){  
        Personne p1=new Personne(29, "toto");  
        Personne p2=new Personne(27, "tata");  
        Employe e1=new Employe(29, "titi",7000);  
        Personne[] tabPers= new Personne[3];  
        tabPers[0]=p1; tabPers[1]=p2; tabPers[2]=e1;  
  
        for (int i=0; i< tabPers.length; i++) { ...  
            if ((tabPers[i].type()).equals("Employe")) {  
                System.out.println("Employe " + i  
                    + " salaire "  
                    + ((Employe)tabPers[i]).salaire());  
            }  
        }  
    }  
}
```



- ❖ **Vérifier le type de l'objet référencé**
 - **Utilisation de la méthode Java : instanceof**

```
public class Personne {  
    private int _age;  
    private String _nom;  
    ...  
    public String type() {  
        return "Personne";  
    }  
    ...  
}
```

Vérification du
type de l'objet / instanceof

Cast :
Réf. sur Personne -> Réf. sur Employe

```
Personne : toto : 29 / type = Personne  
Personne : tata : 27 / type = Personne  
Employe : titi : 29 / 7000 / type = Employe  
Employe 2 salaire 7000
```

```
public class Employe extends Personne {  
    private int _salaire;  
    public int salaire() { return _salaire;}  
    public String type() {  
        return "Employe";  
    }  
    public static void main(String[] args){  
        Personne p1=new Personne(29, "toto");  
        Personne p2=new Personne(27, "tata");  
        Employe e1=new Employe(29, "titi",7000);  
        Personne[] tabPers= new Personne[3];  
        tabPers[0]=p1; tabPers[1]=p2; tabPers[2]=e1;  
  
        for (int i=0; i< tabPers.length; i++) {  
            if (tabPers[i] instanceof Employe)  
                System.out.println("Employe " + i  
                    + " salaire "  
                    + ((Employe)tabPers[i]).salaire());  
            ...  
        }  
    }  
}
```

A decorative graphic consisting of several concentric, overlapping circular arcs in a light blue color, creating a sense of motion or a stylized 'C' shape, positioned behind the chapter title.

Chapitre 15

Contrôles d'accès

❖ Contrôle d'accès des membres d'une classe

- **membre public**
 - tout le monde peut y accéder de partout
- **membre sans préciser (rappelle la notion d'ami)**
 - seules les classes du même *package* peuvent y accéder (visibilité au niveau *package*)
- **membre protected**
 - toutes les classes du même package peuvent y accéder ! En plus les classes dérivées (même de *packages* différents) peuvent y accéder
- **membre private**
 - seuls les membres de la classe peuvent y accéder

Droits d'accès	private	(sans préciser)	protected	public
depuis la classe	ok	ok	ok	ok
depuis une classe du même package	non	ok	ok	ok
depuis une classe d'un package différent	non	non	ok si classe dérivée	ok

A decorative graphic consisting of several concentric, overlapping circular arcs in a light blue color, creating a sense of motion or a stylized 'C' shape, positioned on the right side of the slide.

Chapitre 16

Classe Object

- ❖ **Toutes les classes JAVA héritent par défaut de la classe «Object»**
 - la classe Object est donc la racine hiérarchique de toutes les classes

<pre>class Personne { ... }</pre>	<i>// implicitement équivalent à</i>
-----------------------------------	--------------------------------------

<pre>class Personne extends Object { ... }</pre>
--

- ❖ **Quelques méthodes de la classe Object (redéfinissables)**

Object	
	Object()
String	toString()
Object	clone()
boolean	equals(Object)
...	

- représentation d'un objet sous forme de chaîne
- pour la création de la copie d'un objet (cf. notion de clonage...)
- pour définir un critère d'égalité entre deux objets

A decorative graphic consisting of several concentric, overlapping circular arcs in a light blue color, creating a sense of motion or a stylized 'C' shape that frames the central text.

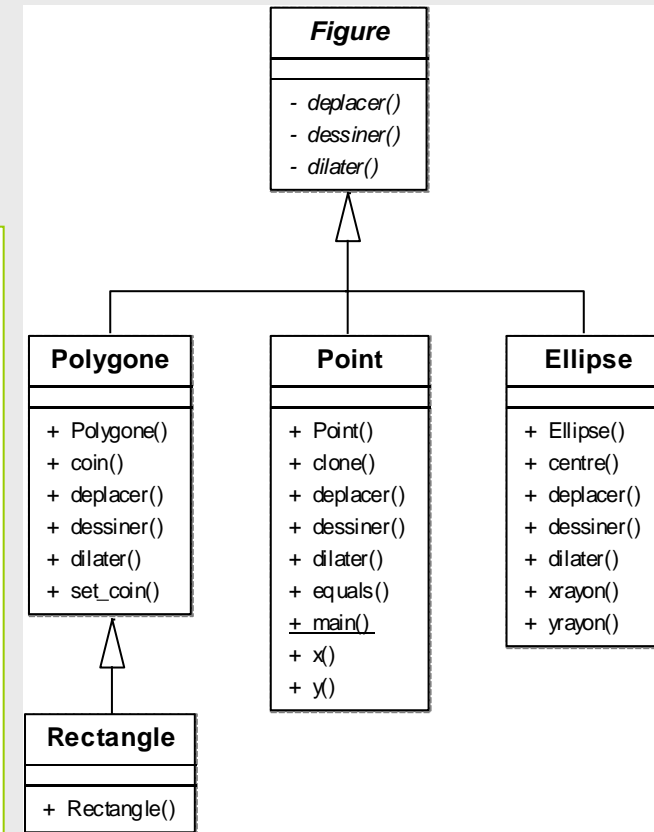
Chapitre 17

Egalités d'objets (redefinition)

❖ Ecriture d'un opérateur equals

- Possibilité de le surcharger
 - => pas de polymorphisme

```
class Point extends Figure{
//...
public boolean equals(Point p){ return ((p._x==_x) && (p._y==_y)); }
public static void main(String[] args){
    Figure[] fig = new Figure[4];
    fig[0] = new Ellipse(new Point(150, 150),100,100);
    fig[1] = new Rectangle(new Point(140, 140), new Point(160, 160));
    fig[2] = new Point(130, 200);
    fig[3] = new Point(1, 2);
    for (int i=0; i< fig.length; i++){
        System.out.println(fig[i].equals(new Point(1,2)));
    }
    System.out.println(".....");
    for (int i=0; i< fig.length; i++){
        if (fig[i] instanceof Point) {
            System.out.println(((Point)fig[i]).equals(new Point(1,2)));
        }
    }
}
```



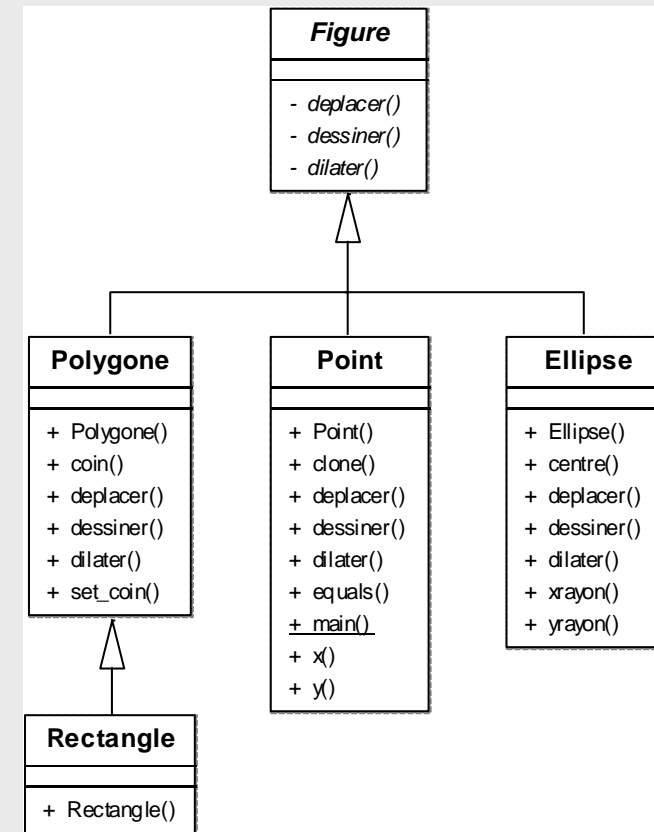
false
 false
 false
 false

 false
 true

❖ Ecriture d'un opérateur equals

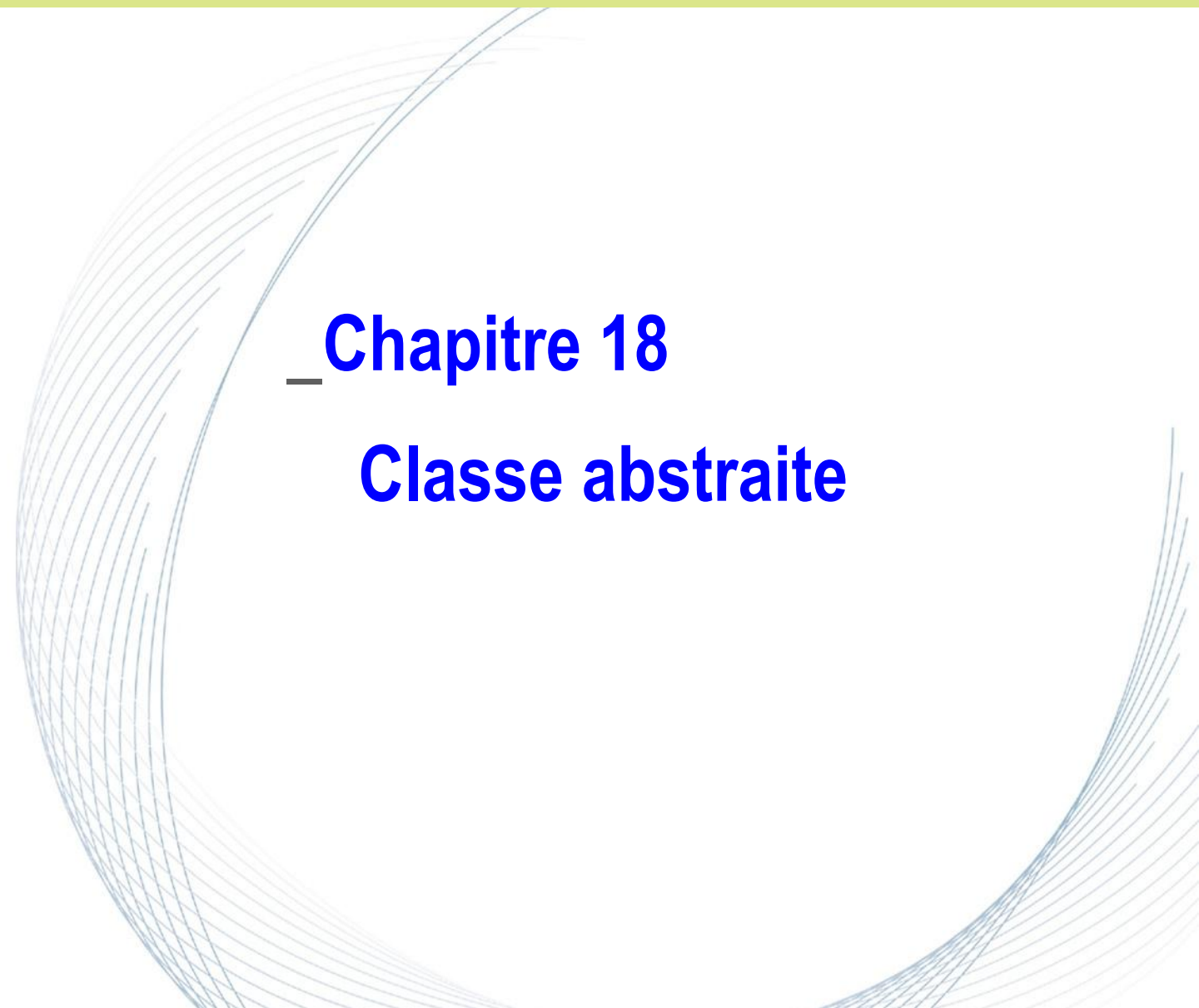
- Possibilité de le redéfinir / class Object
 - => polymorphisme

```
class Point extends Figure{
public boolean equals(Object p){
    if (p instanceof Point) {
        return (((Point)p)._x==_x) && (((Point)p)._y==_y); }
    else return false;
}
public static void main(String[] args){
    Figure[] fig = new Figure[4];
    fig[0] = new Ellipse(new Point(150, 150),100,100);
    fig[1] = new Rectangle(new Point(140, 140), new Point(160, 160));
    fig[2] = new Point(130, 200);
    fig[3] = new Point(1, 2);
    for (int i=0; i< fig.length; i++){
        System.out.println(fig[i].equals(new Point(1,2)));
    }
    System.out.println(".....");
    for (int i=0; i< fig.length; i++){
        if (fig[i] instanceof Point) {
            System.out.println(((Point)fig[i]).equals(new Point(1,2)));
        }
    }
}
```



false
 false
 false
 true

 false
 true

A decorative graphic consisting of several concentric, overlapping circular arcs in a light blue color, creating a sense of motion or a stylized 'C' shape, positioned behind the chapter title.

Chapitre 18

Classe abstraite

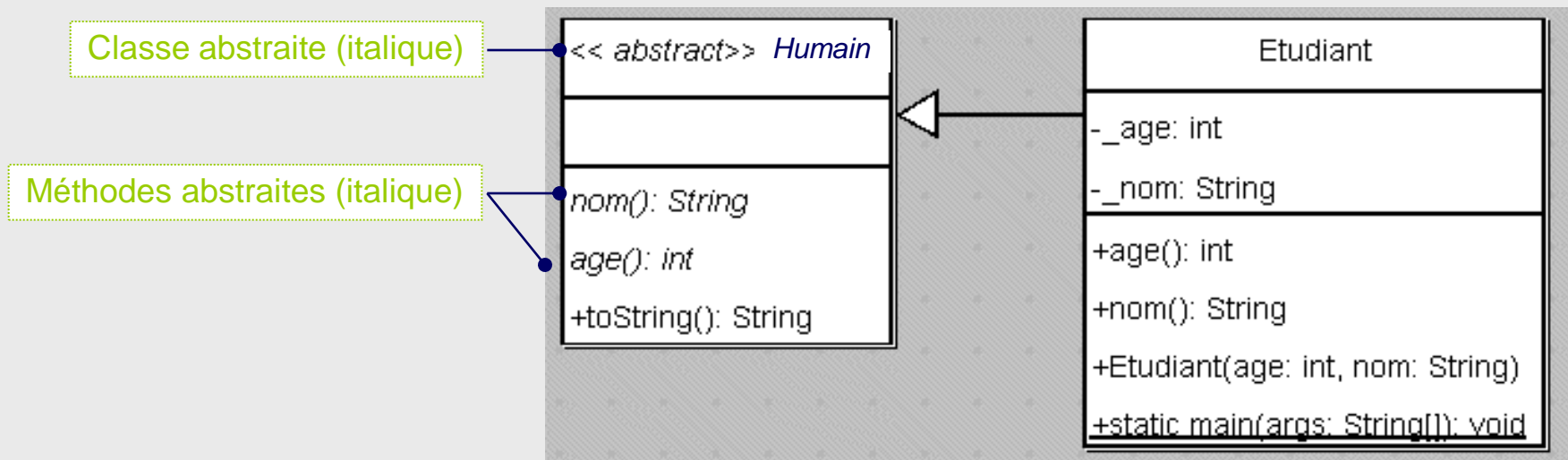
❖ Une classe abstraite

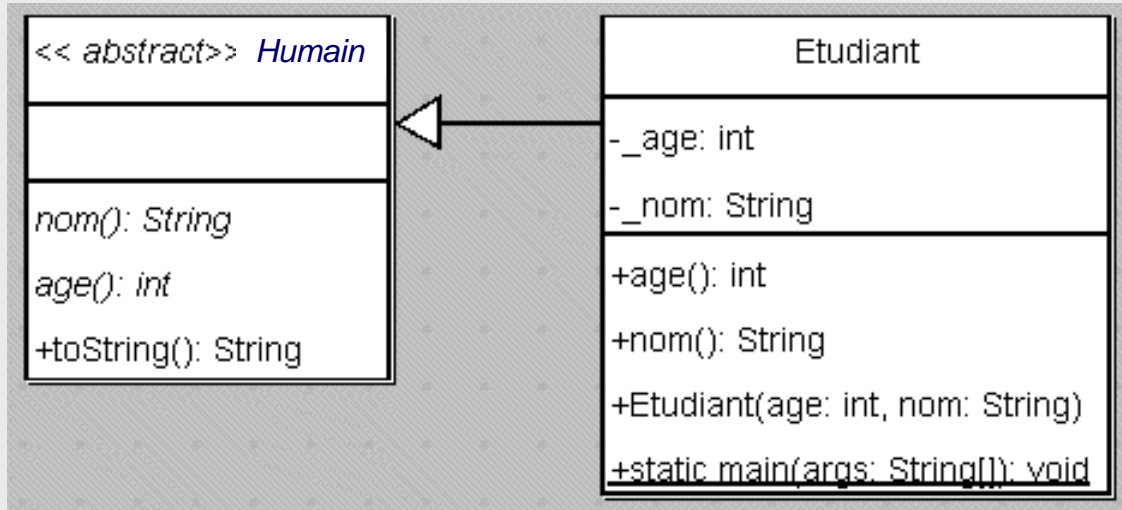
- elle possède 0 ou plusieurs méthodes abstraites
- une classe déclarée abstraite ne peut pas être directement instanciée

❖ Une méthode abstraite

- elle possède un corps vide
- si une méthode est abstraite => classe déclarée comme abstraite
- objectif : forcer les classes dérivées (concrètes) à redéfinir toutes les méthode(s) abstraites de la classe de base abstraite

❖ Exemple de diagramme de classes





abstract **class** X { ... }

```

abstract class Hmain {
    abstract String nom();
    abstract int age();
    public String toString() {
        return nom();
    }
}
  
```

```

public class Etudiant extends Hmain {
    private int _age;
    private String _nom;
    public int age() { return _age; }           // redef. obligatoire
    public String nom() { return _nom; }       // redef. obligatoire
    public Etudiant (int age, String nom) {
        _age=age;
        _nom=nom;
    }
    public static void main(String[] args){
        Etudiant e1=new Etudiant(29, "toto");
        System.out.println(e1);
    } ...}
  
```

A decorative background graphic consisting of several concentric, overlapping circular arcs in a light blue color, creating a sense of motion or a stylized 'C' shape.

Chapitre 19

Interface

❖ Définition

- Une **interface Java** est une sorte de classe abstraite "pure" :
 - Sans donnée (sauf *static* et *final*)
 - Possède seulement des fonctions abstraites (publiques)
- Une **classe** peut "**implémenter**" plusieurs Interfaces (\neq héritage)
 - Lorsqu'une classe implémente une interface elle garantit de définir toutes les méthodes de l'interface
 - \Leftrightarrow elle garantit de respecter le protocole défini par l'interface

❖ Implantation

```
interface Imprimable {
    public void imprime();
}
```

```
public class Personne implements Imprimable
{
    ...
    // redéfinition obligatoire
    // imposée par l'interface Imprimable
    public void imprime() {
        System.out.println(this);
    }
    ...
}
```

```
public class Employe extends Personne {...}
```

❖ On caractérise les classes <Imprimable>

- avec l'Interface <Imprimable>
- Par exemple : les classes <Personne> et <Employe>

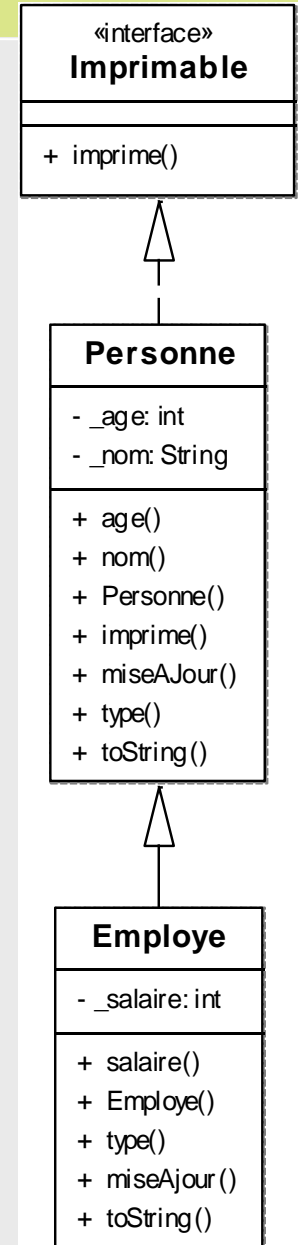
```
public class Personne implements Imprimable {
    public void imprime() { System.out.println(this); }
    ...
}
```

```
public class Employe extends Personne {
    // ...

    public static void main(String[] args){
        Personne p1=new Personne(29, "toto");
        Personne p2=new Personne(27, "tata");

        Imprimable i1=new Employe(29, "titi",7000);
        Imprimable i2=p1;
        Imprimable i3=p2;
        i1.imprime();
        i2.imprime();
        i3.imprime();
    }
}
```

```
Employe : titi : 29 / 7000
Personne : toto : 29
Personne : tata : 27
```



❖ Exemple avec les classes <Personne> <Employe> et <Rectangle>

- On peut rassembler dans un tableau des classes respectant le protocole <Imprimable>
 - L'interface garantit que tous les objets du tableau (même de nature différente) sont imprimables

```
interface Imprimable {
    void imprime();
}
```

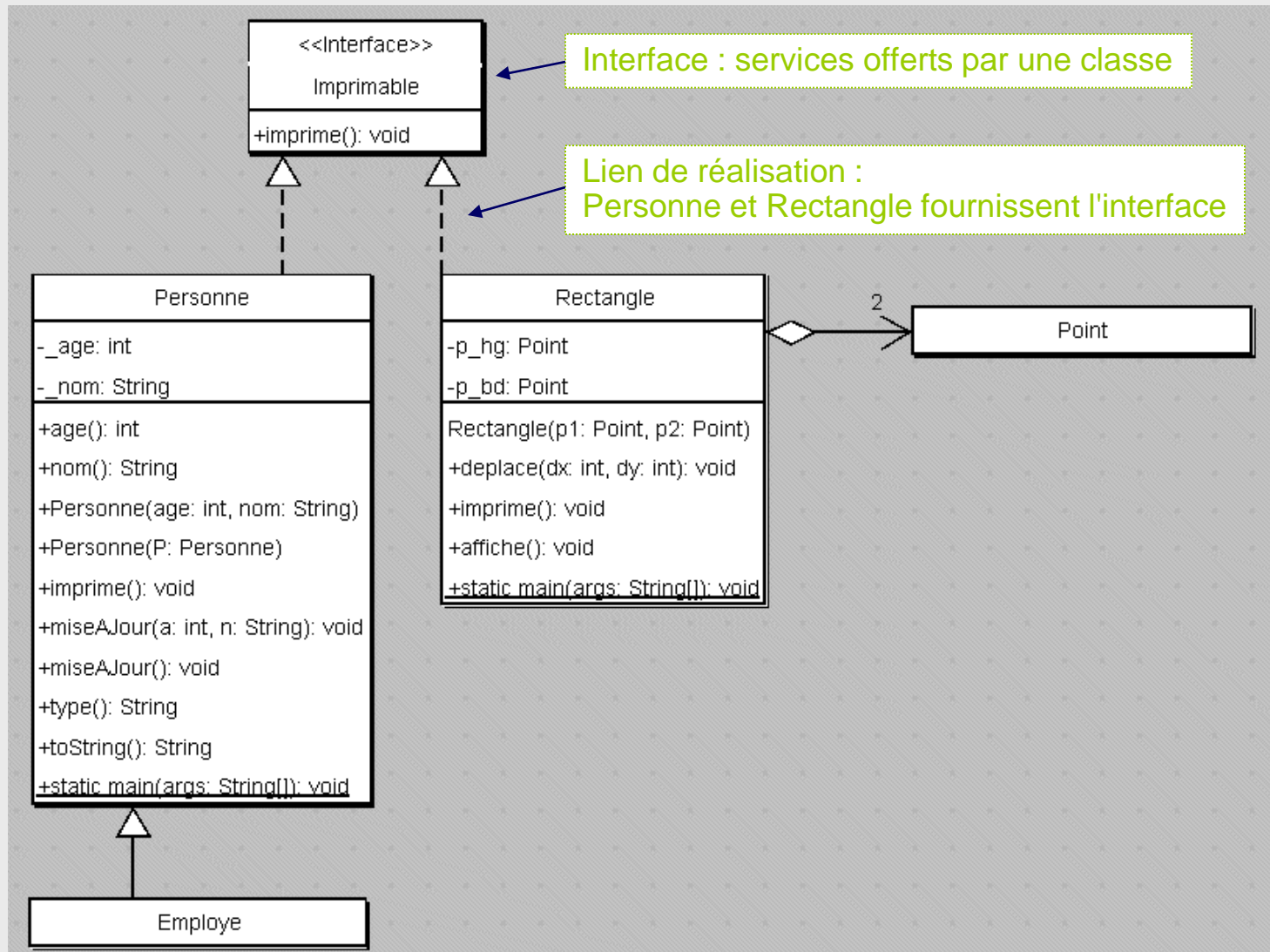
```
public class Rectangle
    implements Imprimable {
    private Point p_hg;
    private Point p_bd;
    public void imprime(){
        System.out.println("Rectangle : " +
            p_hg + " " + p_bd);
    }
    ...
}
```

```
public static void main(String[] args){
    Personne p1=new Personne(29, "toto");
    Employe e1=new Employe(29, "titi",7000);
    Rectangle r1 = new Rectangle (new Point(1,1) ,
                                   new Point (100,100));

    Imprimable[] tablImpression= new Imprimable[3];
    tablImpression[0]=p1;
    tablImpression[1]=e1;
    tablImpression[2]=r1;
    for (int i=0; i< tablImpression.length; i++) {
        tablImpression[i].imprime();
    }
}
```

```
Personne : toto : 29
Employe : titi : 29 / 7000
Rectangle : (1,1) (100,100)
```

❖ Diagramme de classes



❖ Héritage classique

- Les interfaces peuvent être organisées de manière hiérarchique via le mécanisme d'héritage.

Interface intFaceA extends intFaceB { ...}

- On peut avoir un héritage multiple d'interfaces :

interface intFaceA extends intFaceX, intFaceY {...}

❖ Une Interface

- spécifie la forme (spécification, définition d'un comportement) de quelque chose (d'un concept) (\neq fournir une implémentation)
- Les méthodes de l'interface définissent le protocole (signature) à faire respecter pour ce concept

❖ Classe abstraite

- Une classe abstraite est une classe incomplète qui nécessitera une spécialisation (une dérivation)
- A utiliser pour initialiser une hiérarchisation de classes (classe de base)

A decorative background graphic consisting of several concentric, overlapping circular arcs in a light blue color, creating a sense of motion or a stylized 'C' shape.

Chapitre 20

Collection Java

❖ Les collections

- Structures de données de différents types permettant le regroupement d'un "ensemble" d'éléments dans une même entité
- Permet le stockage et la manipulation des ces éléments

❖ Les collections en Java2 (outils et boîte à outils)

- Java2 offre un "framework" pour la gestion des collections
 - Structuration générique basée sur une hiérarchie d'interfaces
 - Organisation des structures de données en fonction de leur type
 - Implémentation de nouvelles structures guidée par les contraintes de généricité (réutilisation et interopérabilité facilité)
 - Manipulation unifiée des éléments
 - indépendamment du type de la structure
 - Algorithmes de base fournis pour les opérations sur les collections
 - tri, recherche, etc.

❖ Organisation en Java2 :

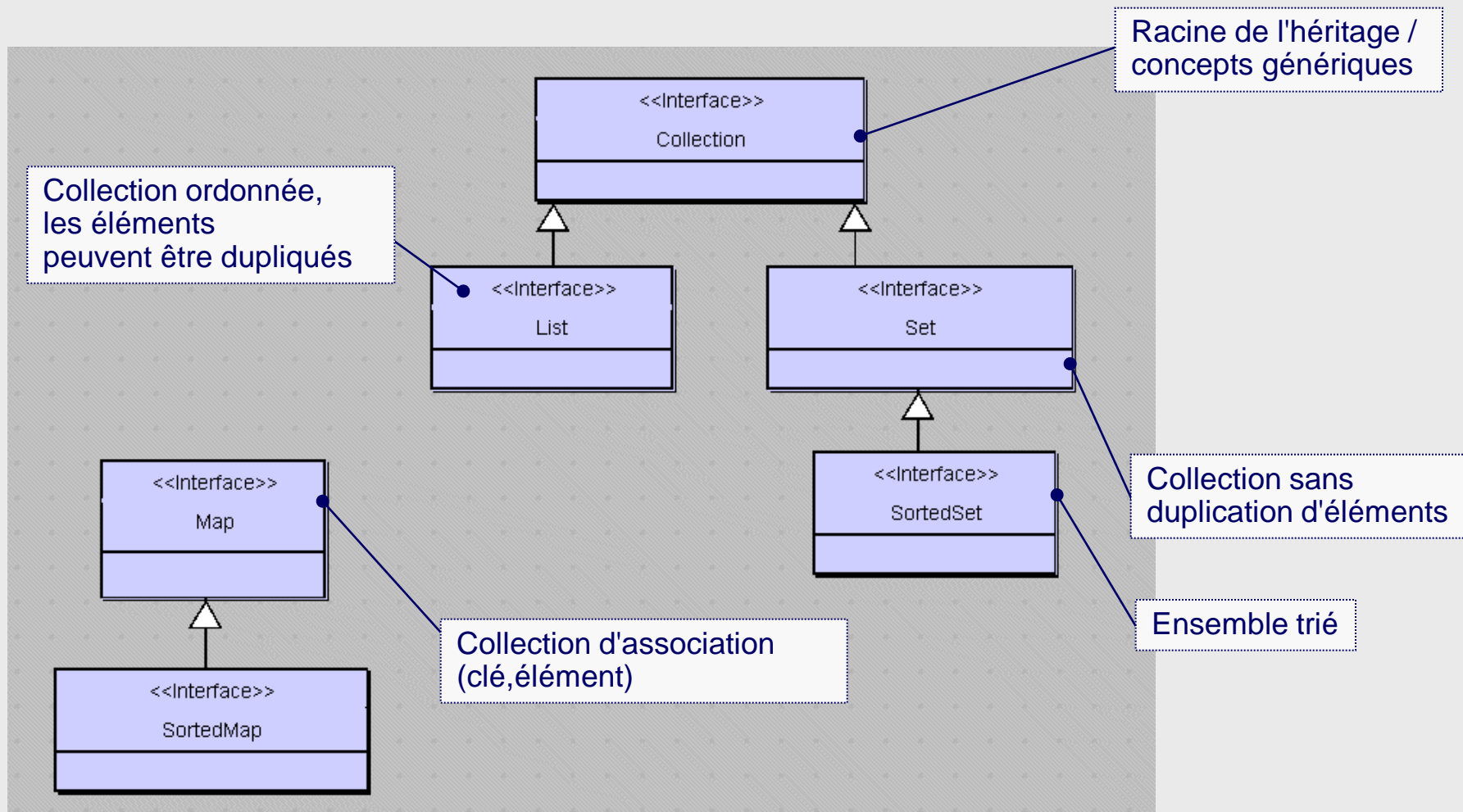
- Gestion homogène des structures de données :
interface / classe abstraite / classe concrètes : java.util

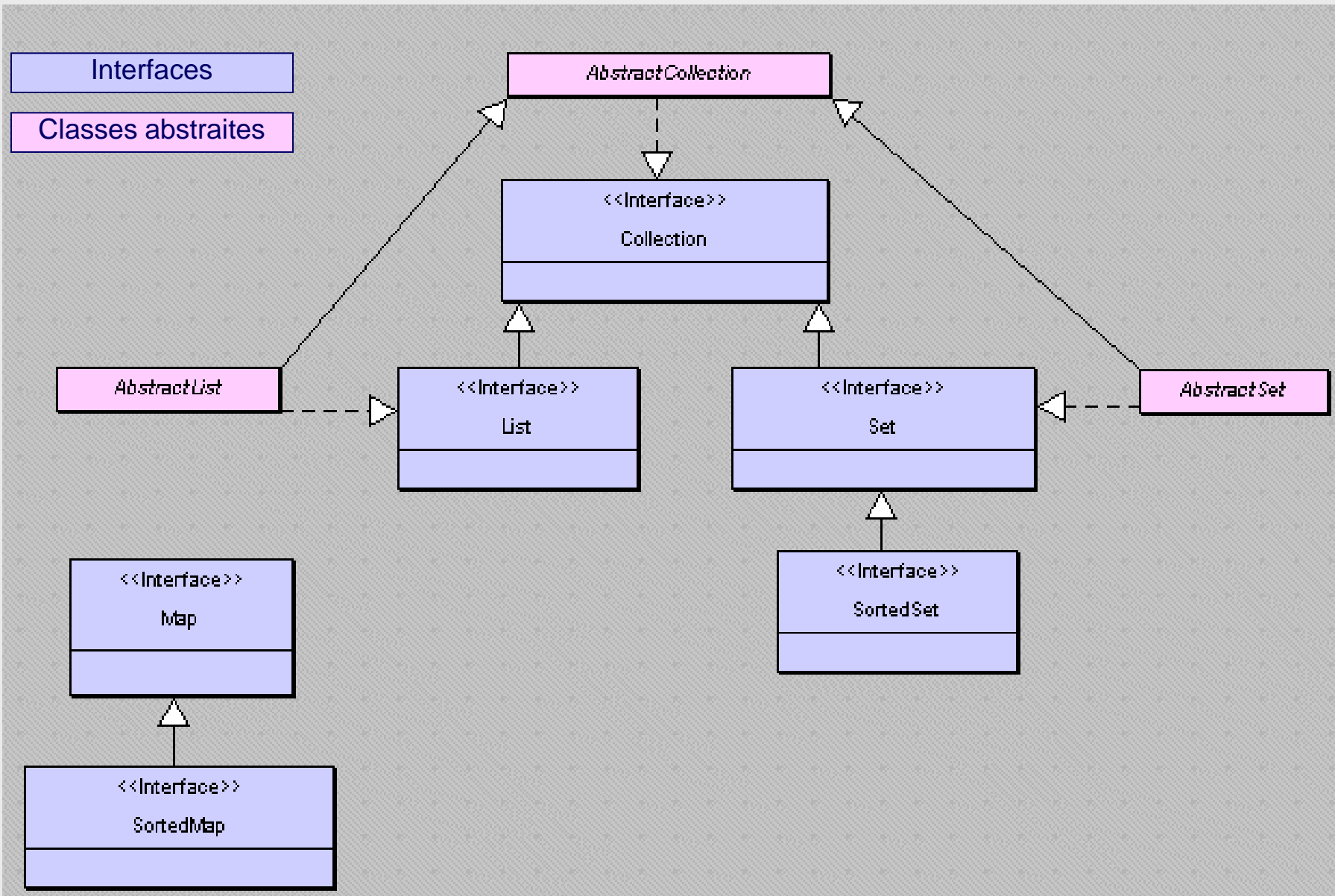
Interface		Classes abstraites / concrètes	
Collection	Set (collections sans duplication d'éléments)	AbstractSet	HashSet TreeSet
	List (collections ordonnées, duplication possible)	AbstractList	ArrayList Vector ...
	...		
Map (collections associant à chaque élément une clé / maintient des clés selon un ordre ...)	SortedMap		HashMap TreeMap ...
... autre ...			Stack, Arrays, Bitset ...

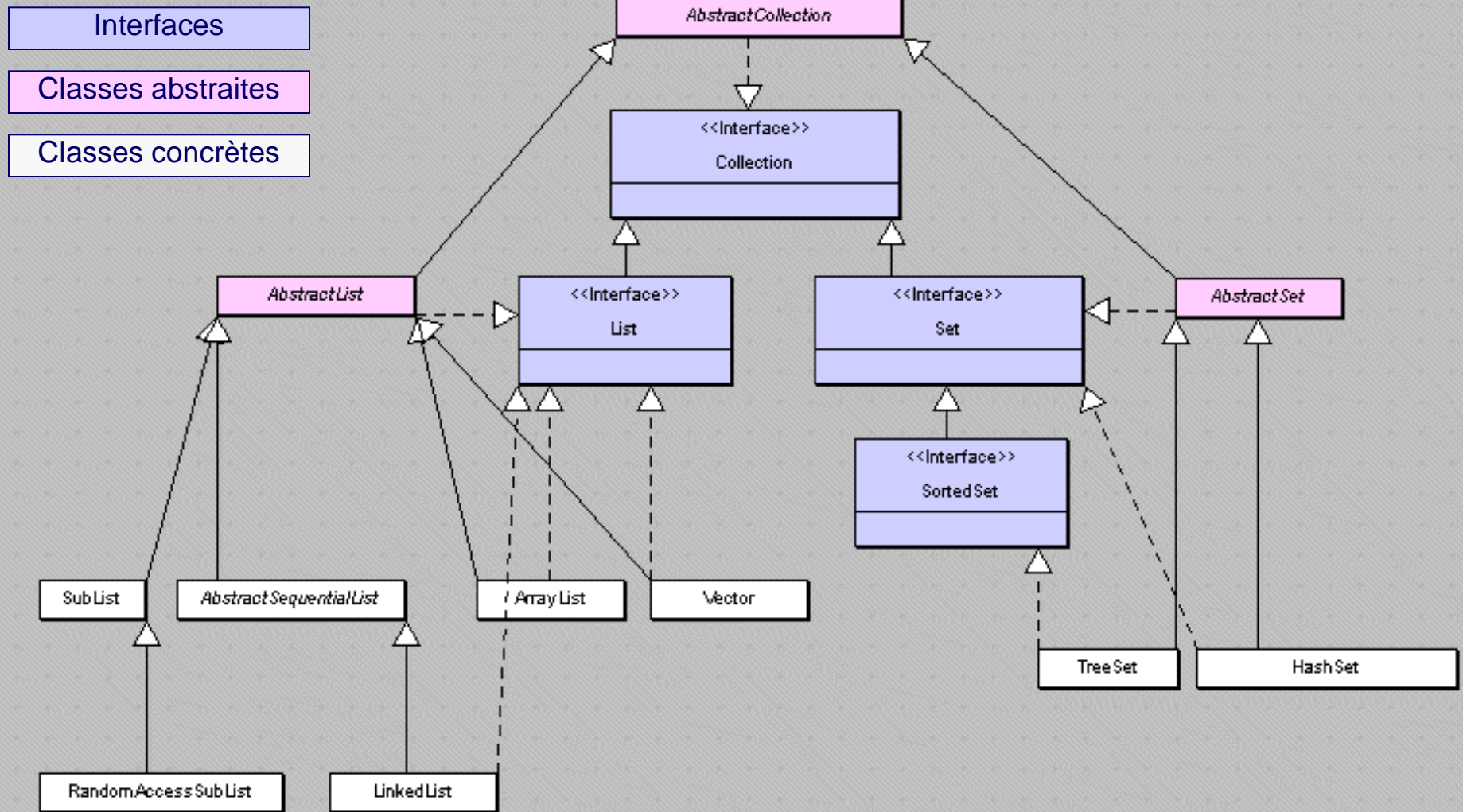
❖ **L'interface collection**

- **Définition d'un protocole minimum pour l'utilisation et la conception de Collections en Java**
- **Exemple**
 - **add(Object o)**
pour ajouter un élément à une collection
 - **remove(Object o)**
pour retirer une instance d'un élément à une collection
- **Pas d'implémentation directe en Java**

<<Interface>> Collection
size(): int isEmpty(): boolean contains(o: Object): boolean iterator() toArray(): Object[] toArray(a: Object[]): Object[] add(o: Object): boolean remove(o: Object): boolean containsAll(c: Collection): boolean addAll(c: Collection): boolean removeAll(c: Collection): boolean retainAll(c: Collection): boolean clear(): void equals(o: Object): boolean hashCode(): int







❖ L'interface Iterator

- Permet le parcours d'une collection sans connaître le détail de sa structure

❖ L'interface ListIterator

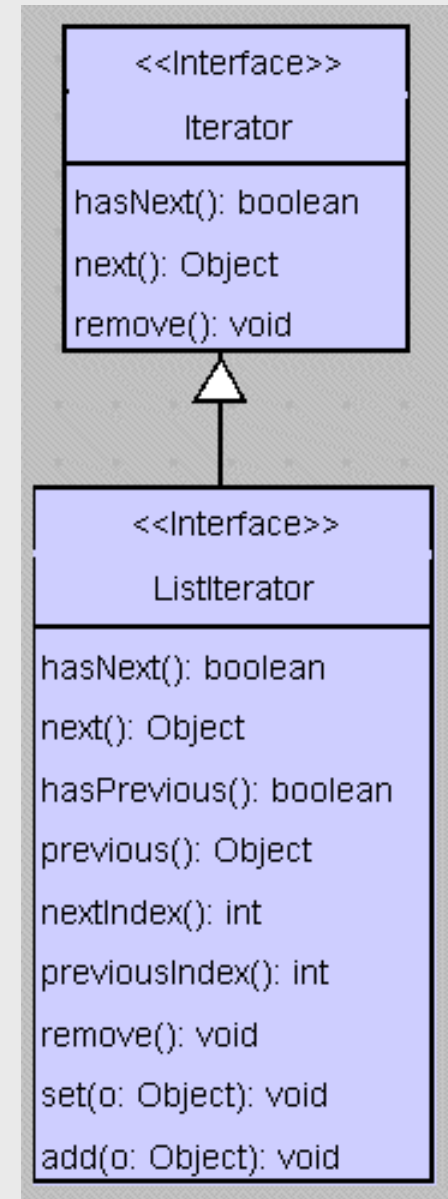
- spécialisation permettant d'itérer dans les deux sens

// exemple de méthode générique qui fonctionne sur une collection quelque soit son type

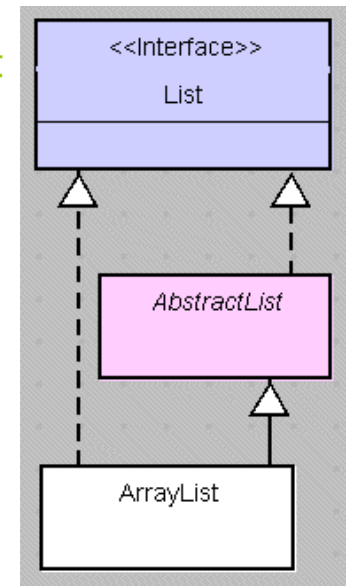
```
public static void Parcour() {
    Iterator it = MaCollection.iterator();    // on récupère un
                                              // itérateur sur une
                                              // collection

    while (it.hasNext()) {
        // hasNext() vérifie qu'il reste au moins 1 élément à parcourir

        System.out.println("elts : " + it.next());
        // next() retourne le prochain élément dans l'itération si il y
        // en a un, sinon exception
    }
}
```




```
import java.util.*;
public class ExempleCollection {
    static ArrayList maArrayList= new ArrayList(); // idem à un tableau dynamique
    // Collection de type ArrayList : Liste utilisant un tableau(Array) comme structure de données
    public static void init() {
        Integer[] tabEnt =new Integer[] {    new Integer(2), new Integer(4),new Integer(6),
                                             new Integer(8),new Integer(10),new Integer(12) };
        List maList = java.util.Arrays.asList(tabEnt);
        // utilisation d'une classe "d'aide" sur les Array // la classe Arrays
        // asList(Object[]) : permet la transformation d'un Array en une Liste
        maArrayList.addAll(maList); // insertion de tous les éléments de maList
    }
    public static void affiche() {
        Iterator it = maArrayList.iterator();
        while (it.hasNext()) { System.out.println("elts : " + it.next()); }
        System.out.println("---");
    }
    public static void main (String[] arg) {
        init();
        affiche();
        maArrayList.remove(1); // enlève l'élément situé à l'index 1
        maArrayList.add("coucou"); // ajout d'un élément en plus en fin de Liste
        affiche();
    }
}
```



A decorative background graphic consisting of several concentric, overlapping circular arcs in a light blue color, creating a sense of motion or a stylized 'C' shape.

Chapitre 21

Exception

❖ Différents types d'erreurs

- Mauvaise gestion des classes : accès hors tableau, ...
- Entrées utilisateurs non valides : effacer un fichier inexistant, ...
- Liées aux périphériques : manque de papier dans l'imprimante, ...
- Limitation physique : disque plein, ...
- ...

❖ Le traitement des erreurs / les différentes possibilités

- Retourner un code d'erreur
- Ne rien faire (absorber l'erreur)
- Imprimer des messages d'erreur
- Mettre à jour des variables globales d'erreur
- Utiliser le mécanisme d'exception unifié et standardisé

.... Java supportent le mécanisme de gestion d'exception ...

Le traitement de cas exceptionnel ne peut très souvent être opéré que dans un contexte supérieur à la détection de l'erreur

❖ Mécanisme d'exception / gestionnaire d'exception

- Permet de transmettre le problème du traitement de l'erreur dans un **contexte qualifié** (de niveau supérieur) pour gérer l'erreur
- **Simplifie** en allégeant le code de la gestion locale des erreurs par une **recentralisation** des procédures de traitement d'erreur

❖ Mise en œuvre des exceptions

- on **déclenche** (lance) une exception par l'instruction **throw**
- on **capte** (attrape) une exception dans un bloc de type **try**
- on **traite** (gère) une exception avec l'instruction **catch**

Public class **TestException**

```
{ // ...
  public methode1()
  { try
    {
      methode2() //...
    }
    catch (IOException e)
    {
      System.out.println("Gestion de l'exception");
      ...
    }
    catch (Exception e)
    { ... }
  }
  finally
  { // bloc "finally" optionnel
    ...
  }
  // suite du code ...
}
```

void methode2 () **throws** IOException

```
{
  methode3();
  // ...
}
```

void methode3 () **throws** IOException

```
{
  1 write (...) // Erreur : exception déclenchée !!!!!
  //...
}
```

- 1** Arrêt de l'exécution : lancement de l'exception
- 2** Transmission du control au bloc supérieur ...
- 3** Capture de l'exception par un bloc "catch"
- 4** Traitement de l'exception
- 5** Le bloc "finally" est toujours exécuté : qu'il y ait capture, propagation ou déroulement normal du programme
- 6** Le programme reprend après le bloc "finally"

-----> Exécution sans exception

❖ Lorsque qu'intervient une exception (réception par la JVM)

- L'exécution normale du programme est arrêté
- Recherche du bloc de traitement de l'exception (catch)
 - en local,
 - puis on remonte la liste des appelants
- Traitement de l'exception
- Tous les blocs "finally" rencontrés sont exécutés
- Reprise du code

```
i == 1
i == 2
erreur i est égal a 0
reprise du code
```

```
public class TestException {

    public void test(int i) throws Exception {
        if (i==0) throw new Exception("erreur i est égal a 0");
        System.out.println("i == " + i);
    }

    public static void main(String[] args){
        TestException t = new TestException();
        try {
            t.test(1); t.test(2); t.test(0); t.test(3);
        }
        catch(Exception e) {
            System.out.println(e.getMessage());
        }
        System.out.println("reprise du code");
    }
}
```

- ❖ Une exception est lancée par la commande **throw e**;
 - e : est un objet qui DOIT dériver de la classe Throwable
- throw** new IllegalArgumentException("Pb en lecture");
- ❖ **Spécification obligatoire des exceptions susceptibles d'être lancées**
 - Toute fonction susceptible d'émettre des exceptions "explicites" doit le mentionner (*vérification à la compilation*) :
 - Exceptions levées dans la méthode et non attrapées par celle-ci
 - Exceptions levées dans des méthodes appelées par la méthode ...
 - Exceptions levées et traitées par la méthode puis propagées

```
public void read(DataInputStream in) throws IOException
{
    ... double s = in.readDouble ( );           // peut générer une exception
    ... }
```

- Une fonction sans clause **throws**
 - garantit qu'elle ne va pas générer d'exception explicite

❖ Capture (bloc try) et traitement d'exceptions (clause catch)

```
try
{ // code susceptible de déclencher une exception
}
catch (Type1 id1) { // capture des exception de type Type1
                  // ou type dérivé de Type1
                  // traitement de l'exception de Type1
}
catch (Type2 id2) { // capture des exception de type Type2 ...
                  // traitement de l'exception de Type2
}
// etc.
```

❖ Remarques

- Les blocs try encapsulent de nombreux appels de fonctions ...
- Le transfert de contrôle est donné à la 1ère clause catch de bon type
- Il est possible d'exploiter la notion d'héritage pour hiérarchiser les exceptions => l'ordre des blocs catch doit respecter l'ordre d'héritage

❖ Il est possible de retransmettre une exception

```
try
{
    // code
}
catch (Throwable t)    // capture tout type d'exception
{
    ...
    throw t;            // retransmet l'erreur courante
}
```

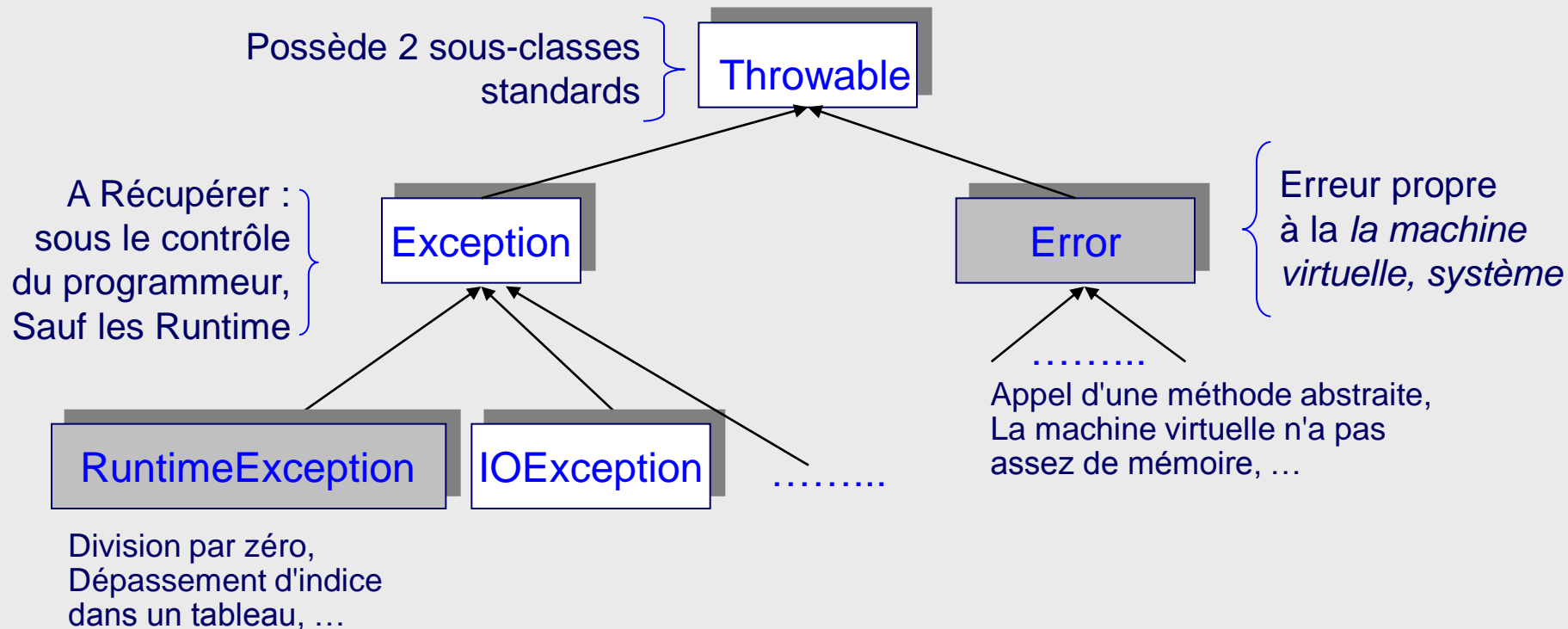
❖ Si une exception est propagée sans être rattrapée :

- Si propagation jusqu'à la méthode "main"

```
public static void main (String[] args) throws Exception
```

- Affichage d'un message d'erreur et de la pile des appels,
- Arrêt de l'exécution du programme.

❖ Java possède une hiérarchie d'exceptions standards



- **Exception techniques et d'erreurs (Runtime, Error) : un programme n'est pas obligé de lever ces 2 types d'exception (raisons essentiellement pratiques)**
- **Exception explicite ou applicatives : pour toutes les autres exceptions le programme doit les lever (imposé par le compilateur)**
- **Pour définir de nouveaux types d'exception on hérite en général de `java.lang.Exception`**

❖ **Java.lang.Throwable**

Throwable
String message
Throwable() Throwable(string s) String getMessage() Void printStackTrace() Void printStackTrace(PrintStream) ...

- Message d'erreur décrivant l'exception
- Constructeur avec et sans message d'erreur
- Retourne le message d'erreur
- Imprime sur la sortie standard ou sur un stream, l'exception et la trace de l'exception dans la pile

❖ **Exemple de récupération du message de l'exception**

```

try { ....}
catch (Exception e)
{
    System.out.println(e.getMessage());           // ou
    System.out.println(« Exception » + e);        // cf. notion de toString
}

```

❖ Un petit exemple Java

```
public class MonException extends Exception {  
    public MonException ( ) {}  
    public MonException (String msg) {  
        super(msg);  
    }  
}
```

```
public class UneClasse  
{  
    public void g() throws MonException { // ...  
        throw new MonException("Origine: fonction g()");  
    }  
    public void h() throws MonException { // ...  
        g();  
    }  
    public static void main (String[] args) {  
        UneClasse C1=new UneClasse();  
        try { C1.h(); }  
        catch (MonException e) { e.printStackTrace(); }  
    }  
}
```

```
MonException: Origine: fonction g()  
    at UneClasse.g(UnClasse.java:9)  
    at UneClasse.h(UnClasse.java:13)  
    at UneClasse.main(UnClasse.java:17)
```

```

class MonException1 extends Exception {
    public MonException1(String msg) {
        super(msg);
        System.out.println("cons MonException1" );
    }
}
class MonException2 extends MonException1 {
    public MonException2(String msg) {
        super(msg);
        System.out.println("cons MonException2" );
    }
}
public class TestException {
    public void methodeA(int p) throws MonException1 {
        if (p==1) throw new MonException1("p==1: methodeA");
        if (p==2) throw new MonException2("p==2: methodeA");
        System.out.println("fin de methodeA");
    }
    public void methodeB() throws MonException1 {
        // ...
        try {
            methodeA(2);
        } catch (MonException1 e) {
            System.out.println("Traitement partiel
                               dans methodeB : " + e);

            throw e;
        } finally {
            System.out.println("finally de methodeB");
        }
        System.out.println("fin de methodeB");
    }
}

```

```

public static void main(String[] args) throws MonException1{
    TestException C1 = new TestException();
    try {
        C1.methodeB();
    } catch (MonException2 e) {
        System.out.println("Traitement dans main() : " + e);
    } finally {
        System.out.println("finally de main()");
    }
    C1.methodeA(1);
    System.out.println("fin de main()");
}

```

cons MonException1
 cons MonException2
 Traitement partiel dans methodeB :
 MonException2: p==2: methodeA
 finally de methodeB
 Traitement dans main() :
 MonException2: p==2: methodeA
 finally de main()
 cons MonException1
 Exception in thread "main"
 MonException1: p==1: methodeA
 at TestException.methodeA(TestException.java:17)
 at TestException.main(TestException.java:45)

— Chapitre 22

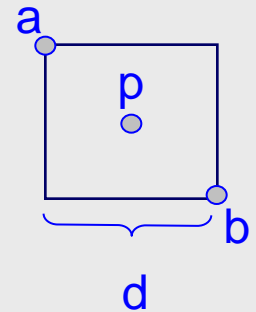
Clonage

❖ En JAVA

- Copie d'identificateur : pas de recopie des objets identifiés
- Il faut souvent implanter la recopie (... en profondeur si besoin ...)

❖ Gestion des recopies d'objet avec le constructeur par recopie

```
Rectangle Carre (Point p, int d) {
    Point a= new Point(p) ; // constructeur par recopie
    a.deplacer(-d / 2 , -d / 2);
    Point b= new Point(p) ; // constructeur par recopie
    b.deplacer(d / 2 , d / 2);
    Rectangle r=new Rectangle(a,b);
    return r;
}
```

❖ Utilisation de la méthode // alternative pour le polymorphisme : **clone()**

```
Rectangle Carre (Point p, int d) {
    Point a= (Point)p.clone() ;
    a.deplacer(-d / 2 , -d / 2);
    Point b= (Point)p.clone() ;
    b.deplacer(d / 2 , d / 2);
    Rectangle r = new Rectangle(a,b);
    return r; // résultat correct
}
```

❖ La méthode clone de la classe Object ne fait par défaut que

- la réservation mémoire
et
- la recopie superficielle (ok pour : **types primitifs** et les objets **invariants**)

❖ De plus

- clone() est une fonction protégée (surcharger "public" possible) de la classe de base Object.
- l'accès à cette fonction est protégé et seules les classes mettant en œuvre l'interface de balisage Cloneable peuvent y accéder (une sorte d'avertissement)

=> Implantation explicite de la méthode clone() pour vos classes

❖ **Implantation de la méthode clone()**

- l'interface Cloneable doit être mise en œuvre
- le contrôle d'accès à cette fonction nécessite la mise en place du traitement d'exception

 *gestion d'exception

```
class Point implements Cloneable
{
    public Object clone( )
    {
        try { return super.clone(); } // copie superficielle
        catch (CloneNotSupportedException e) { return null; }
    }
    ...
    private int _x;   private int _y;
}
```

❖ Copie en profondeur

- Les champs « **constants** » sont implicitement gérés (nombres, string, ...)
- Les champs « **identificateurs** » doivent être clonés
(toutes les classes référencées doivent être "récursivement" clonées)

```
class Rectangle implements Cloneable {  
    public Object clone( ) {  
        try {  
            Rectangle r = (Rectangle)super.clone() ;  
            r._pt_hg = (Point) _pt_hg.clone() ;  
            r._pt_bd = (Point) _pt_bd.clone() ;  
            return r;  
        }  
        catch (CloneNotSupportedException e) {  
            return null;  
        }  
    }  
    ...  
}
```

*// fonction clone() à définir
// pour la classe Point*

La notion de « cloneable » est transmise aux classes dérivées

❖ **La méthode `super.clone()`, l'héritage et l'attachement dynamique**

```
class Niv1 implements cloneable {  
    public Niv1(int i) { _i=i; }  
    public Object clone()  
    {    try { return super.clone(); }    // copie superficielle  
        catch (CloneNotSupportedException e) { return null; }  
    }  
    ...  
    private int _i;  
}
```

```
class Niv2 extends Niv1 {  
    private int j;    // type primitif → copie automatique  
    public Niv2(int i) { super(i); }  
    ...  
}
```

- Quand `Niv1.clone()` est appelée à travers `Niv2.clone()` ; `Object.clone()` est appelée à travers `super.clone()` qui travaillera alors avec Niv2

❖ Interface Cloneable

```
interface Cloneable { }
```

```
class Object
{
    protected Object clone()
    {
        if (! (this instanceof Cloneable))
            throw new CloneNotSupportedException();

        ...

    }

    ...
}
```

❖ La classe Vector,... : met en œuvre une copie superficielle

- Recopie en profondeur :
 - faire appel à la méthode clone() (copie superficielle)
 - **Vector v2=(Vector)v1.clone();**
 - parcourir le vecteur pour cloner explicitement tous ses éléments
 - **for (int i=0; i<v2.size(); i++)**
v2.setElementAt((v2.elementAt(i)).clone(), i);

A decorative graphic consisting of several concentric, curved blue lines that sweep across the lower half of the slide, creating a sense of motion and depth.

Chapitre 23

Classe interne

❖ Possibilité de déclarer une classe à l'intérieur d'une autre classe

- Association de la visibilité de bloc avec la visibilité de classe
- Accès aux membres de la classe englobante
- Permet de "cacher" les classes internes liées à une implémentation spécifique et de rassembler les classes connexes

❖ Trois déclinaisons des classes internes

- Classe membre d'une classe
- Classe locale à une méthode membre d'une classe
- Classe anonyme définie à l'intérieur d'une expression

```
class Englobe { ...
    class ClasseMembre { ... }
}
```

```
class Englobe { ...
    void methode() {
        class ClasseLocale { ... }
    }
}
```

```
class Englobe { ...
    void methode() {
        Ob.addActionListner( new ClasseAnonyme() { ... });
    }
}
```

- Une classe membre interne peut être déclarée comme : *public*, *private*, *protected*.
- Elle a accès aux membres (attributs/méthodes), même privés, de la classe l'englobant

```

class Englobe {
    private int att1;
    private class LocalePr {
        private int att2;
        public void test() {
            att2=att1; // équivalent à this.att2=Englobe.this.att1;
        } ...
    }
    public class LocalePu { ... }
    private LocalePr vpr; // utilisation de la classe interne privée
    ...
}
...
Englobe E = new Englobe();
Englobe.LocalePu vpu= E.new LocalePu(); // classe interne publique

```

❖ **Classes internes déclarées dans un bloc de code**

- Uniquement visibles et utilisables dans leur bloc (idem aux variables locales)
- Elles ont accès en plus aux variables locales "Final" de leur bloc
 - garantit la cohérence avec les variables locales
- Elles ne peuvent pas être spécifiées comme (public, private, ...)

```

class Englobe2 {
    private int att1;
    void methode() {
        final int att2=2;
        int att3=2;
        class ClasseLocale {
            private int att;
            public void test() {
                att=att1;
                att=att2;
                att=att3; // erreur (att3 non final)
            }
        }
        // .....
    }
}

```


- Classe locale sans nom
- Contraction(combinaison) : définition et instanciation d'une classe
- Souvent utilisée dans le contexte de la gestion des évènements

```

class Englobe3 {
    private int att1;
    public void test(Point P) {
        System.out.println(P);      P.deplacer(1,1);      System.out.println(P);
    }
    public void test2() {
        test( new Point(10,10) { •
            public int _z;
            public void deplacer(int dx, int dy) {
                _x += dx; _y += dy; _z=111;
            }
            public String toString() {
                return("Point3D : " + x() + " " +y() + " " + _z);
            }
        });
    }
    public static void main(String[] args) {
        Englobe3 Eng=new Englobe3();
        Eng.test2();
    }
}

```

Classe anonyme dérivée de la classe Point

Les arguments sont passés au constructeur de la classe de base (Point)

```

Point3D : 10 10 0
Point3D : 11 11 111

```



❖ Une classe anonyme peut implémenter une interface

- Elle devient alors une classe dérivée de Object
- Il n'y a donc jamais d'arguments à sa construction

```
import java.awt.*;import javax.swing.*;
import java.awt.event.*;

public class jbouton {

    public static void main(String [] args) {
        JFrame jf = new JFrame();
        jf.setSize(400,100);
        jf.setVisible(true);

        JButton plum = new JButton("Appuyez ici");
        jf.getContentPane().setLayout(new BorderLayout());
        jf.getContentPane().add(plum, "East");
        jf.pack();

        plum.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.out.println("nombre de clic "+ i++);
            }
            int i=1;
        });
    }
}
```

La méthode `addActionListener` attend un objet qui implémente l'interface `ActionListener` :

Possibilité de construire une classe anonyme de ce type.

Contraction pour la création directe **d'une classe anonyme (combinaison de la définition et d'une instance)**

Implémente l'interface `ActionListener`

nombre de clic 1
nombre de clic 2
nombre de clic 3
fermeture de l'application

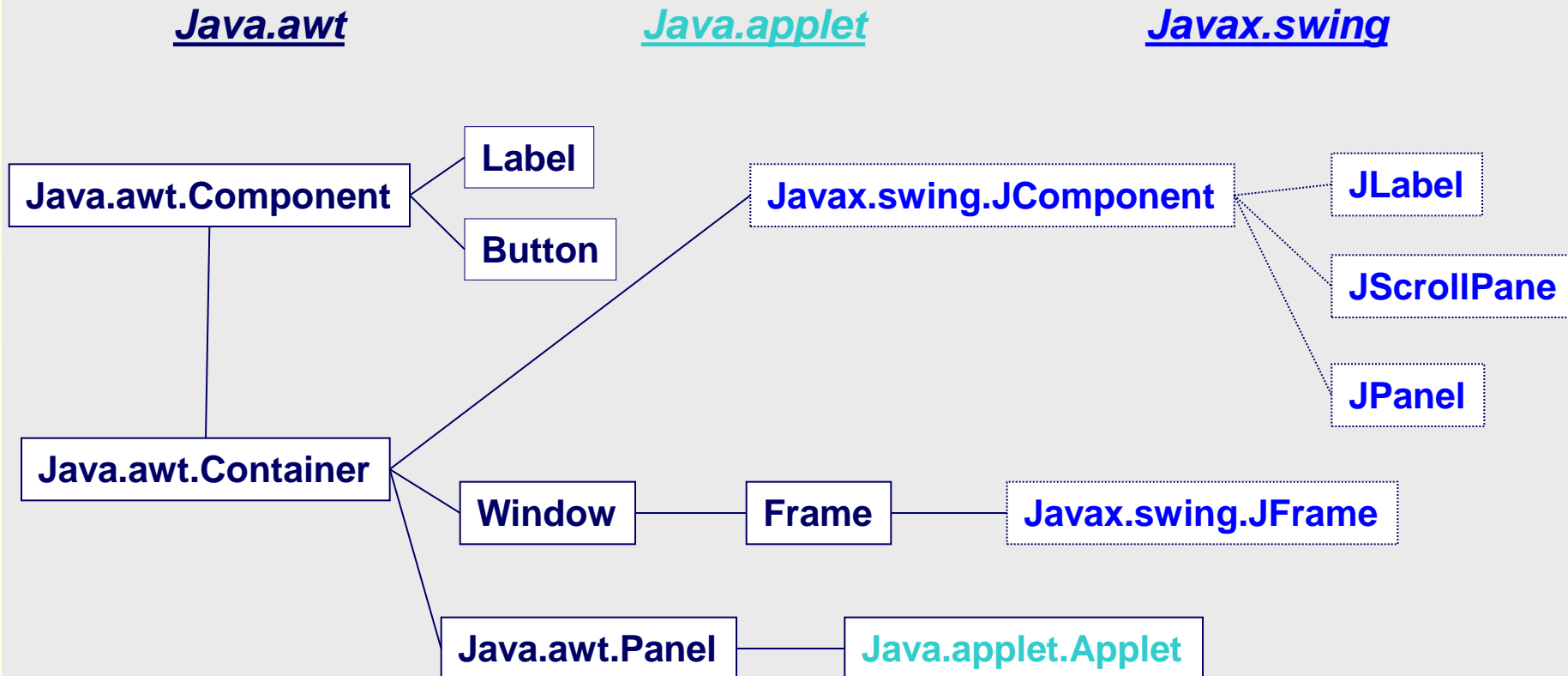
Decorative graphic element consisting of several concentric, curved blue lines that sweep across the lower half of the slide, creating a sense of motion and depth.

Chapitre 24

IHM : AWT

- ❖ **AWT : Abstract Window Toolkit** (java.awt.*)
 - **Package (JDK 1.0/1.1) pour la création d'interface**
 - gestion de fenêtre, bouton, etc.
 - **À partir de la version JDK 1.1 : cette librairie a été étendue/remplacée avec une librairie plus riche Swing.**
- ❖ **JFC (JDK 1.2) : Java Foundation Classes** (javax.swing.*)
 - **Nouvelle librairie plus riche pour la gestion d'interface qui intègre :**
 - look and feel, java 2D API (Application Programmer Interface)
 - drag-and-drop, les composants Swing, ...
 - **Les composants Swing (JScrollBar, JButton, JTextField, ...) remplacent les composants AWT qui sont plus simples mais plus rudimentaires.**
- ❖ **Terminologie :**
 - **Control** : Terme générique (widget sous Unix) définissant les éléments manipulables de l'écran : bouton, scrollbar, Text, Menu, etc.
 - **Container** (Conteneur) : fenêtres Windows pouvant contenir un ensemble de Controls ou d'autres Containers
 - **Component** (Composant) : nom collectif pour les Containers et Controls

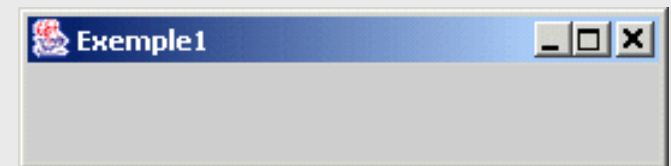
❖ Une partie de la hiérarchie de composants et conteneurs



❖ Container : JFrame Fenêtre avec bordure

- Une fenêtre d'application est modélisée par l'instance d'une classe dérivant de la classe **Frame**
 - La fenêtre va jouer un rôle de conteneur dans lequel on va ajouter des composants (menu, bouton, etc.)
 - Remarques :
 - la fenêtre est redimensionnable
 - la fermeture de la fenêtre est à la charge du concepteur

```
import javax.swing.*;  
  
public class FrameDemo {  
    public static void main(String[] args) {  
        JFrame frame = new JFrame("Exemple1");  
        frame.setSize(400,100);  
        frame.setVisible(true);  
    }  
}
```





Container : **JFrame**
Fenêtre avec bordure

```
public class LabelDemo extends JFrame{
    public LabelDemo (String s) {
        super(s);
        setSize(400,100);
        setVisible(true);
        getContentPane().setLayout( new FlowLayout() );
    }
    public static void main(String[] args) {
        JFrame frame = new LabelDemo("Exemple2");
        JLabel jl1 = new JLabel("Bonjour");
        JLabel jl2 = new JLabel(" * ");
        frame.getContentPane().add( jl1 );
        frame.getContentPane().add( jl2 );
        frame.pack();
    }
}
```

Gestionnaire d'agencement :
FlowLayout

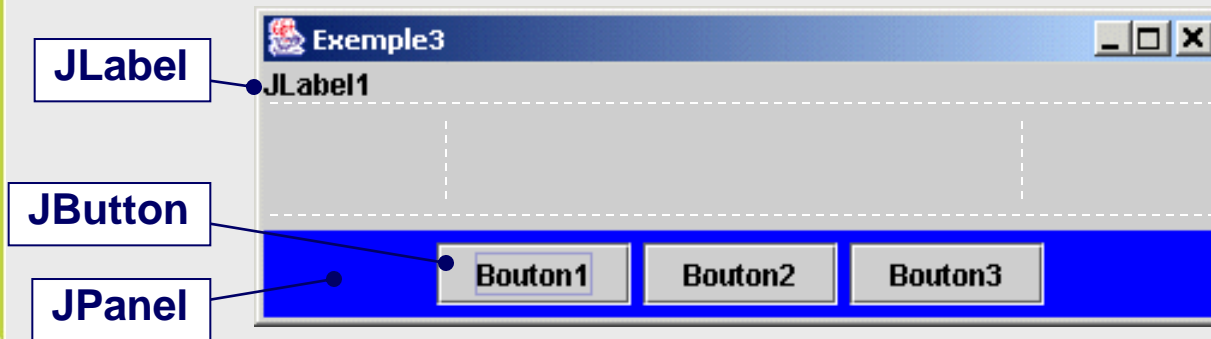
Méthode de placement des
composants d'un container

Ici : composants ajoutés
les uns après les autres,
de la gauche vers la droite

Ajout de 2 champs "texte" statiques :
JLabel

❖ Structuration d'un ensemble d'éléments dans un JPanel

- Exemple : création d'une barre de boutons



Agencement :
BorderLayout

les composants sont répartis en 5 zones :

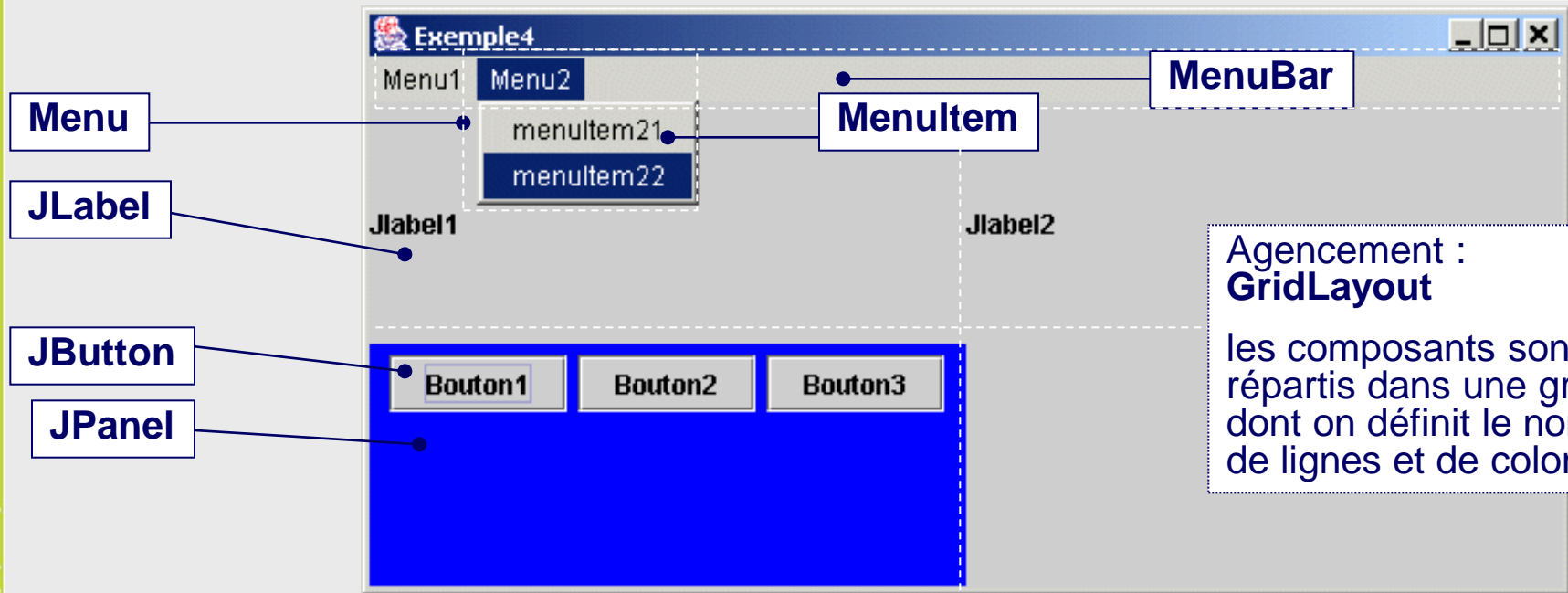
North, South, West, East, Center

```
import javax.swing.*;
import java.awt.*;

public class mesButton extends JPanel {
    public mesButton(){
        this.setBackground(Color.blue);
        this.add(new JButton ("Bouton1"));
        this.add(new JButton ("Bouton2"));
        this.add(new JButton ("Bouton3"));
    }
}
```

```
import javax.swing.*;
import java.awt.*;

public class Frame2Demo {
    public static void main(String[] args) {
        JFrame maFrame = new JFrame("Exemple3");
        maFrame.setSize(400,100);
        Container c=maFrame.getContentPane();
        c.setLayout( new BorderLayout() );
        c.add( new JLabel("JLabel1"),"North");
        c.add( new mesButton(),"South");
        maFrame.setVisible(true);
    }
}
```

```
public class monMenu extends MenuBar {
    public monMenu(){

        Menu menu1 = new Menu("Menu1");
        menu1.add(new MenuItem("menuItem11"));
        menu1.add(new MenuItem("menuItem12"));
        menu1.add(new MenuItem("menuItem13"));

        Menu menu2 = new Menu("Menu2");
        menu2.add(new MenuItem("menuItem21"));
        menu2.add(new MenuItem("menuItem22"));

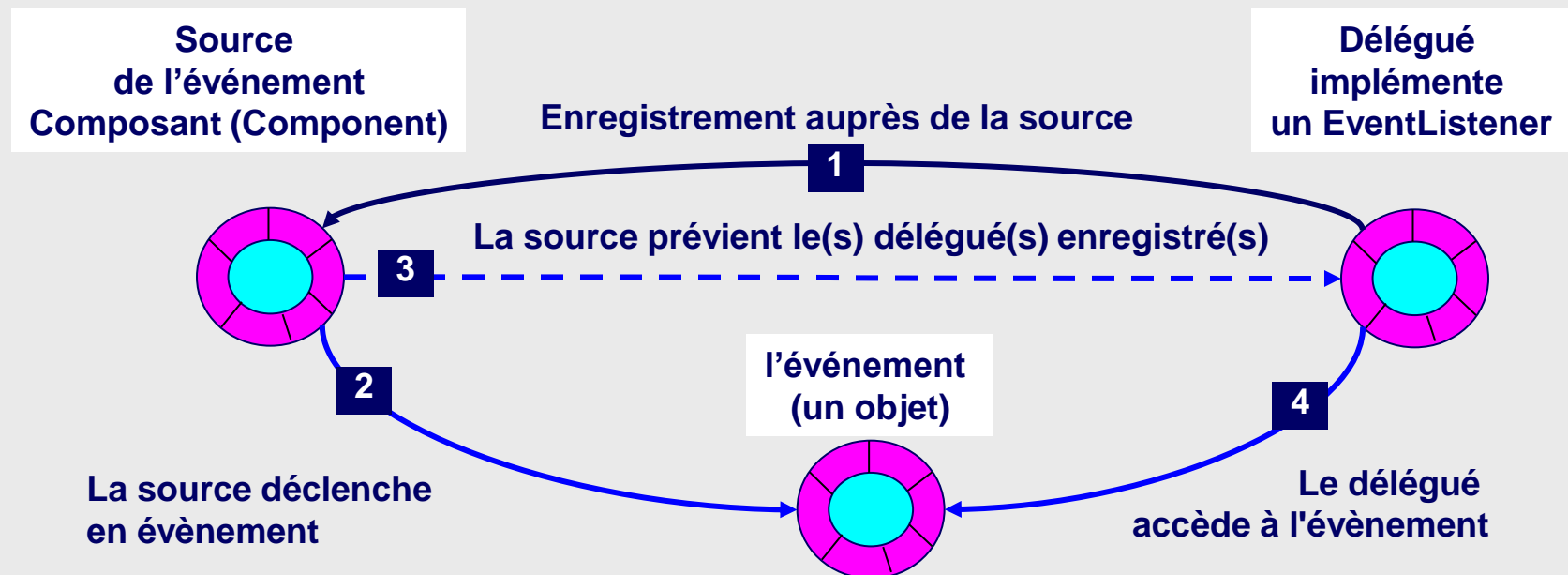
        this.add(menu1);
        this.add(menu2);

    }
}
```

```
public class Frame3Demo {
    public static void main(String[] args) {
        JFrame maFrame = new
        JFrame("Exemple4");
        maFrame.setSize(400,100);
        maFrame.setMenuBar(new monMenu());
        Container c=maFrame.getContentPane();
        c.setLayout( new GridLayout(2,2) );
        c.add( new JLabel("Jlabel1"));
        c.add( new JLabel("Jlabel2"));
        c.add(new mesButton());
        maFrame.setVisible(true);

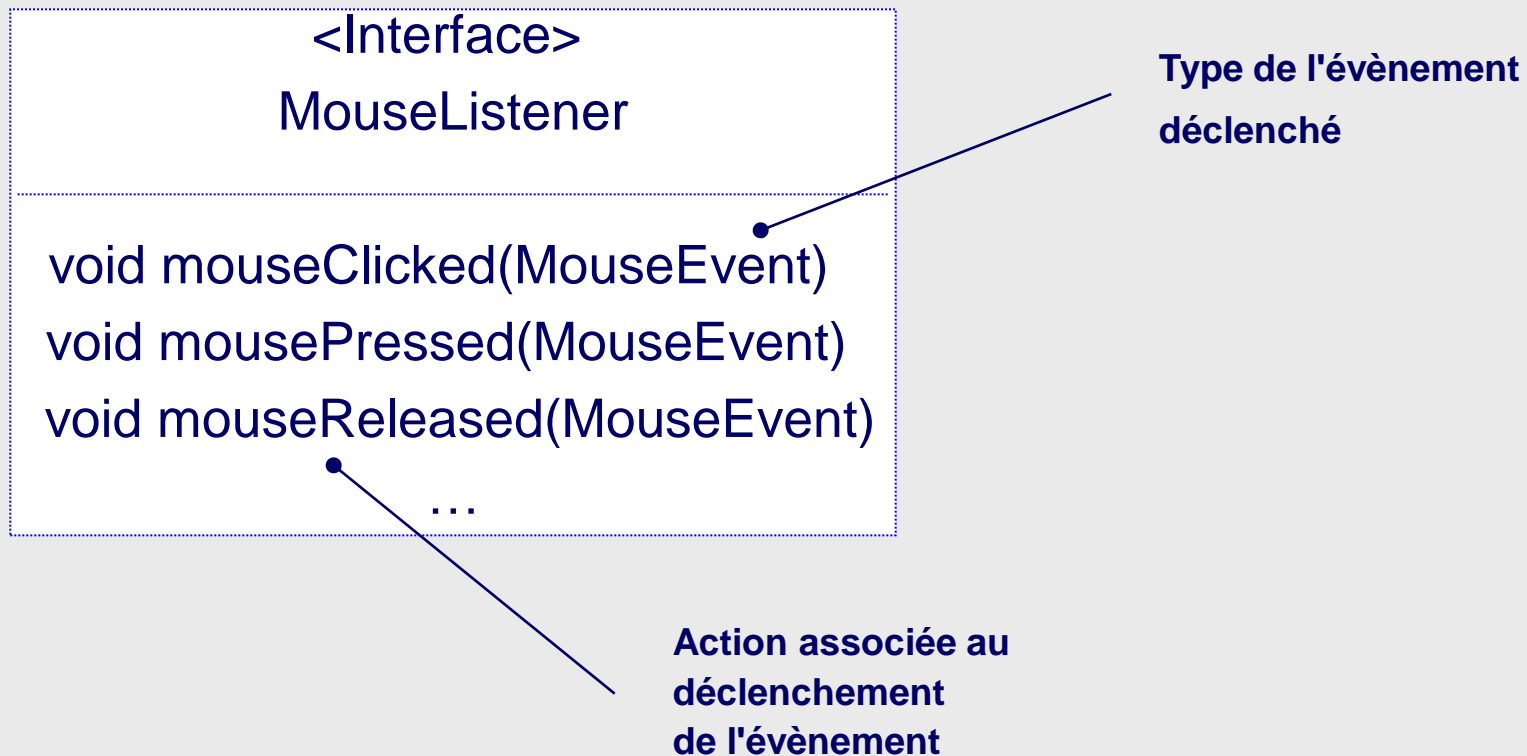
    }
}
```

- ❖ **Le modèle des événements se fonde sur 3 types d'objets**
 - **Les objets qui sont à la source des événements**
 - Les composants (component) : container, canvas, button, ...
 - **Les objets qui sont récepteurs d'évènements**
 - Les délégués de la gestion des événements (hérite de **EventListener**).
 - **Les objets événements**
 - **ActionEvent**, **MouseEvent**, **WindowEvent**



- ❖ **Les événements sont des objets classés par thèmes :**
 - **<type>Event : java.awt.event.**
 - **ComponentEvent**
 - **KeyEvent**
 - **MouseEvent**
 - **WindowEvent**
 - **...**
- ❖ **A chaque classe d'événements est associée une interface (Listener)**
 - **définit les méthodes d'écoute associées aux évènements**
 - **les objets délégués implémentent ces interfaces**
 - **<type>Listener**
 - **ComponentListener**
 - **KeyListener**
 - **MouseListener**
 - **WindowListener**
 - **...**

- ❖ Une interface définit les méthodes d'écoute associées à chaque type d'évènements
- ❖ Exemple
 - interface MouseListener associée à l'évènement MouseEvent



❖ Exemple : Gestion de la fermeture de la fenêtre

**Fenêtre vide (cadre)
avec titre : « exemple »**

```
import javax.swing.*;
import java.awt.event.*;

public class Ex1Evt {
    public static void main (String[] args) {
        • JFrame jframe = new JFrame ("exemple");
        jframe.setSize (400,100);
        jframe.setVisible(true);

        MonDelegate Del=new MonDelegate();
        jframe.addWindowListener(Del);
    }
}
```

**Enregistrement du délégué
(du Listener)**

**Délégué de type
WindowListener (interface)**

```
class MonDelegate implements WindowListener {
    public void windowClosing(WindowEvent e)
        {System.exit(0);}

    public void windowClosed(WindowEvent e) {}
    public void windowOpened(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowDeiconified(WindowEvent e) {}
    public void windowActivated(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}
}
```

**Méthodes d'écoute (de réaction)
aux évènements**

- ❖ **Pour éviter les contraintes des interfaces "Listener"**
 - Exemple : on veut uniquement redéfinir une méthode "windowClosing"
- ❖ **Utiliser un ListenerAdapter**
 - C'est une classe abstraite qui implémente les interfaces Listener avec des méthodes vides (non abstraite !)
 - Exemple : [WindowAdapter](#) / WindowListener

```
Public abstract class WindowAdapter implements WindowListener {  
    public void windowClosing(WindowEvent e) { }  
    public void windowClosed(WindowEvent e) { }  
    public void windowOpened(WindowEvent e) { }  
    public void windowIconified(WindowEvent e) { }  
    public void windowDeiconified(WindowEvent e) { }  
    public void windowActivated(WindowEvent e) { }  
    public void windowDeactivated(WindowEvent e) { }  
}
```

```
import java.awt.*;import javax.swing.*;import java.awt.event.*;

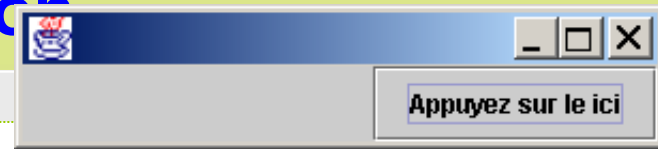
public class jboutton {

    public static void main(String [] args) {
        JFrame jf = new JFrame();
        jf.setSize(400,100);
        jf.setVisible(true);

        -----
        WindowListener l = new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.out.println("fermeture de l'application");
                System.exit(0);
            }
        };
        jf.addWindowListener(l);

        -----
        JButton plum = new JButton("Appuyez ici");
        jf.getContentPane().setLayout(new BorderLayout());
        jf.getContentPane().add(plum, "East");
        jf.pack();

        plum.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.out.println("nombre de clic "+ i++);
            }
            private int i=1;
        } );
    }
}
```



nombre de clic 1
nombre de clic 2
nombre de clic 3
fermeture de l'application

Délégation de la gestion des événements :
WindowAdapter

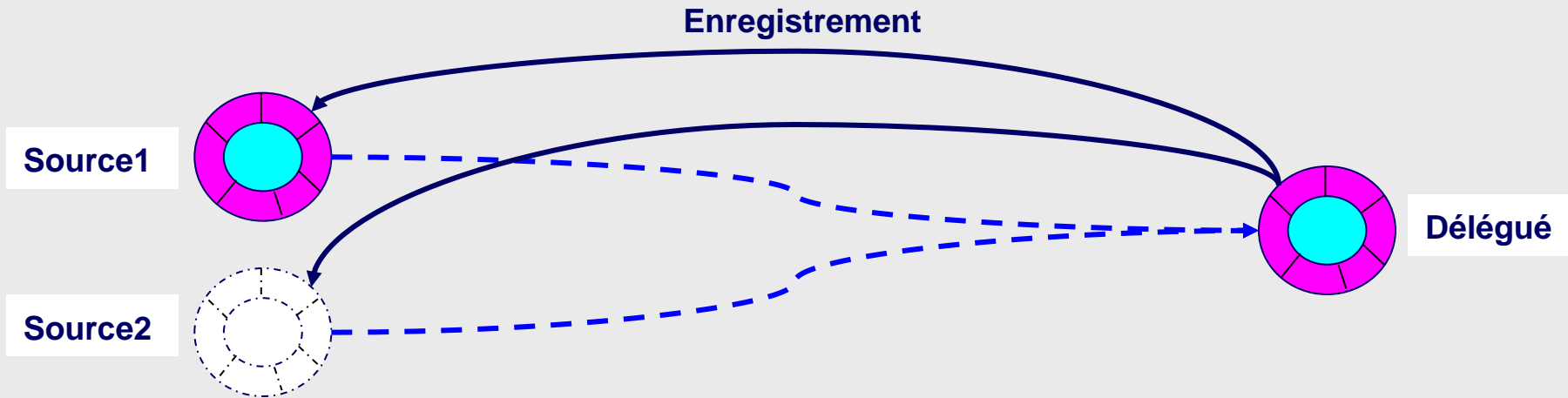
Contraction pour la création directe **d'une classe anonyme (combinaison de la définition et d'une instance)**

Gestionnaire d'agencement :
BorderLayout

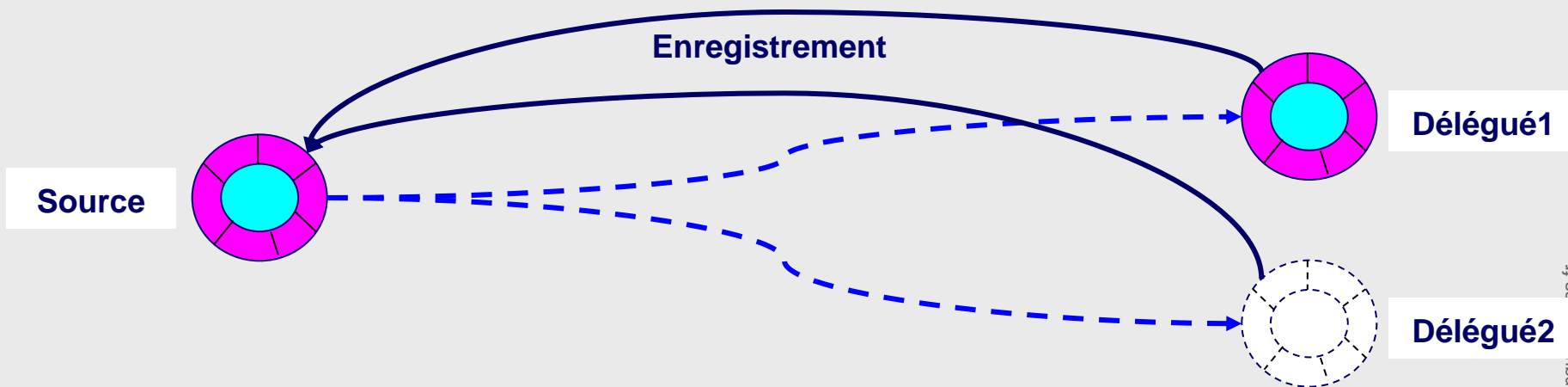
Méthode de placement des composants d'un container

Ici : décomposition du conteneur en 5 zones (North,South, ...)

- ❖ Un délégué peut être enregistré auprès de plusieurs sources



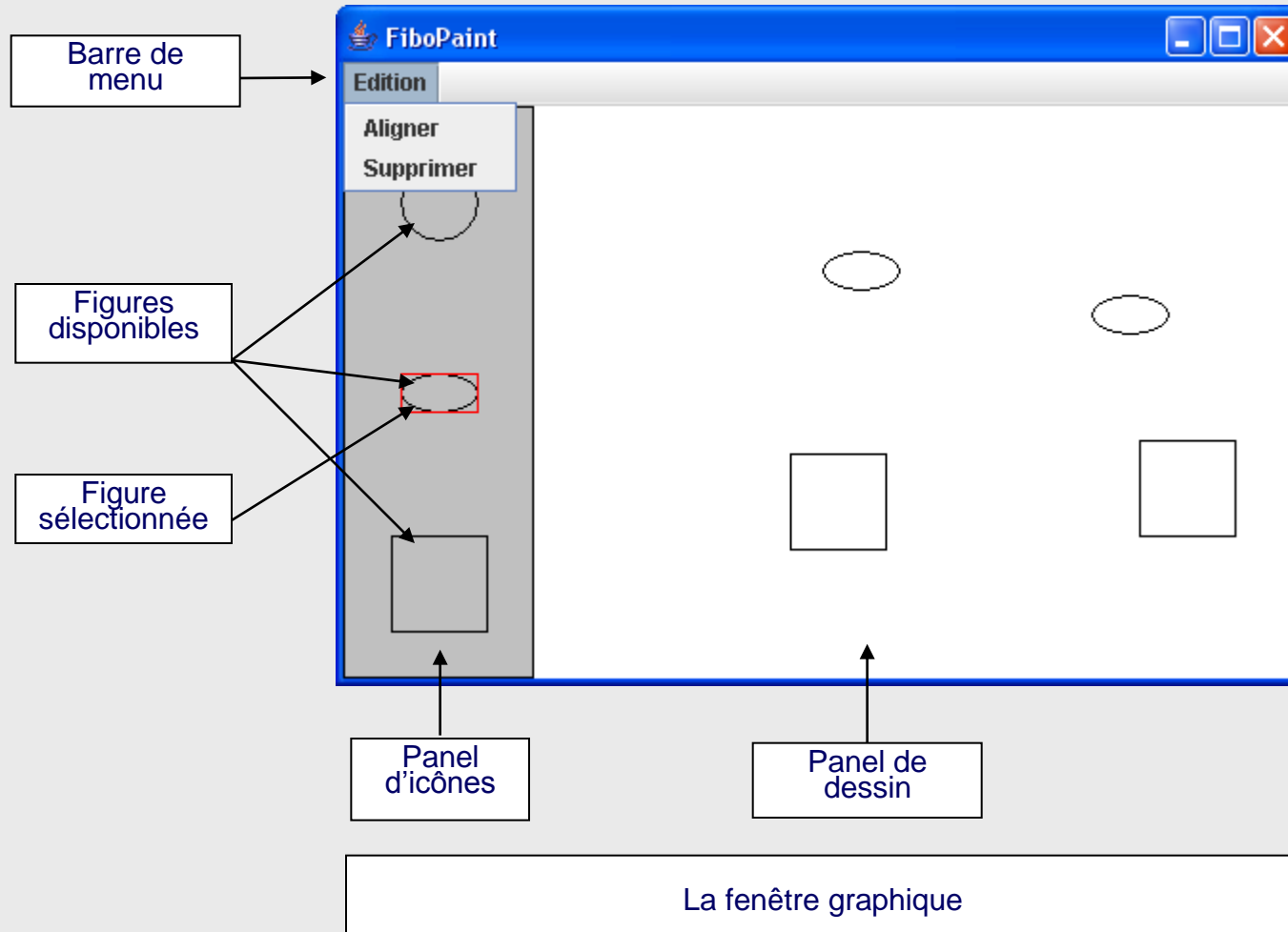
- ❖ Plusieurs délégués peuvent réagir à une même source



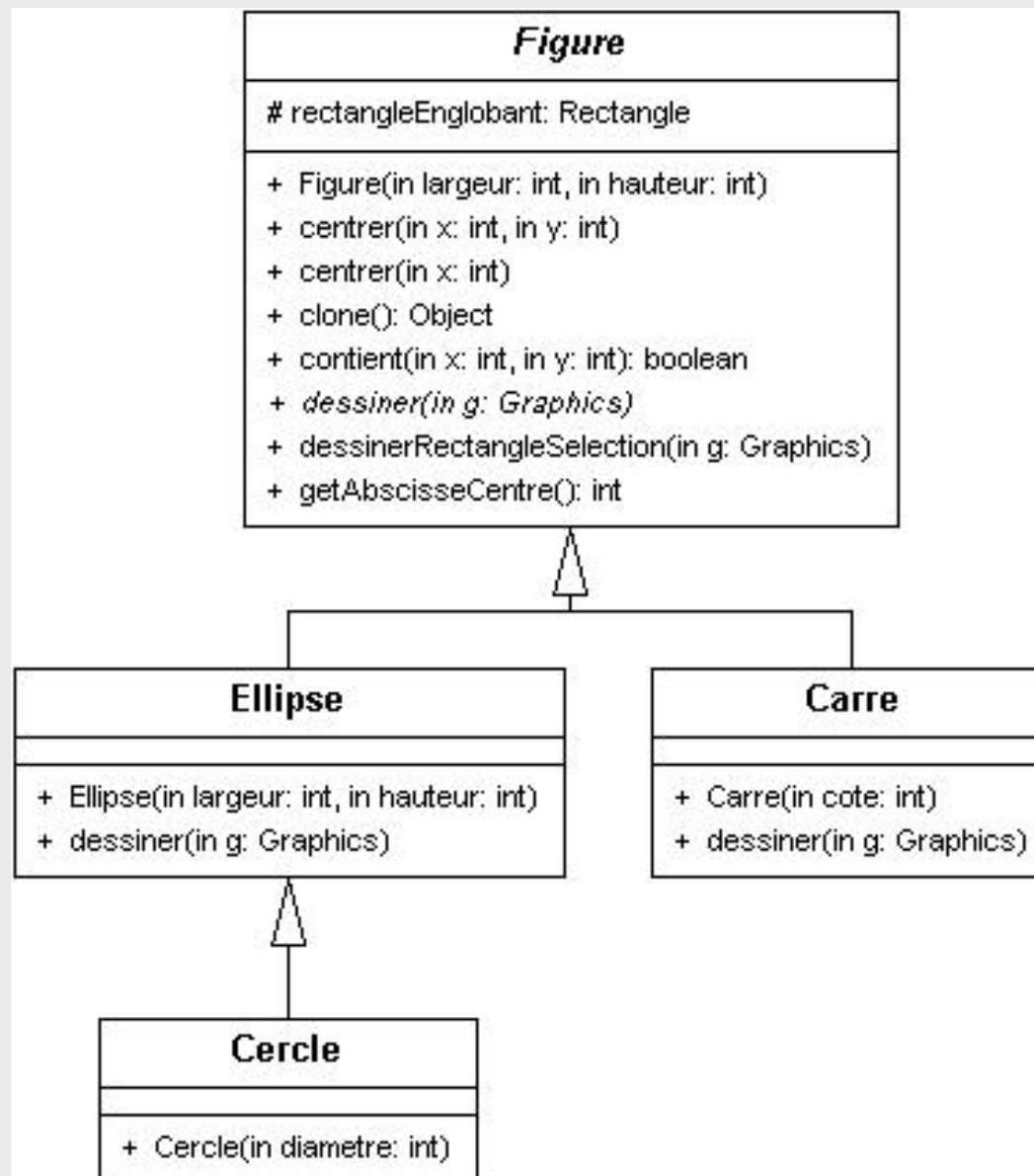
Chapitre 25

Exemple : TP / application graphique

❖ Vers un petit éditeur graphique



❖ Définition de classes de figures à partir de la classe abstraite Figure



❖ Exemple : La classe Ellipse

▪ Redéfinition de la méthode dessiner

```
public class Ellipse extends Figure{
```

```
/**
```

```
 * Constructeur
```

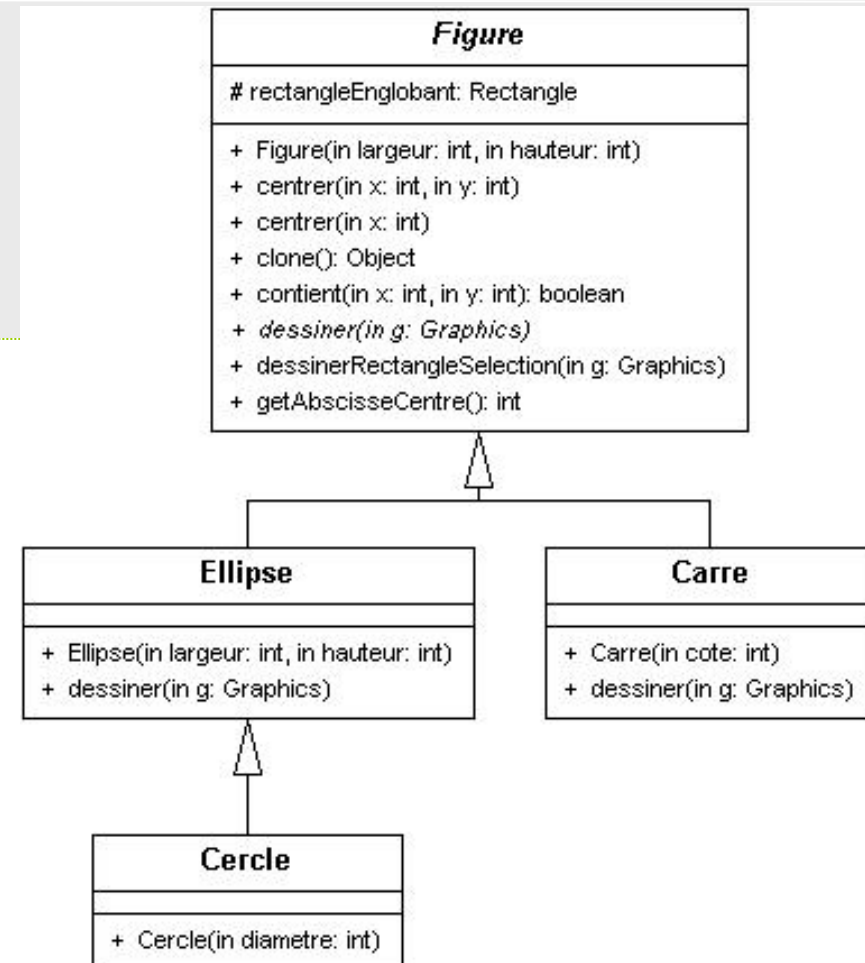
```
 * @param largeur largeur de l'ellipse
```

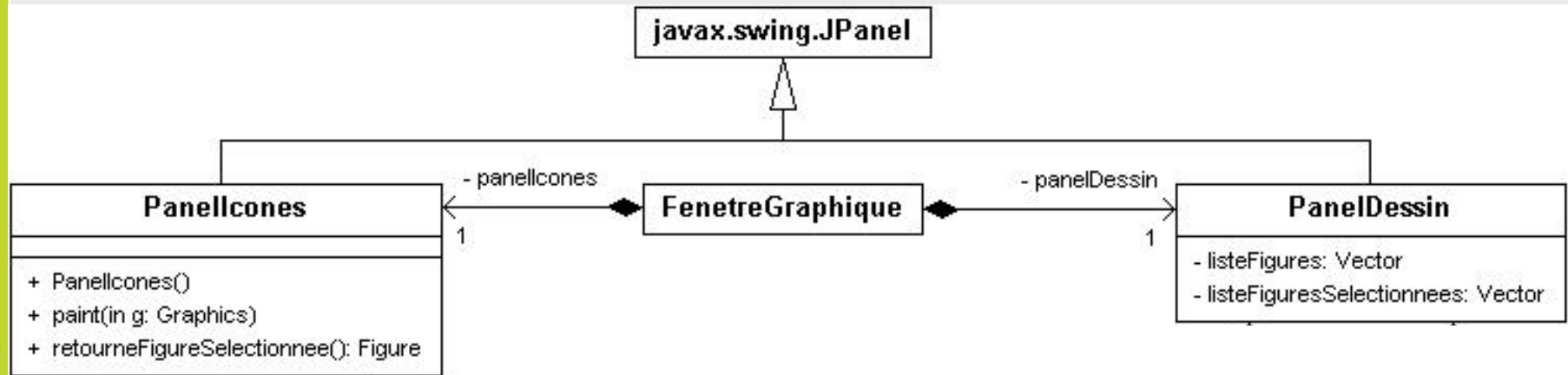
```
 * @param hauteur hauteur de l'ellipse
```

```
 */
```

```
public Ellipse(int largeur, int hauteur){
    super(largeur, hauteur);
}
```

```
public void dessiner(Graphics g){
    g.setColor(Color.BLACK); // couleur de tracé
    g.drawOval(rectangleEnglobant.x, rectangleEnglobant.y,
        rectangleEnglobant.width, rectangleEnglobant.height);
}
}
```





```

public class FenetreGraphique {
    // Panel support du dessin des figures
    private PanelDessin panelDessin;
    //Panel support des icones-figures
    private Panellcones panellcones;
    //...
}
    
```

```

public class Panellcones extends JPanel{
    // liste des figures-icones
    private Vector listeFigures;
    // Figure-icone sélectionnée
    private Figure figureSelectionnee;
    // ...
}
    
```

```

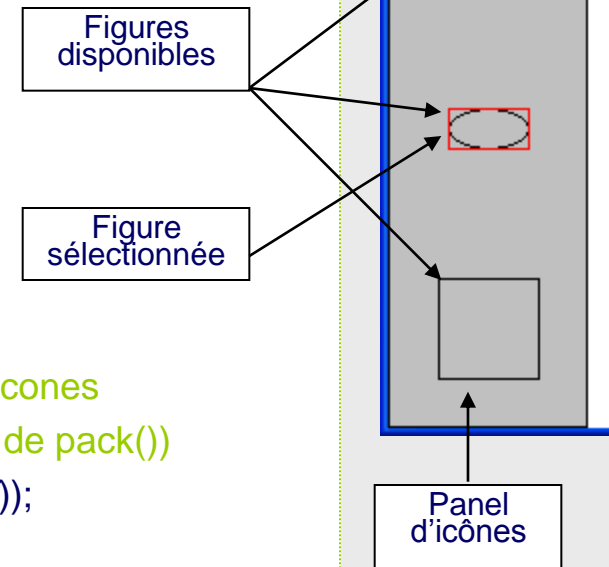
public class PanelDessin extends JPanel{
    // liste de toutes les figures du Panel de dessin
    private Vector listeFigures;
    // Liste des figures sélectionnées
    private Vector listeFiguresSelectionnees;
    // ...
}
    
```

```

public class Panellcones extends JPanel{
// Côté d'un icone
private static final int COTE_CASE = 100;
// ordonnée du centre du prochain icone ajouté
private int yCourant;
// liste des figures-icones
private Vector listeFigures;
// Figure-icone sélectionnée
private Figure figureSelectionnee;
public Panellcones(){
    setBackground(Color.LIGHT_GRAY); // fond du panel
    listeFigures = new Vector(); // initialisation de la liste des figures-icones
    // définition de la taille de référence du Panel (associé à la notion de pack())
    setPreferredSize(new Dimension(COTE_CASE, 3*COTE_CASE));
    // ajout des icones
    yCourant=COTE_CASE/2;
    ajouter(new Cercle(40));           // Cercle, Ellipse, ... figures graphiques
    ajouter(new Ellipse(40, 20));     // la méthode ajouter gère l'ajout dans la liste
    ajouter(new Carre(50));           // et la mise jour du décalage (yCourant)
    // figure sélectionnée par défaut
    figureSelectionnee = (Figure)listeFigures.get(0);

    // association d'un délégué pour gérer les évènements sur le Panellcone
    addMouseListener(new Deleguelcones());
}

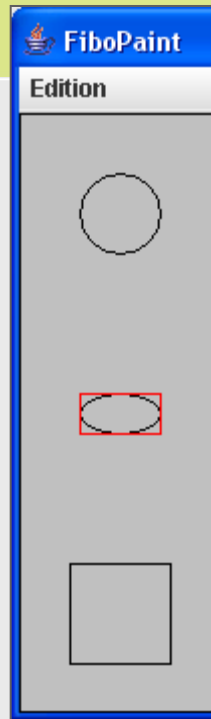
```



❖ La méthode paint(Graphics g)

- est appelée à chaque fois que la fenêtre à besoin d'être redessinée (g représente alors le bon contexte graphique pour pouvoir dessiner ...)

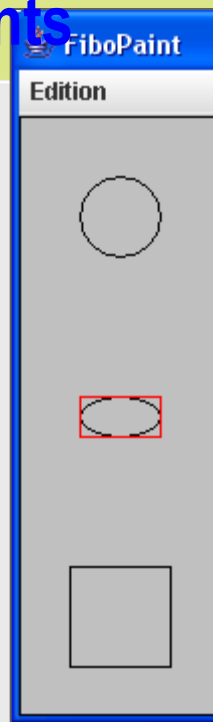
```
public class PanellIcones extends JPanel{
// Dessin du Panel
public void paint(Graphics g){
    super.paint(g); // pour effacer le fond (nécessaire pour les Jpanel)
    // dessin d'un cadre noir
    g.setColor(Color.BLACK);
    g.drawRect(getX(), getY(), getWidth()-1, listeFigures.size()*COTE_CASE-1);
    // dessin des 3 icones
    for(int i=0; i<listeFigures.size(); i++){
        Figure f = (Figure)listeFigures.get(i);
        f.dessiner(g);
        if(f==figureSelectionnee) //l'icone sélectionnée est entourée en rouge
            f.dessinerRectangleSelection(g);
    }
}
// Retourne la figure sélectionnée
public Figure retourneFigureSelectionnee(){ return figureSelectionnee; }
}
```



- ❖ **La classe interne privée déléguée** : Deleguelcones
 - Enregistrée dans le Main au près de Panellcone
 - Met à jour de l'icône sélectionnée dans le Panellcone en fonction des clics souris

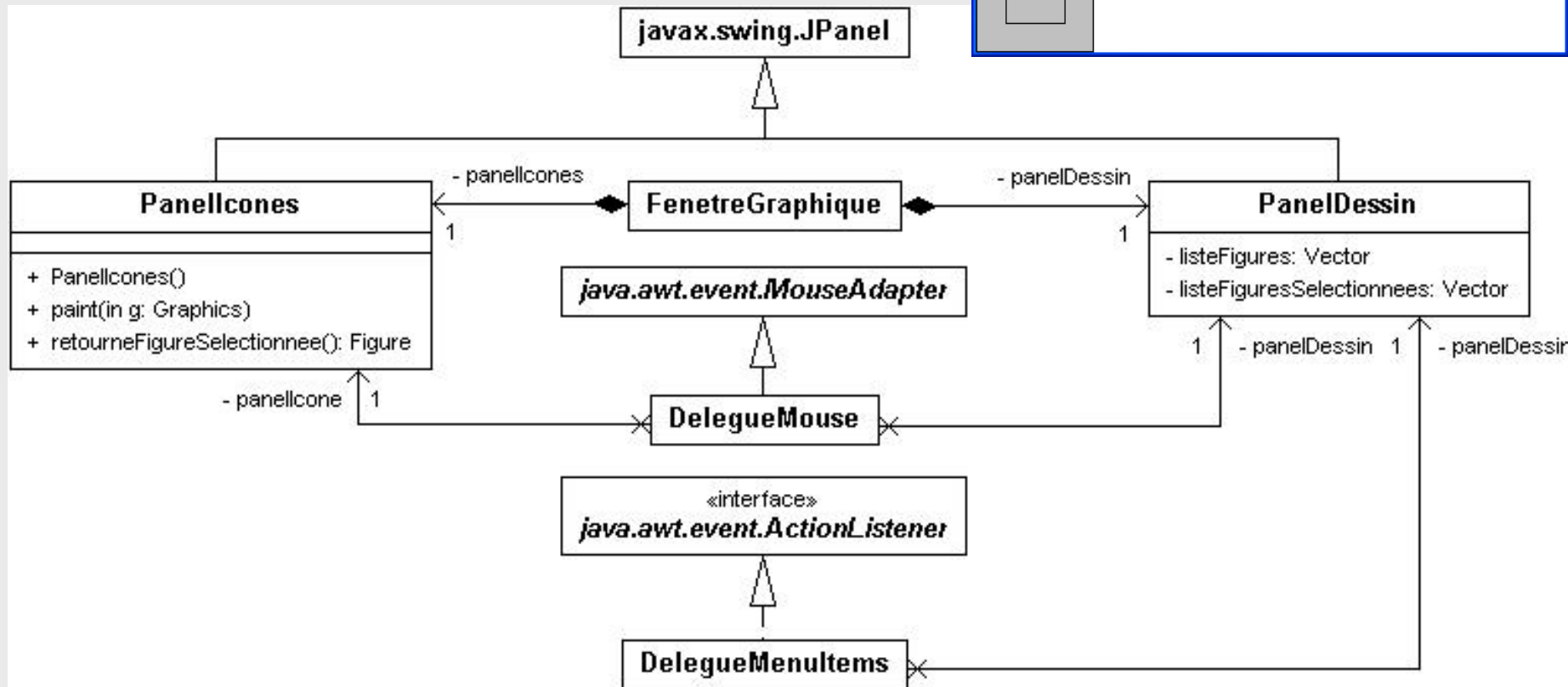
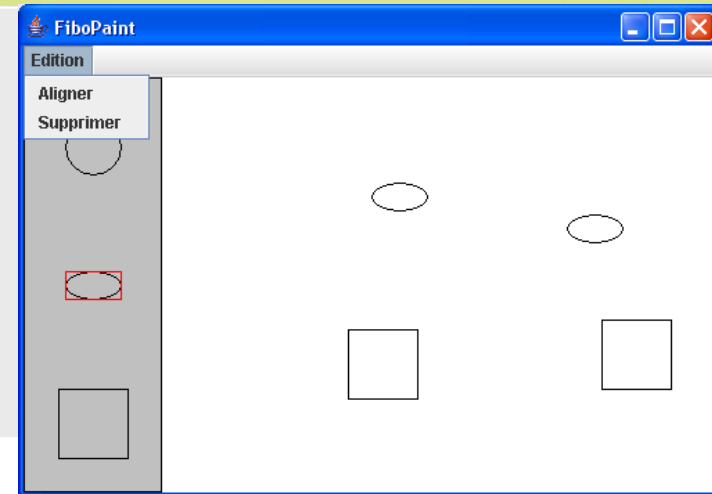
// Classe **interne** de Panellcône deleguee pour les événements souris

```
private class Deleguelcones extends MouseAdapter {
/**
 * Fonction appelée lors d'un clic souris
 * Sélectionne la figure cliquée
 */
public void mousePressed(MouseEvent e){
    Boolean stop=false;
    for(int i=0; i<listeFigures.size()&&!stop; i++){
        Figure f = (Figure)listeFigures.get(i);
        if(f.contient(e.getX(), e.getY())){
            figureSelectionnee = f;
            stop=true;
        }
    }
    repaint(); // demande un rafraichissement de l'écran => appel indirect au Paint
}
}
```



❖ Il reste à faire :

- **Le Panel Dessin : où l'on va dessiner les figures**
- **La fenêtre graphique qui va contenir**
 - Menu, Panel Dessin, Panel Icone
- **La gestion des événements sur le Panel Dessin: la classe DeleegueMouse**
 - gère les clics souris au niveau du panel de dessin :
 - ajout, par un clic dans une zone libre du panel de dessin, d'une figure du type de celle sélectionnée dans le panel d'icônes,
 - sélection/désélection d'une figure déjà présente dans le panel de dessin par un clic sur cette figure ;
 - *cette classe doit contenir une référence vers le panel de dessin pour pouvoir agir sur ses figures, ainsi qu'une référence vers le panel des icônes pour récupérer la figure à ajouter.*
- **La gestion des événements sur le Menu : la classe DeleegueMenuItems**
 - gère la sélection des items « supprimer » et « aligner » :
 - « supprimer » : suppression des figures sélectionnées du panel de dessin,
 - « aligner » : alignement selon l'axe vertical des figures sélectionnées du panel de dessin.
 - *cette classe doit contenir une référence vers le panel de dessin pour pouvoir agir sur ses figures.*



A decorative graphic consisting of several concentric, overlapping arcs in a light blue color, creating a sense of motion or a stylized 'C' shape, positioned behind the chapter title.

Chapitre 26

Introduction aux Applets

❖ Programme de type Applet

- programme Java destiné à fonctionner via un navigateur Web

❖ Sécurité

- ne peut pas accéder au système de fichier local (sauf applet signées)

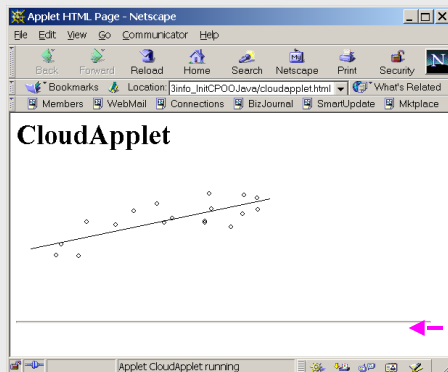
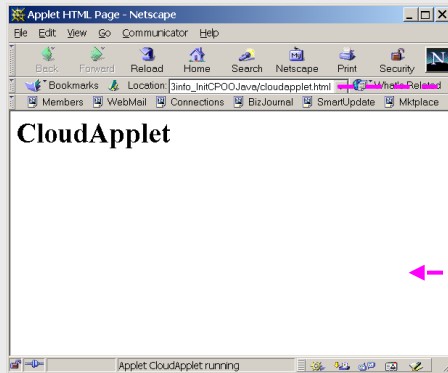
❖ Développement

- basé sur le package `java.applet` : un *framework* très simple
- le programmeur va
 - dériver certaines classes de base ;
 - surcharger certaines fonctionnalités ;
 - écrire ses fonctions spécifiques.
- la classe Applet ne possède pas de méthode `main()`

❖ Méthodes pour la manipulation d'une Applet (classe Applet)

- `init()` : est appelée au 1er chargement de l'Applet en mémoire : pour initialiser les structures de données de l'Applet
- `start()` : est appelée à chaque entrée dans la page Web (pour les animations, ...)
- `stop()` : est appelée quand on va quitter l'Applet
- `paint(Graphics g)` : est appelée à chaque fois que la fenêtre a besoin d'être redessinée (`g` représente alors le bon contexte graphique pour pouvoir dessiner ...)

Navigateur Web



Client
Possédant un
navigateur Web
avec JVM

Serveur
Qui va fournir
l'Applet

Demande de chargement
d'une page
associée à une Applet

1

Chargement de la page HTML
(CloudApplet.html)

2

Demande de chargement
de l'Applet
invoquée dans la page

3

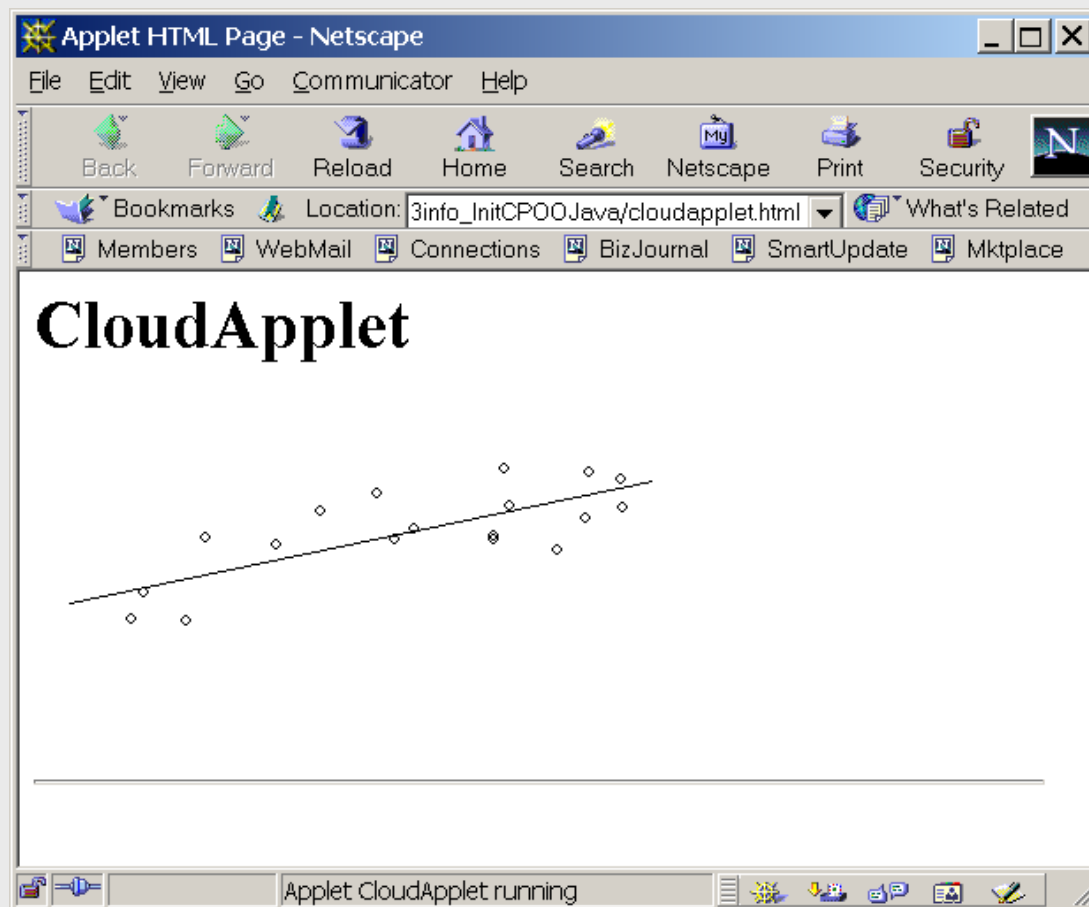
Chargement du byte-code
de l'Applet
(.class)

4

Exécution de l'Applet avec
la JVM du navigateur du client

❖ Réaction aux clics souris

- Création d'un point
- Calcul de la droite de régression
- Gestion de l'affichage



```

import java.applet.*; import java.awt.event.*; import java.awt.*; import java.util.*;

public class CloudApplet extends Applet implements MouseListener {
    private Vector _points = new Vector();
    public void init() {
        addMouseListener(this);
    }
    public void mousePressed(final MouseEvent e)
    {
        _points.addElement(new Point(e.getX(), e.getY()));
        repaint();
    }
    public void mouseReleased    (final MouseEvent p1) { }
    public void mouseEntered   (final MouseEvent p1) { }
    // ...

    public void paint(Graphics g)
    {
        for (int i = 0; i < _points.size(); i++)
        {
            Point p = (Point)_points.elementAt(i);
            g.drawOval(p.x - 2, p.y - 2, 5, 5);
        }
        if (regression())
        {
            int xright = getSize().width;
            g.drawLine(0, (int)b, xright, (int) (m * xright + b)); }
    }
    private boolean regression() {...};
}

```

Version compactée où la source
(Component ici l'applet) est
aussi le délégué (MouseListener)

Gestion des
événements

- ❖ **Création d'un Fichier HTML pour invoquer l'Applet**
- ❖ **Utilisation de l'appletviewer ou un browser Web pour exécuter l'Applet**
- ❖ **Exemple basic : attention plus compliqué avec les Japplet**
 - Cf. versions 1.2 de Java /plugins / IE / Netscape ...
 - Cf. <http://java.sun.com/products/plugin/1.2/docs/tags.html>

```
<HTML>
<HEAD>
  <TITLE> Applet HTML Page </TITLE>
</HEAD>

<BODY>
  <h1> CloudApplet </h1>
  <APPLET code="CloudApplet.class" width=350 height=200> </APPLET>
  <hr>
</BODY>
</HTML>
```

CloudApplet.html

Demo.html

```

import java.applet.*;import java.awt.event.*;import java.awt.*;import java.util.*;
public class CloudApplet extends Applet implements MouseListener {
    static int decal=1;    // pour observer les appels a paint
    private Vector _points = new Vector();
    private double m; private double b;
    public void init() {
        addMouseListener(this);
        decal=1;
    }
    public void paint(Graphics g) {
        for (int i = 0; i < _points.size(); i++) {
            Point p = (Point)_points.elementAt(i);
            g.drawOval(p.x - 2, p.y - 2, 5, 5);
        }
        decal+=1;
        if (regression()) {
            int xright = getSize().width;
            g.drawLine(0+decal, (int)b, xright, (int) (m * xright + b));
        }
    }
    private boolean regression() { ... }
    public void mousePressed(final MouseEvent e) { _
        points.addElement(new Point(e.getX(), e.getY()));
        repaint();
    }
    public void mouseReleased(final MouseEvent p1) { }
    public void mouseEntered(final MouseEvent p1) { }
    public void mouseClicked(final MouseEvent p1) { }
    public void mouseExited(final MouseEvent p1) { }
}

```

Exemple de mise en évidence des mécanismes sous tendus par le framework :

Exemple : Gestion automatique du rafraîchissement / décalage de la droite à chaque appel de la méthode paint.

Chapitre 27

Introduction aux génériques

❖ Objectifs

- Possibilité d'abstraire des types (on parle de types paramétrés)
- Exemple d'utilisation : invocation de types paramétrés
 - Amélioration de la lisibilité et de la robustesse

// avant java 5

```
List l1 = new ArrayList();
l1.add(new Integer(0));
Integer x = (Integer) l1.iterator().next();
l1.add(new Character('a'));
```

// liste d'Object
 // ajout d'un Integer qui est un Object
 // nécessité d'un cast : Object -> Integer
 // Possible !?: ajout d'un Character qui est un Object

// apres java 5

```
List<Integer> l2 = new ArrayList<Integer>();
l2.add(new Integer(0));
Integer x2 = l2.iterator().next();
```

// Liste d'Integer (le type de la liste a été spécifié : <Integer>)
 // ajout d'un Integer à une liste d'Integer : OK!
 // Plus besoin de Cast ! On est certain du type des éléments de la liste

```
l2.add(new Character('a'));
```

// Maintenant Impossible : vérification de type statique : à la compilation
 // Maintenant => The method add(Integer) in the type List<Integer>
 // is not applicable for the arguments (Character)

- ❖ On spécifie à la déclaration du « Generics » (classe paramétrée) les paramètres formels entre <>

```
public interface List<E> {           // E = paramètre formel de l'interface List
    void add(E x);
    Iterator<E> iterator();
}
public interface Iterator<E> {      // E = paramètre formel de l'interface Iterator
    E next();
    boolean hasNext();
}
```

❖ Principe des « Generics »

- Invocation d'une liste d'entier : `List<Integer>`

<=>

```
public interface IntegerList {
    void add(Integer x)
    Iterator<Integer> iterator();
}
```

Attention oui et NON

- Pas de duplication du code pour chaque type de paramètre utilisé

- Le « Generic » est compilé en une unique classe
 - Les paramètres formels sont remplacés à l'invocation du « Generic » par les paramètres effectifs

```
List<String> ls = new ArrayList<String>();  
List<Integer> li = new ArrayList<Integer>();  
  
System.out.println(ls.getClass() == li.getClass());  
System.out.println(ls.getClass().getName());  
System.out.println(li.getClass().getName());
```

=> réponse

```
true  
java.util.ArrayList  
java.util.ArrayList
```

❖ Héritage et generics

- Si **CD** hérite de **CB**
Alors **ClassG<CD>** n'hérite PAS de **ClasseG<CB>**

```
List<String> ls = new ArrayList<String>();
```

```
List<Object> lo = ls; // Type mismatch: cannot convert from List<String> to List<Object>
```

- Pourquoi ? Si c'était possible : ls et lo référenceraient la même liste !
 - donc

```
lo.add(new Object());
```

```
String s = ls.get(0); // Pb on essaie d'assigner un objet à une string !
```

- => ls pourrait contenir des objets n'étant pas des String !

❖ Utilisation du wildcard <?>

```
class Fenetre{
    public void draw(Figure s) { s.draw(this); }
```

```
    public void drawAll(List<Figure> Figures) {
        for (Figure s: Figures) { s.draw(this); }
    } // on ne peut passer en paramètre que des List<Figure>
```

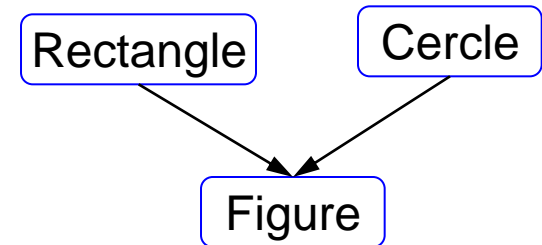
```
    public void drawAll1(List<?> Figures) { // wildcards
        for (Figure s: (List<Figure>)Figures) { s.draw(this); } // Cast ... dangereux !
    } // on peut passer en paramètre des List<de n'importe quel objet>
```

```
    public void drawAll2(List<? extends Figure> Figures) { // Bounded Wildcards
        for (Figure s: Figures) { s.draw(this); } // ok
    } // on peut passer en paramètre des List< Objet héritant de Figure>
```

```
    public static void main(String[] args) {
        Fenetre ca=new Fenetre();
        ca.draw(new Cercle());
        ca.drawAll(new ArrayList<Cercle>());

        ca.drawAll1(new ArrayList<Cercle>());
        ca.drawAll2(new ArrayList<Cercle>());
    }
}
```

// The method drawAll(List<Figure>) in the type Canvas
// is not applicable for the arguments (ArrayList<Cercle>)
// ok
// ok



❖ Recopie d'éléments d'une liste à l'autre

```
public static void recopie( List<? extends Rectangle> Figures,
                           List<Rectangle> Rectangles) {
    for (Rectangle r: Figures) { Rectangles.add(r); }
}
```

❖ Comment généraliser ?

```
public static void recopie1(List<?> l1, List<?> l2) {
    for (Object r: l1) { l2.add(r); }
}
```

// Erreur à la compilation !

❖ Utilisation d'une méthode générique

```
public static <T> void recopie2(List<? extends T> l1, List<T> l2) {
    for (T r: l1) { l2.add(r); } // ok !
}

public static <T,S extends T> void recopie3(List<S> l1, List<T> l2) {
    for (T r: l1) { l2.add(r); } // ok : mais préférer la solution précédente
}
```

- **Préférer l'utilisation des Wildcards : solution + concise et + claire !**

A decorative background graphic consisting of several concentric, overlapping circular arcs in a light blue color, creating a sense of motion or a stylized 'C' shape.

Chapitre 28

Remarques



- **Passer par une bonne analyse des concepts associés au problème avant toute implémentation**
 - analyse et conception objet, gestion de projet de type spirale,...
- **Chaque classe doit représenter un concept compact et bien défini**
 - éviter les objets trop gros, avec trop de méthodes,...
 - utiliser les mécanismes objets simples,...
- **Tester et Valider de manière unitaire vos objets**
 - faire des *main()* pour chaque classe
 - des exemples d'utilisation
- **Penser aux éléments essentiels d'une classe "type"**
 - *equals*,...
 - constructeur par recopie,...



- **Agrégation à héritage**
 - l'agrégation est bien souvent la meilleure option pour réutiliser des concepts
 - limiter l'héritage pour des modélisations adaptées
 - penser à la notion d'interface : simple, puissante et utile !
- **Respecter une convention de notation pour le nommage des variables**
 - **UneClasse** **uneMethode** *// en Java*
- **A la conception d'une classe penser à l'utilisateur et à la maintenance**
 - encapsuler, protéger, commenter,...
 - offrir une gestion complète des erreurs: exceptions,...
- **Documenter vos programmes, concepts, analyses**
 - Doc++, javadoc,...

Chapitre 29

Annexe : Java 5 (Tiger)

❖ Automne 2004

- Révision de Java : Java 5 (nom de code Tiger) : J2SE 5.0

❖ Les Principales évolutions

- Autoboxing et Auto-Unboxing des types Primitifs
- Les itérations sur des collections
- Type énuméré : enum
- Nombre d'arguments variable pour une méthode
- Les imports statiques
- Les Generics – introduction
- ... les Annotations, Supervision de la JVM, Synchronisation, Client lourd

❖ Intégration avec Eclipse

- Nécessité de passer à la version Eclipse 3.1 pour exploiter les nouvelles fonctionnalités de Java5
 - Preferences>java>compiler>Compiler compliance level : 5.0

❖ Simplification du passage (Type Primitifs / objet)

- boolean/Boolean, byte/Byte, double/Double, short/Short, int/Integer, long/Long, float/Float, char/Character

❖ Exemple

```
Vector tabDyn=new Vector();
```

// avant Java 5

```
Character oCar = new Character('a');
```

// Objet

```
char car=oCar.charValue();
```

// unBoxing -> type primitif

```
tabDyn.addElement(new Character(car));
```

// boxing -> Objet

// maintenant avec Java 5

```
Character oCar2 = new Character('a');
```

// objet

```
char car2=oCar2;
```

// autoUnBoxing

// maintenant ok

// avant => Type mismatch: cannot convert from Character to char

```
tabDyn.addElement(car2);
```

// autoBoxing

// maintenant ok

// avant => The method addElement(Object) in the type Vector

// is not applicable for the arguments (char)

❖ Evolution du for pour l'itération sur des collections

- For (paramètre formel : Objet Iterable) { corps }
- Notation plus légère et concise

```
List Figures=new ArrayList();  
Figures.add(new Rectangle());
```

//...

// avant java5

```
for (Iterator i = Figures.iterator(); i.hasNext(); ) {  
    ((Figure)i.next()).draw(ca);  
}
```

// avec Java 5

```
for (Object s: Figures) {  
    ((Figure)s).draw(ca);  
}
```

- NB: On ne peut pas modifier la collection avec cette notation

❖ Avant Java 5 : utilisation de constante de type int

```
public class Etapes {  
    public static final int ETAPE0=0;  
    public static final int ETAPE1=1;  
    public static final int ETAPE2=2;  
    //...  
}
```

```
int e=Etapes.ETAPE1;  
System.out.println(e);    // affichage : 1  
e=10;                     // possible ! Danger !
```

❖ Avec Java5 : la construction des enum est type-safe

```
public enum EtapesJava5 { ETAPE0, ETAPE1, ETAPE2 };
```

```
EtapesJava5 e2=EtapesJava5.ETAPE0;  
System.out.println(e2);    // affichage : ETAPE0  
e2=10;                     // erreur (type-safe)Type mismatch:  
                           // cannot convert from int to Canvas.EtapesJava5
```

❖ D'autres raffinements sont encore possibles ...

❖ Notation : « Type ... »

```
static void argTest(String ... args) { // nombre variable d'arguments de type String
    for (String o : args) {
        System.out.println(o);
    }
}

// ...

argTest("arg1","arg2","arg3"); // 3 paramètres
argTest("bonjour","monsieur"); // 2 Paramètres
```

❖ Résultat

```
arg1
arg2
arg3
bonjour
monsieur
```

❖ **Similaire à la fonction « C » printf !**

```
float d1=1.1f;  
float d2=2f;
```

```
System.out.printf("d1= %f  et  d2= %1.2f \n", d1,d2);
```

❖ **Résultat**

```
d1= 1,100000 et d2= 2,00
```

❖ **Formatage avancé**

- Cf. `java.util.Formatter`

❖ Sur l'entrée standard / Méthode next : blocante !

```
Scanner s= new Scanner(System.in);
String entreString=s.next();
int entreInt=s.nextInt();
System.out.printf("entreString= %s et entreInt= %d \n",entreString,entreInt);
s.close();
```

< maChaine

< 2005

> entreString= maChaine et entreInt= 2005

❖ Sur un fichier

```
Scanner sc = null;
try { sc = new Scanner(new File("sequence_ordres")); }
catch (java.io.FileNotFoundException e) { System.out.println(e); }
while (sc.hasNextInt()) { // est-ce qu'il y a un nouvel élt (token) à analyser ?
    System.out.println(sc.nextInt()); // on récupère l'élt suivant (token)
}
```

❖ Définition de délimiteurs // StringTokenizer

```
String entree="s-a-l-u-t";
Scanner s2= new Scanner(entree).useDelimiter("\\s*-\\s*");
String sortie = s2.next() + s2.next()+ s2.next()+ s2.next()+ s2.next();
s2.close();
System.out.printf(sortie);
```

salut

❖ Objectifs

- **Rendre visible des méthodes et variables statiques**

// Avant

```
getContentPane().add(plum, BorderLayout.CENTER);
```

// Avec Java5 et les imports statiques

```
Import static java.awt.BorderLayout.*  
getContentPane().add(plum, CENTER);
```

- **Attention aux conflits de nommage**