Programmation Orientée Objet application au langage Java

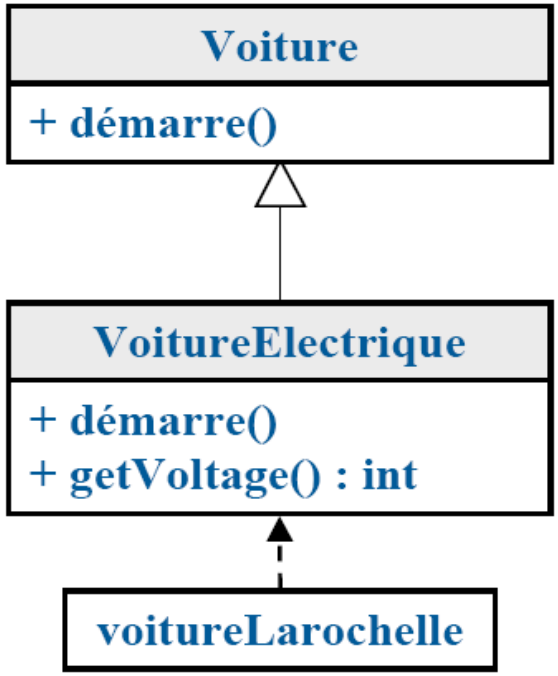
Héritage et Polymorphisme

Salima Hassas

Transparents de Mickaël BARON (SERLI Informatique)

Définition du polymorphisme

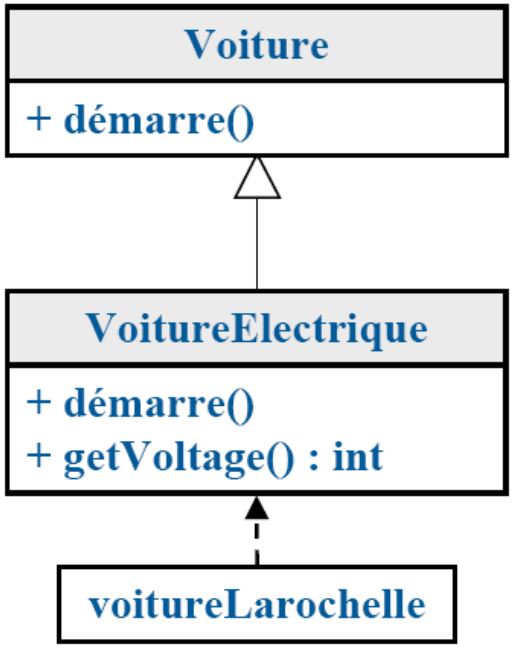
- Définition
  - Un langage orienté objet est dit polymorphique, s'il offre la possibilité de pouvoir percevoir un objet en tant qu'instance de classes variées, selon les besoins
  - **Une classe B qui hérite de la classe A peut être vu comme un sous-type du type défini par la classe A**



- Rappel
  - voitureLarochelle est une instance de la classe *VoitureElectrique*
- Mais aussi
  - voitureLarochelle est une instance de la classe *Voiture*

Polymorphisme et Java : surclassement

- Java est polymorphique
  - A une référence déclarée de la classe *Voiture*, il est possible d'affecter une valeur qui est une référence vers un objet de la classe *VoitureElectrique*
  - On parle de **surclassement** ou **upcasting**
  - Plus généralement à une référence d'un type donné, soit A, il est possible d'affecter une valeur qui correspond à une référence vers un objet dont le type effectif est n'importe quelle sous classe directe ou indirecte de A



Objet de type sous-classe directe de Voiture

```
public class Test {
    public static void main (String[] argv) {

        Voiture voitureLarochelle =
            new VoitureElectrique(...);
    }
}
```

Polymorphisme et Java : surclassement

- A la compilation
  - Lorsqu'un objet est « surclassé », il est vu par le compilateur comme un objet du type de la référence utilisée pour le désigner
  - Ses fonctionnalités sont alors restreintes à celles proposées par la classe du type de la référence

```
public class Test {
    public static void main (String[] argv) {

        // Déclaration et création d'un objet Voiture
        Voiture voitureLarochelle = new VoitureElectrique(...);

        // Utilisation d'une méthode de la classe Voiture
        voitureLarochelle.demarre();

        // Utilisation d'une méthode de la classe VoitureElectrique
        System.out.println(voitureLarochelle.getVoltage()); // Erreur
    }
}
```

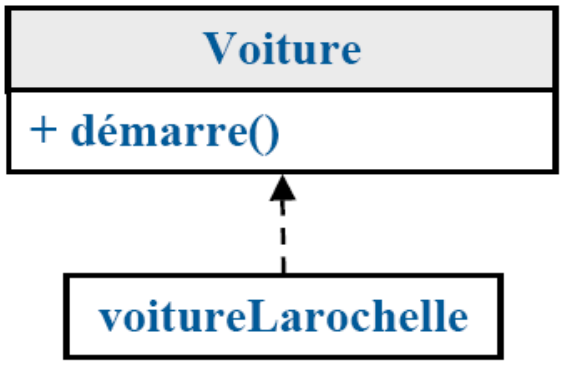
Examiner le type de la référence

La méthode *getVoltage()* n'est pas disponible dans la classe Voiture!!!

Polymorphisme et Java : surclassement

```
public class Test {
    public static void main (String[] argv) {
        // Déclaration et création d'un objet Voiture
        Voiture voitureLarochelle = new VoitureElectrique(...);

        // Utilisation d'une méthode de la classe Voiture
        voitureLarochelle.demarre();

        // Utilisation d'une méthode de la classe VoitureElectrique
        System.out.println(voitureLarochelle.getVoltage());
    }
}
```

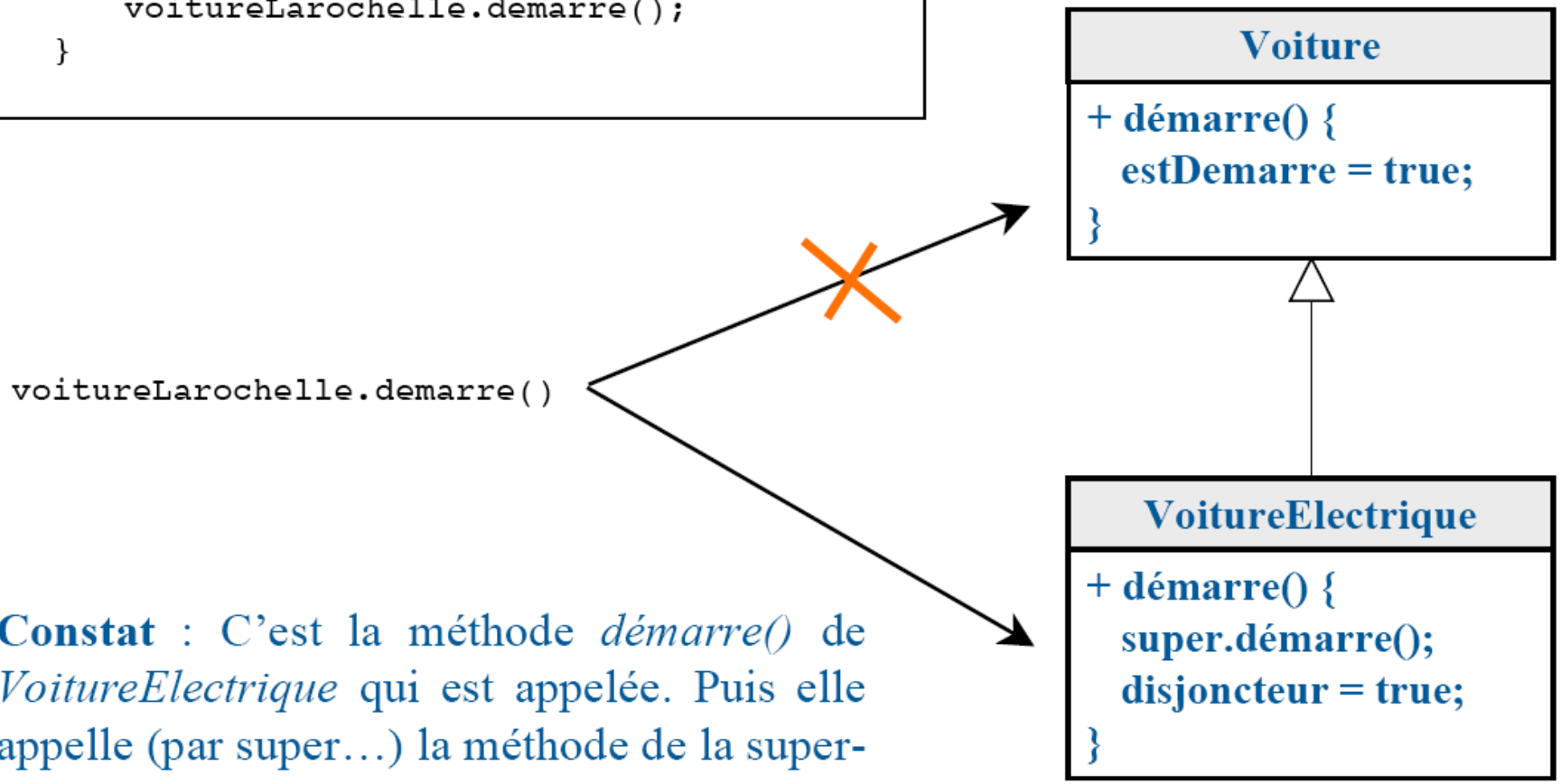
Remarque : Quel code va être effectivement exécuté lorsque le message *demarre()* est envoyé à voitureLarochelle ??



Polymorphisme et Java : lien dynamique

```
public class Test {
    public static void main (String[] argv) {
        Voiture voitureLarochelle =
            new VoitureElectrique(...);

        voitureLarochelle.demarre();
    }
}
```

L'objet voitureLarochelle initialise les attributs de la classe *VoitureElectrique*



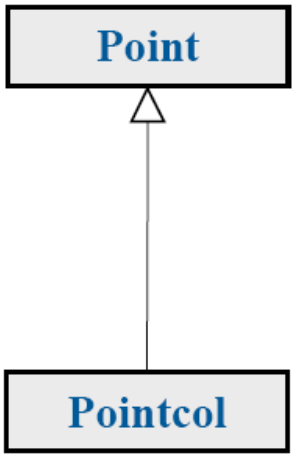
Constat : C'est la méthode *démarre()* de *VoitureElectrique* qui est appelée. Puis elle appelle (par super…) la méthode de la super-classe

www.serli.com

Polymorphisme et Java : lien dynamique

```
public class Point {
    private int x,y;
    public Point(int x, int y) { this.x = x; this.y = y; }
    public void deplace(int dx, int dy) { x += dx; y+=dy; }
    public void affiche() { System.out.println("Je suis en "+ x + " " + y);}
}
```

```
public class Pointcol extends Point {
    private byte couleur;
    public Pointcol(int x, int y, byte couleur) {
        super(x,y);
        this.couleur = couleur;
    }
    public void affiche() {
        super.affiche();
        System.out.println("et ma couleur est : " + couleur);
    }
}
```



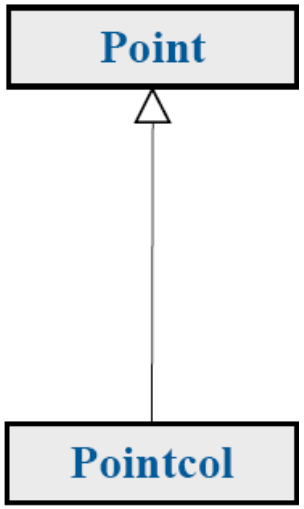

```
public class Test {
    public static void main (String[] argv)        {
        Point p = new Point(23,45);
        p.affiche();
        Pointcol pc = new Pointcol(5,5,(byte)12);
        p = pc;
        p.affiche();
        p = new Point(12,45);
        p.affiche();
    }
}
```

www.serli.com
serli informatique

Polymorphisme et Java : lien dynamique

```
public class Point {
    private int x,y;
    public Point(int x, int y) { this.x = x; this.y = y; }
    public void deplace(int dx, int dy) { x += dx; y+=dy; }
    public void affiche() {
        this.identifie();
        System.out.println("Je suis en "+ x + " " + y);
    }
    public void identifie() {System.out.println("Je suis un point");}
}
```

```
public class Pointcol extends Point {
    private byte couleur;
    public Pointcol(int x, int y, byte couleur) {...}
    public void affiche() {
        super.affiche();
        System.out.println("et ma couleur est : " + couleur);
    }
    public void identifie() {System.out.println("Je suis un point coloré");}

}
```




```
public class Test {
    public static void main (String[] argv)        {
        Point p = new Point(23,45);
        p.affiche();
        Pointcol pc = new Pointcol(5,5,(byte)12);
        p = pc;
        p.affiche();
        p = new Point(12,45);
        p.affiche();
    }
}
```

www.serli.com
serli informatique

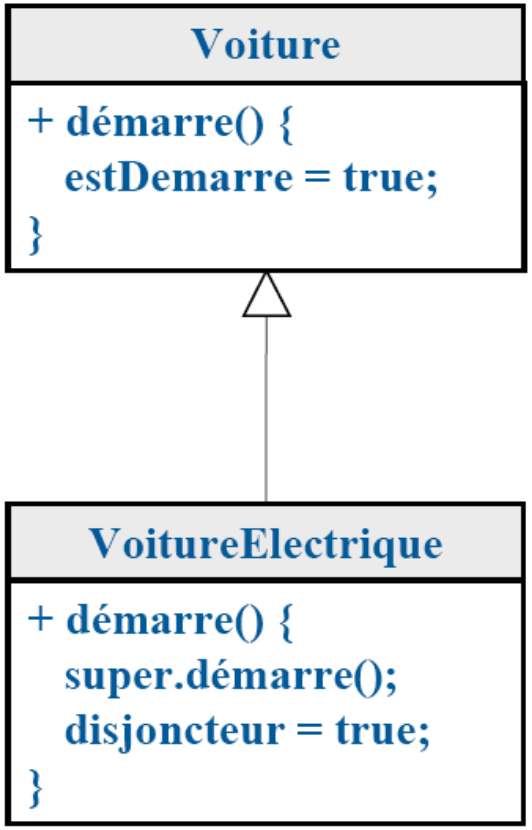
Polymorphisme et Java : lien dynamique

- A l'exécution
  - Lorsqu'une méthode d'un objet est accédée au travers d'une référence « surclassée », c'est la méthode telle qu'elle est définie au niveau de la classe effective de l'objet qui est invoquée et exécutée
  - La méthode à exécuter est déterminer à l'exécution et non pas à la compilation
  - On parle de liaison tardive, lien dynamique, dynamic binding, late-binding ou run-time binding

Polymorphisme et Java : bilan

```
public class Test {
    public static void main (String[] argv) {
        Voiture maVoit = new VoitureElectrique(...);
        maVoit.démarre();
    }
}
```

- Surclassement (compilation)
  - Une variable *maVoit* est déclarée comme étant une référence vers un objet de la classe *Voiture*
  - Un objet de la classe *VoitureElectrique* est créé
  - Pour le compilateur *maVoit* reste une référence d'un objet de la classe *Voiture*, et il empêche d'accéder aux méthodes spécifiques à *VoitureElectrique*
- Liaison dynamique (exécution)
  - Une variable *maVoit* est bel et bien une référence vers un objet de la classe *VoitureElectrique*



www.serli.com
serli informatique

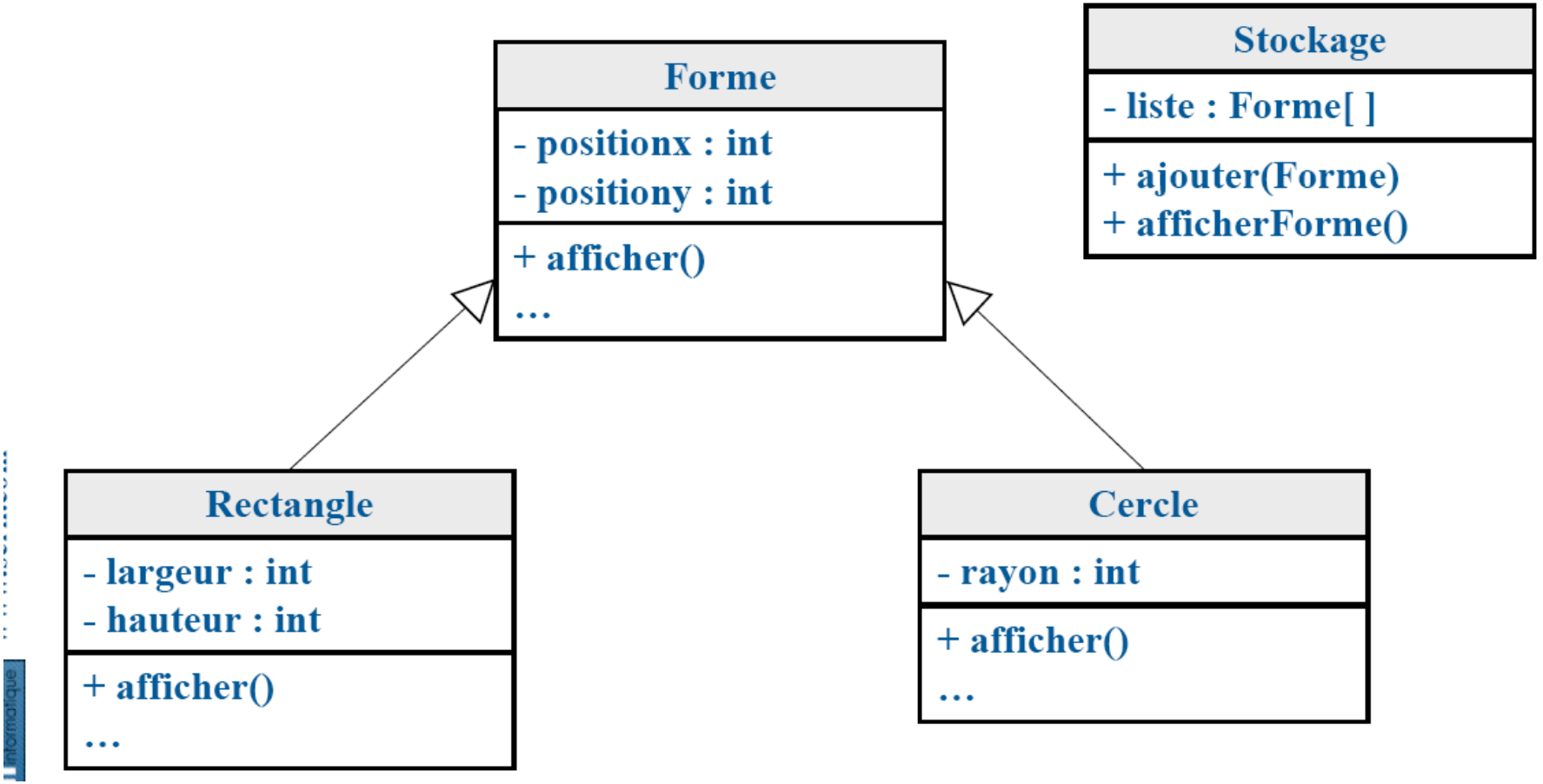
Polymorphisme : ok, mais pourquoi faire ?

- Que des avantages …
  - Plus besoin de distinguer différents cas en fonction de la classe des objets
  - Le polymorphisme constitue la troisième caractéristique essentielle d'un langage orienté objet après l'abstraction des données (encapsulation) et l'héritage
  - Une plus grande facilité d'évolution du code. Possibilité de définir de nouve fonctionnalités en héritant de nouveaux types de données à partir d'une clas de base commune sans avoir besoin de modifier le code qui manipule la cla de base
  - Développement **plus rapide**
  - Plus grande **simplicité** et **meilleure organisation** du code
  - Programmes plus facilement **extensibles**
  - Maintenance du code **plus aisée**

Informatique

Polymorphisme : un exemple typique

- Exemple sur la géométrie
  - Stocker des Forme de n'importe quel type (Rectangle ou Cercle) puis les afficher



Polymorphisme : un exemple typique

```
public class Stockage {
    private Forme[] liste;
    private int taille;
    private int i;

    public Stockage(int taille) {
        this.taille = taille;
        liste = new Forme[this.taille];
        i = 0;
    }

    public void ajouter(Forme f) {
        if (i < taille) {
            liste[i] = f;
            i++;
        }
    }

    public void afficherForme() {
        for (int i = 0; i < taille; i
            liste[i].afficher();
        }
    }
}
```

Si une nouvelle type de Forme est défini, le code de la classe *Stockage* n'est pas modifié

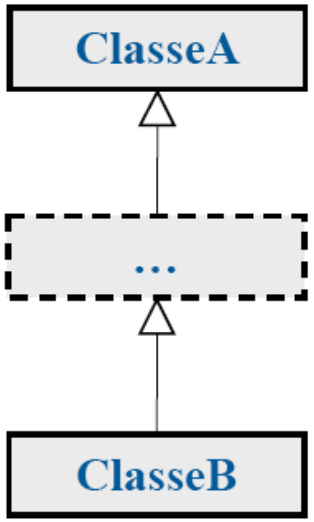
```
public class Test {
    public static void main (String[] argv) {
        Stockage monStock = new Stockage(10);
        monStock.ajouter(new Cercle(...));
        monStock.ajouter(new Rectangle(...));

        Rectangle monRect = new Rectangle(...);
        Forme tonRect = new Rectangle(...);
        monStock.ajouter(monRect);
        monStock.ajouter(tonRect);
    }
}
```

www.serli.com
erli informatique

Polymorphisme : downcasting

- Intérêt
  - Force un objet à « libérer » les fonctionnalités cachés par le surclassement
  - On parle de conversion de type explicite (cast). Déjà vu pour les types primitif

```
ClasseA monObj = ...
ClasseB a = (ClasseB) monObj
```



  - Pour que le « cast » fonctionne, il faut qu'à l'exécution le type effectif de *monObj* soit « compatible » avec le type ClasseB

  - Compatible : on peut tester la compatibilité par le mot clé **instanceof**

```
obj instanceof ClasseB
```

Retourne vrai ou faux

www.serli.com
SERLI informatique

Polymorphisme : downcasting

Exemple

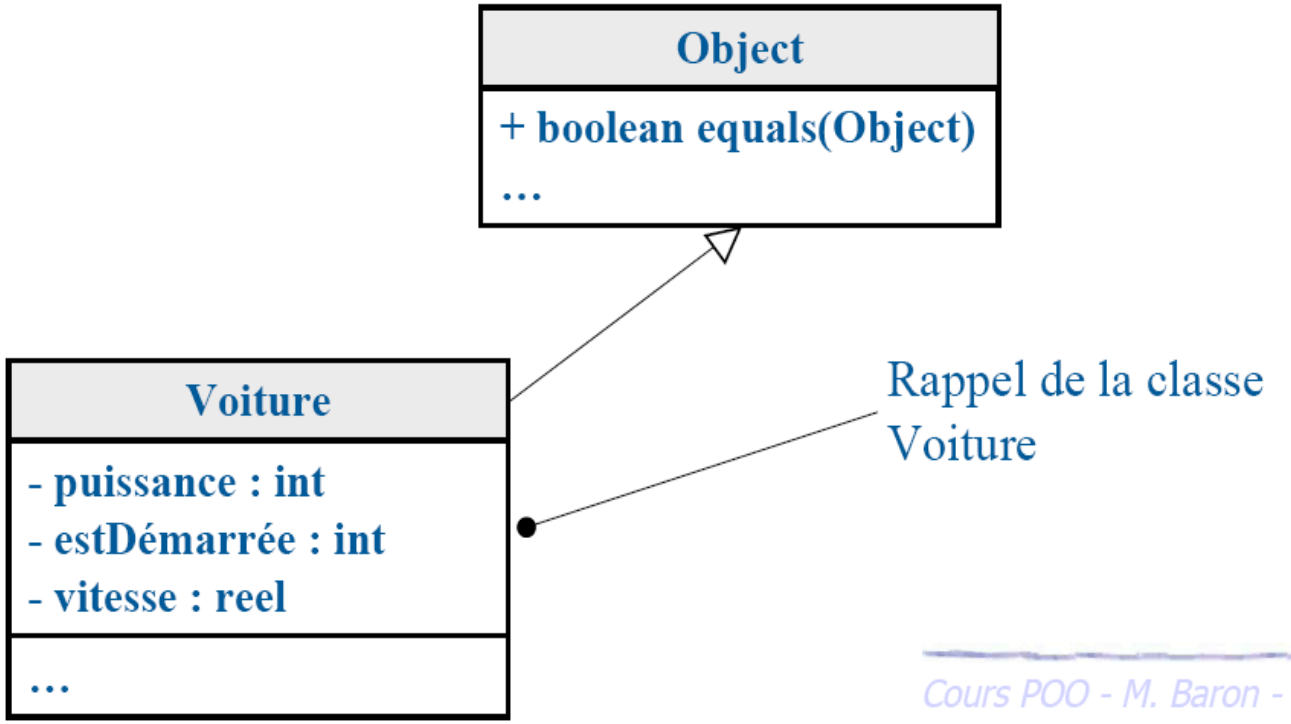
```
public class Test {
    public static void main (String[] argv) {
        Forme maForme = new Rectangle();
        // Je ne peux pas utiliser les méthodes de la classe Rectangle

        // Déclaration d'un objet de type Rectangle
        Rectangle monRectangle;
        if (maForme instanceof Rectangle) {
            monRectangle = (Rectangle)maForme;
            // Utilisation possible des méthodes spécifiques de Rectangle
        }
    }
}
```

Réalisation de la conversion de l'objet de type Forme en objet de type Rectangle

Attention si la compatibilité est fausse et si le cast est effectué une exception de type *ClassCastException* est levée

La méthode « equals() »

- Deux possibilités pour comparer un objet de ses classes
  - Créer une méthode ad-hoc « public boolean comparer(MaClasse c) {…} » qui compare les attributs
  - Redéfinir la méthode « public boolean equals(Object o) » pour garder la compatibilité avec les autres classes de Java.
    - Re-implémenter la méthode « public boolean equals(Object o) » en comparant les attributs (en utilisant une conversion de type explicite)



www.serli.com
serli informatique

La méthode « equals() »

```
public class Voiture extends Object {
    public boolean equals(Object o) {
        if (!o instanceof Voiture) {
            return false;
        }

        Voiture maVoit = (Voiture)o;
        return this.puissance == maVoit.puissance && this.estDémarrée ==
            maVoit.estDémarrée && this.vitesse == maVoit.vitesse;
    }
    ...
}
```

Redéfinition de la méthode equals de la classe Object

Même valeurs d'arguments

```
public class Test {
    public static void main (String[] argv) {
        Voiture maVoit = new Voiture(...);
        VoitureElectrique maVoitele = new VoitureElectrique(...);

        maVoit.equals(maVoitele); --> TRUE
    }
}
```

Attention : l'égalité de référence == vérifie si les références sont les mêmes, cela ne compare pas les attributs

www.serli.com
serli informatique

Retour sur Static

Une colle

- La méthode **`main()`** est nécessairement **`static`.**
  Pourquoi ?

Une colle

- La méthode **`main()`** est nécessairement **`static`**. Pourquoi ?

- ⇒ La méthode **`main()`** est exécutée au début du programme. Aucune instance n'est donc déjà créée lorsque la méthode **`main()`** commence son exécution. Ça ne peut donc pas être une méthode d'instance.

Blocs d'initialisation `static`

- Les blocs **`static`** permettent d'initialiser les variables **`static`** trop complexes à initialiser dans leur déclaration :

```
class UneClasse {
  static int[] unTableau = new int[25];
  static {
    for (int i = 0; i<25; i++) {
      . . . // initialisation de unTableau
    }
  } // fin du bloc static
  . . .
```

- Ils sont exécutés une seule fois, quand la classe est chargée en mémoire

Classes abstraites : intérêts

- On ne connaît pas toujours le comportement par défaut d'une opération commune à plusieurs sous-classes
  - Exemple : toit d'une voiture décapotable. On sait que toutes les décapotables peuvent ranger leur toit, mais le mécanisme est différent d'une décapotable à l'autre
  - Solution : on peut déclarer la méthode « abstraite » dans la classe mère et ne pas lui donner d'implantation par défaut

- Méthode abstraite et conséquences : 3 règles à retenir
  - Si une seule des méthodes d'une classe est abstraite, alors la classe devient aussi abstraite
  - On ne peut pas instancier une classe abstraite car au moins une de ses méthodes n'a pas d'implémentation
  - Toutes les classes filles héritant de la classe mère abstraite doivent implémenter toutes ses méthodes abstraites ou sinon elles sont aussi abstraites

www.serli.com
serli informatique

Classes abstraites et Java

- On utilise le mot clé **abstract** pour spécifier abstraite une classe
- Une classe abstraite se déclare ainsi :

```
public abstract class NomMaClasse {
    ...
}
```

- Une méthode abstraite se déclare ainsi :

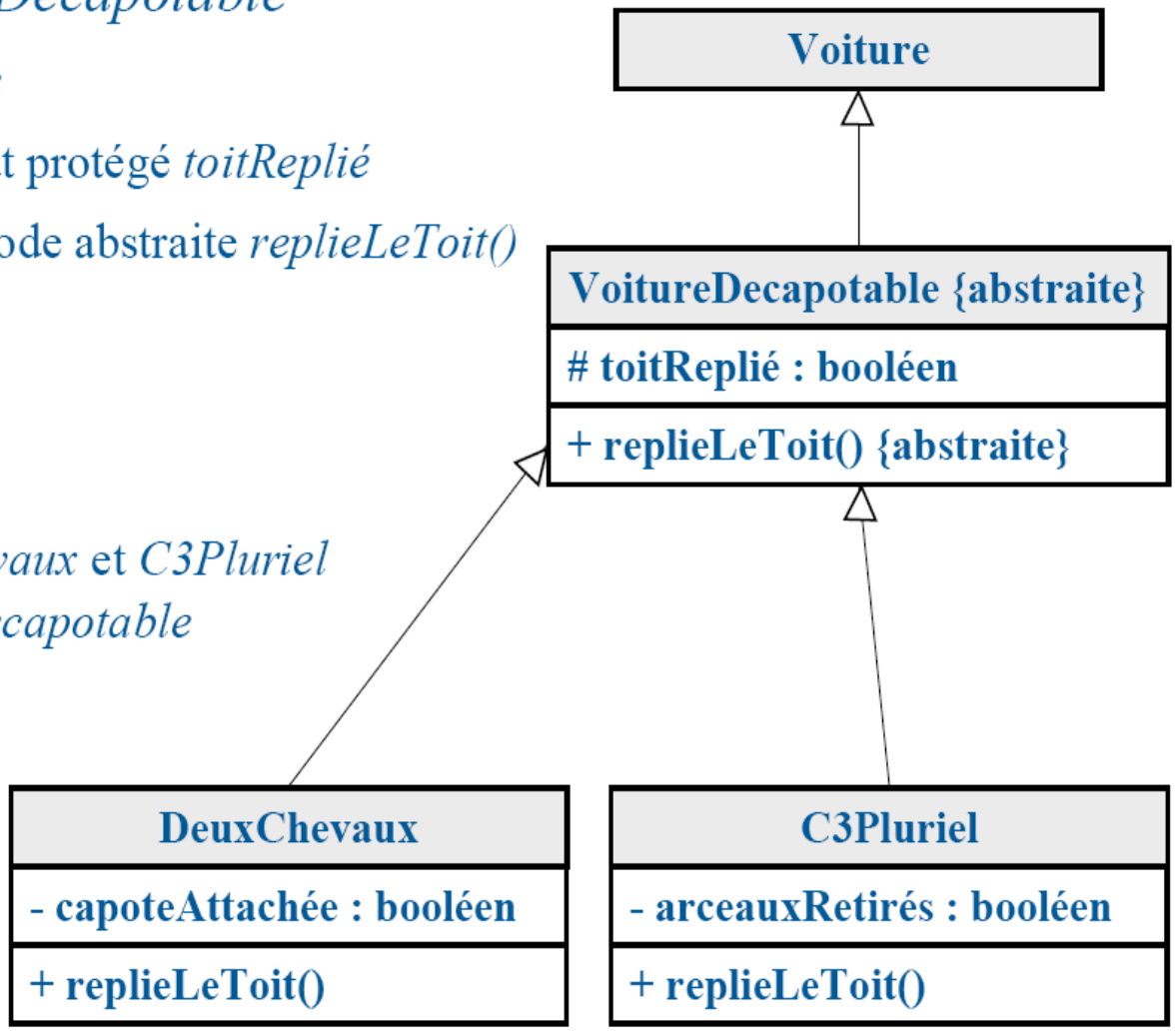
```
public abstract void maMethode(...);
```

Pour créer une méthode abstraite, on déclare sa signature (nom et paramètres) sans spécifier le corps et en ajoutant le mot clé **abstract**

www.serli.com
serli informatique

Classes abstraites : exemple VoitureDecapotable

- La classe *VoitureDecapotable*
  - Hérite de *Voiture*
  - Définit un attribut protégé *toitReplié*
  - Définit une méthode abstraite *replieLeToit()*



- Les classes *DeuxChevaux* et *C3Pluriel* héritent de *VoitureDecapotable*

www.serli.com
serli informatique

Classes abstraites : exemple VoitureDecapotable

Classe abstraite

Méthode abstraite

```
public abstract class VoitureDecapotable
    extends Voiture {
    protected boolean toitReplié;

    public abstract void replieLeToit();
}
```

```
public class DeuxChevaux extends VoitureDecapotable {
    private boolean capoteAttachee;

    public void replieLeToit() {
        this.toitReplie = true;
        this.capoteAttachee = true;
    }
}
```

```
public class C3Pluriel extends VoitureDecapotable {
    private boolean arceauxRetirés;

    public void replieLeToit() {
        this.toitReplie = true;
        this.arceauxRetirés = true;
    }
}
```

Attention : ce n'est pas de la redéfinition. On parle d'implémentation de méthode abstraite

www.serli.com
serli informatique

Classes abstraites : exemple VoitureDecapotable

```
public class Test {
    public static void main (String[] argv) {
        // Déclaration et création d'une DeuxCheveaux
        VoitureDecapotable voitureAncienne = new DeuxCheveaux(...);
        // Envoi de message
        voitureAncienne.replieLeToit();

        // Déclaration et création d'une C3Pluriel
         VoitureDecapotable voitureRecente = new C3Pluriel(...);
        // Envoi de message
        voitureRecente.replieLeToit();

        // Déclaration et création d'une VoitureDecapotable
        VoitureDecapotable voitureDecapotable =
            new VoitureDecapotable(...); // Erreur
    }
}
```

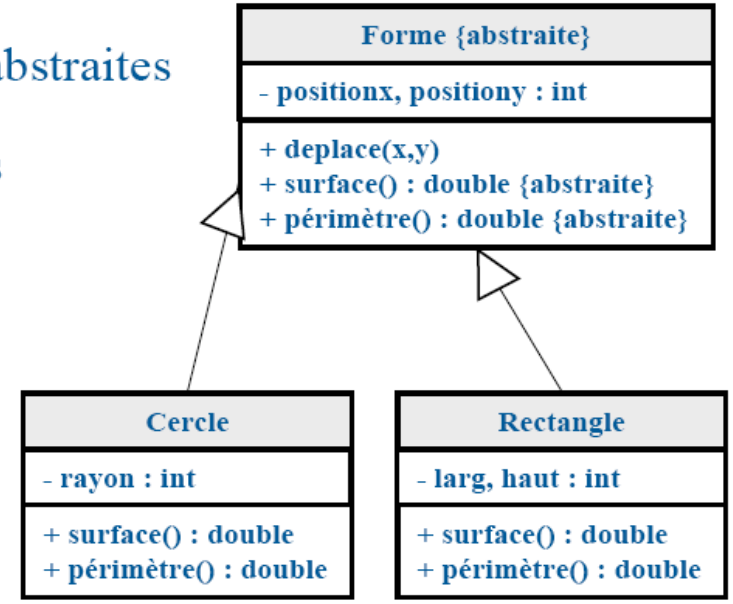
```
java.lang.Error: Problème de compilation non résolu :
    Le type VoitureDecapotable ne peut pas être instancié

    at VoitureDecapotable.main(VoitureDecapotable.java:20)
Exception in thread "main"
```

Attention : La classe VoitureDecapotable ne peut être instanciée puisqu'elle est abstraite

www.serli.com

Classes abstraites : exemple Forme

- La classe *Forme*
  - Les méthodes *surface()* et *périmètre()* sont abstraites
  - Ces méthodes n'ont de « sens » que pour les sous-classes *Cercle* et *Rectangle*

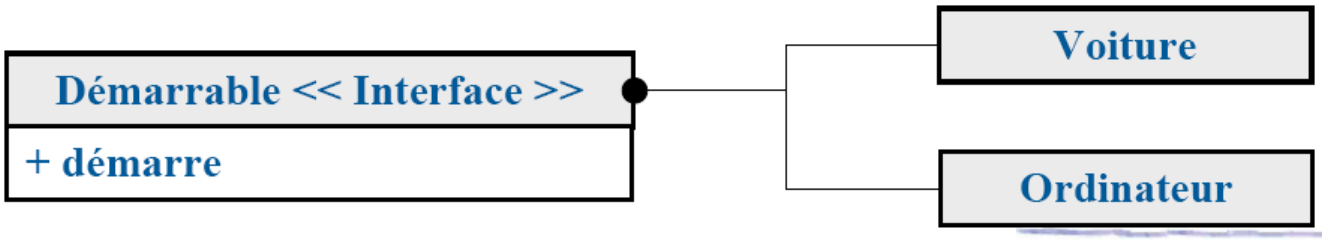



```
public abstract class Forme {
    private int positionx, positiony;

    public void deplacer(double dx, double dy){
        x += dx; y += dy;
    }

    public abstract double périmètre();
    public abstract double surface();
}
```

Pas d'implémentation !!

Notion d'interface

- Une interface est un modèle pour une classe
  - Quand toutes les méthodes d'une classe sont abstraites et il n'y a aucun attribut on aboutit à la notion d'interface
  - Elle définit la signature des méthodes qui doivent être implémentées dans les classes qui respectent ce modèle
  - Toute classe qui implémente l'interface doit implémenter toutes les méthodes définies par l'interface
  - Tout objet instance d'une classe qui implémente l'interface peut être déclaré comme étant du type de cette interface
  - Les interfaces pourront se dériver
- Exemple :
  - Les choses « Démarrable » doivent posséder une méthode « démarre() »



www.serli.com
serli informatique

Notion d'interface et Java

- Mise en œuvre d'une interface
  - La définition d'une interface se présente comme celle d'une classe. On y utilise simplement le mot clé **interface** à la place de **class**

```
public interface NomInterface {
    ...
}
```

Interface : ne pas confondre avec les interfaces graphiques

  - Lorsqu'on définit une classe, on peut préciser qu'elle implémente une ou plusieurs interfaces donnée(s) en utilisant une fois le mot clé **implements**

```
public class NomClasses implements Interface1,Interface3, ... {
    ...
}
```

  - Si une classe hérite d'une autre classe elle peut également implémenter une ou plusieurs interfaces

```
public class NomClasses extends SuperClasse implements Interface1, ... {
    ...
}
```

www.serli.com
serli informatique

Notion d'interface et Java

- Mise en œuvre d'une interface
  - Une interface ne possède pas d'attribut
  - Une interface peut posséder des constantes

```
public interface NomInterface {
   public static final int CONST = 2;
}
```

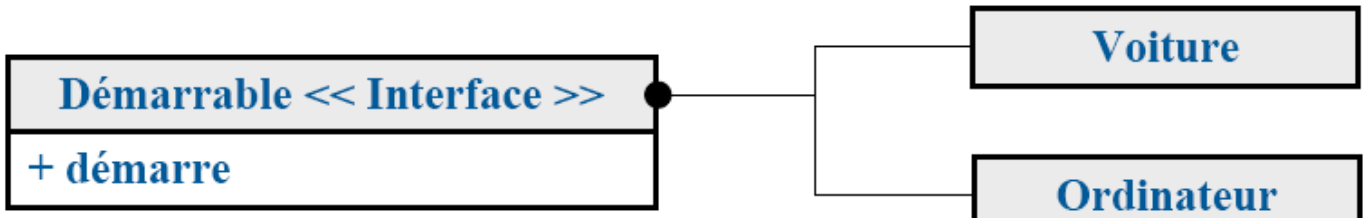
  - Une interface ne possède pas de mot clé **abstract**
  - Les interfaces ne sont pas instanciables (Même raisonnement avec les classes abstraites)

```
NomInterface jeTente = new NomInterface(); // Erreur!!
```

www.serli.com
serli informatique

Notion d'interface et Java

- Toute classe qui implémente l'interface doit implémenter toutes les méthodes définies par l'interface




```
public interface Démarrable {
    public void démarre();
}
```

```
public class Voiture implements Démarrable {
    ...

    public void démarre() {
        estDemarree = true
    }
}
```

Doit implémenter toutes les méthodes des interfaces

```
public class Ordinateur implements Démarrable {
    ...

    public void démarre() {
        estDemarree = true
    }
}
```

Une « Voiture » est « Démarrable »

Un « Ordinateur » est « Démarrable »

www.serli.com
serli

Notion d'interface et Java

- Tout objet instance d'une classe qui implémente l'interface peut être déclaré comme étant du type de cette interface

```
public class Test {
    public static void main (String[] argv) {
        // Déclaration d'un objet de type Démarrable
        Démarrable dem1;
        // Création d'un objet Voiture
        dem1 = new Voiture();

        // Déclaration et création d'un objet Personne
        Personne pers1 = new Personne(dem1);
        pers1.mettreEnRoute();

        // Déclaration d'un objet de type Démarrable
        Démarrable dem2;
        // Création d'un objet Ordinateur
        dem2 = new Ordinateur();

        // Déclaration et création d'un objet Personne
        Personne pers1 = new Personne(dem2);
        pers2.mettreEnRoute();
    }
}
```

Une personne peut démarrer tous les objets « Démarrable »

www.serli.com
serli

Notion d'interface et Java

- Une « Voiture » et un « Ordinateur » sont des objets « Démarrable »

```
public class Personne {

   private Demarrable objetDemarrable;

   public Personne(Demarrable dem) {
      objetDemarrable = dem;
   }

   public void mettreEnRoute() {
      objetDemarrable.demarre();
   }
}
```

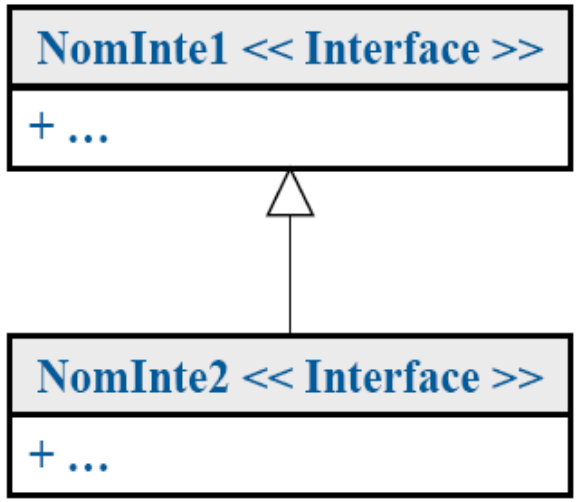
Une personne peut démarrer Voiture et Ordinateur sans connaître leur nature exacte

www.serli.com
serli informatique

Notion d'interface et Java

- Les interfaces pourront se dériver
  - Une interface peut hériter d'une autre interface : « extends »

- Conséquences
  - La définition de méthodes de l'interface mère (NomInte1) sont reprises dans l'interface fille (NomInte2). Toute classe qui implémente l'interface fille doit donner une implémentation à toutes les méthodes mêmes celle héritées



- Utilisation
  - Lorsqu'un modèle peut se définir en plusieurs sous-modèles complémentaires

www.serli.com
serli informatique

Classes abstraites versus interfaces

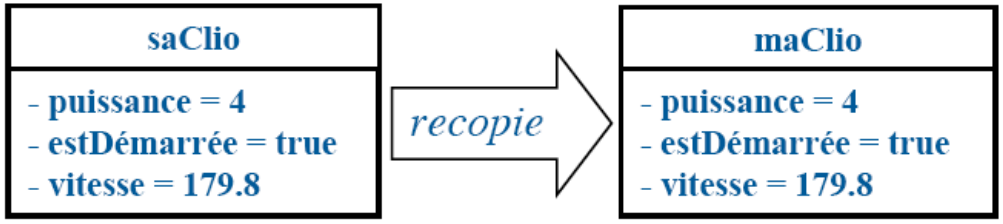
- Les classes
  - Elles sont complètement implémentées
  - Une autre classe peut en hériter
- Les classes abstraites
  - Elles sont partiellement implémentées
  - Une autre classe peut en hériter mais doit donner une implémentation aux méthodes abstraites
  - Elles ne peuvent pas être instanciées mais peuvent avoir un constructeur
- Les interfaces
  - Elles ne sont pas implémentées
  - Toute classe qui implémente une ou plusieurs interfaces doit implémenter toutes leurs méthodes (abstraites)

www.serli.com
serli informatique

L'interface « Cloneable »

- Deux possibilités pour dupliquer un objet de ses classes
  - Créer une méthode ad-hoc « public MaClasse dupliquer() » qui retourne une copie de l'objet en ayant créé une nouvelle instance et initialisé les attributs (en utilisant le constructeur)

saClio != maClio mais le contenu est identique



  - Utiliser l'interface « Cloneable » pour garder la compatibilité avec les autres classes de Java
    - Implémenter la méthode « protected Object clone() » de l'interface Cloneable

```
public class Voiture implements Demarrable, Cloneable {
    protected Object clone() {
        Voiture copie;
        copie = new Voiture(this.puissance, (Galerie)laGalerie.clone());
        return copie;
    }
}
```

www.serli.com
serli informatique

Les classes internes « Inner Classes »

- Règle de base en Java
  - Une classe par fichier et un fichier par classe

```
public class Voiture {
    ...
    class Moteur {
        ...
    }
}
```

- Classes locales ou internes
  - Définies à l'intérieur d'autres classes (Moteur dans Voiture)

- Classes anonymes
  - Sont des instanciations de classes et des implémentations d'une classe abstraite ou d'une interface
  - La ou les méthodes abstraites doivent être implémentées au moment de l'instanciation

```
Demarrable uneInstance =
   new Demarrable(){
        public void demarre() {
            // Code ici
        }
   };
```

Les classes anonymes sont très utilisées pour le développement d'IHM avec Java/Swing

www.serli.com
serli informatique

Les classes internes « Inner Classes »

- Code source : 1 fichier
  - classe
  - classe anonyme
  - classe interne
- Génération de byte-code : 3 fichiers
  - classe « Voiture.class »
  - anonyme « Voiture$1.class »
  - interne « Voiture$Moteur.class

Classe anonyme, implémente l'interface Initialable

```
public class Voiture {

    public Voiture(...) {
        monMoteur = new Moteur(...);
        Initialable monInit = new Initialable() {
            public void initialisation() {
                ...
            }
        };
    }
    class Moteur {
        ...
        public Moteur(...) {
            ...
        }
    }
}
```

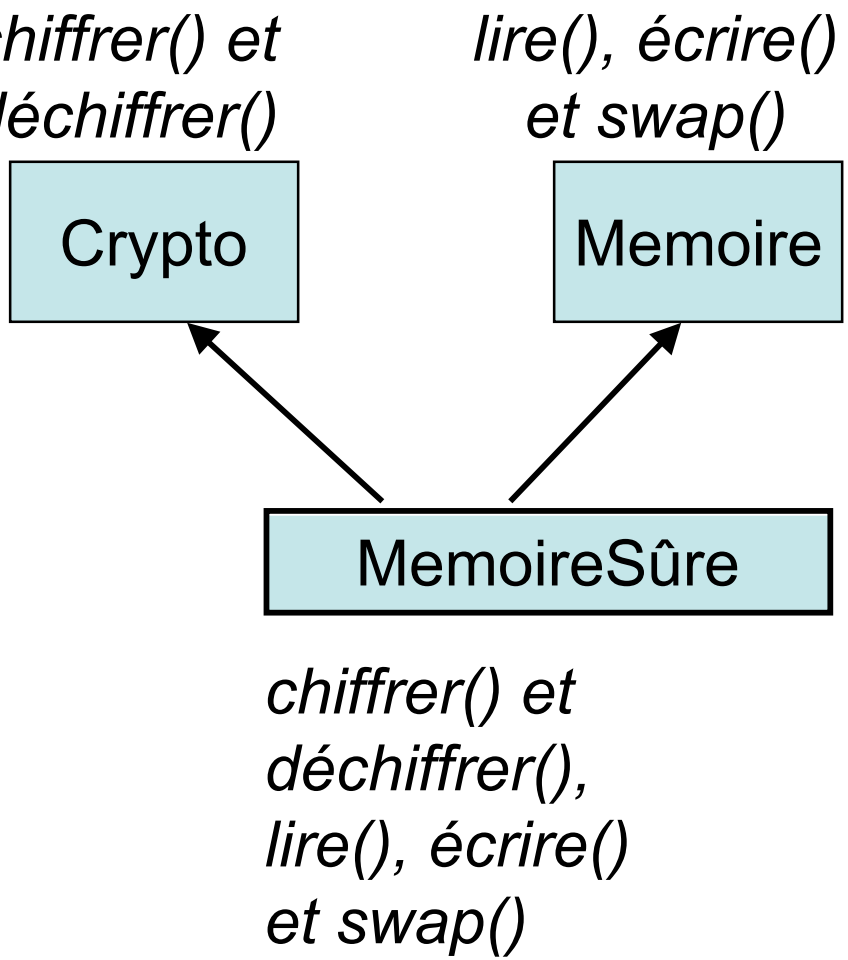
Classe interne

Nom	Taille	Type
Voiture$1.class	1 Ko	Fichier CLASS
Voiture$Moteur.class	1 Ko	Fichier CLASS
Voiture.class	1 Ko	Fichier CLASS
Voiture.java	1 Ko	Java Source File

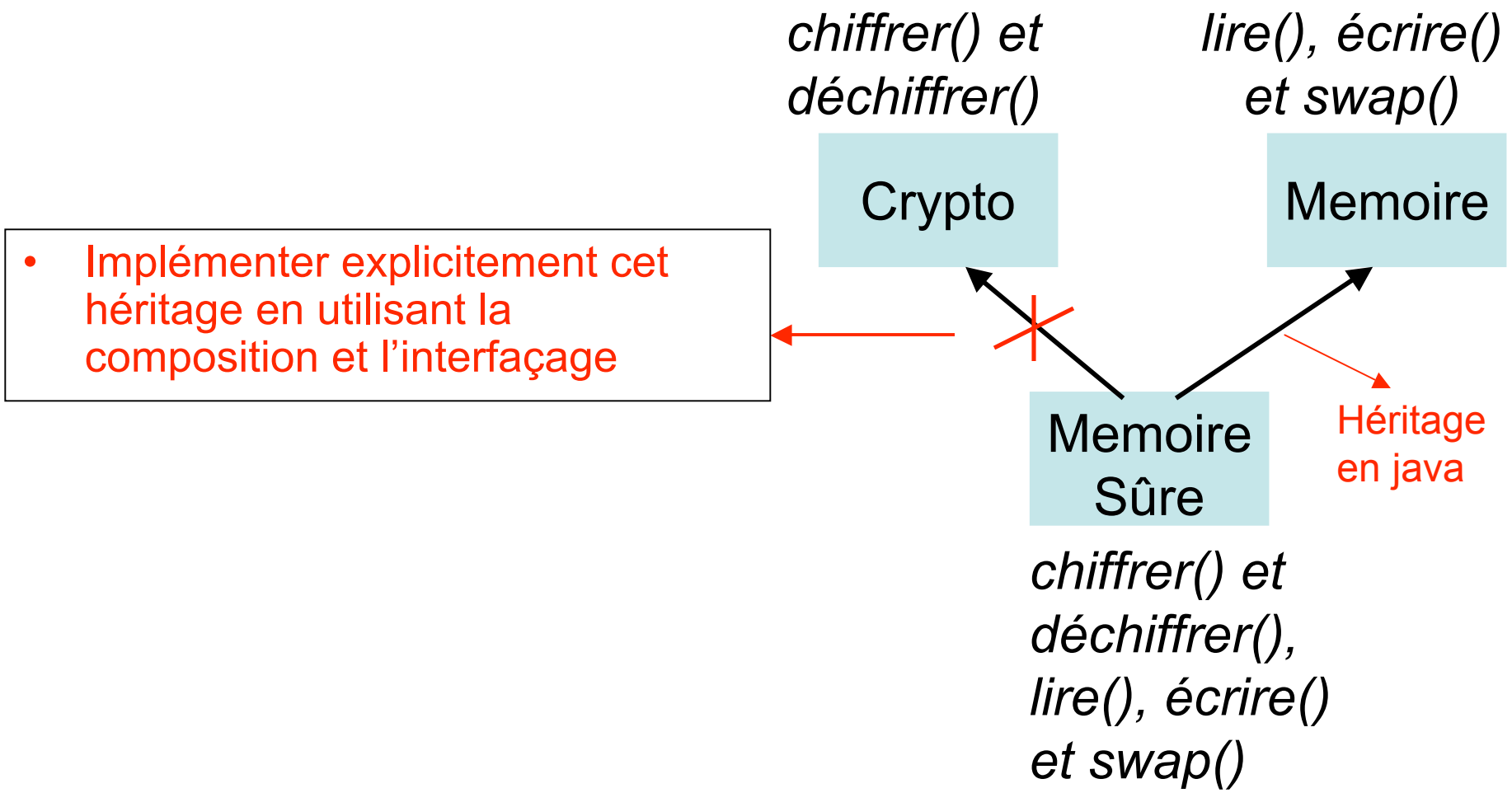
Les fichiers .class qui possède dans leur nom un $ ne sont pas des fichiers temporaires!!!

Héritage Multiple: Implémentation en Java

- La classe Mémoire-Sûre représente une page de mémoire qu'on peut chiffrer et déchiffrer



Héritage Multiple: Implémentation en Java



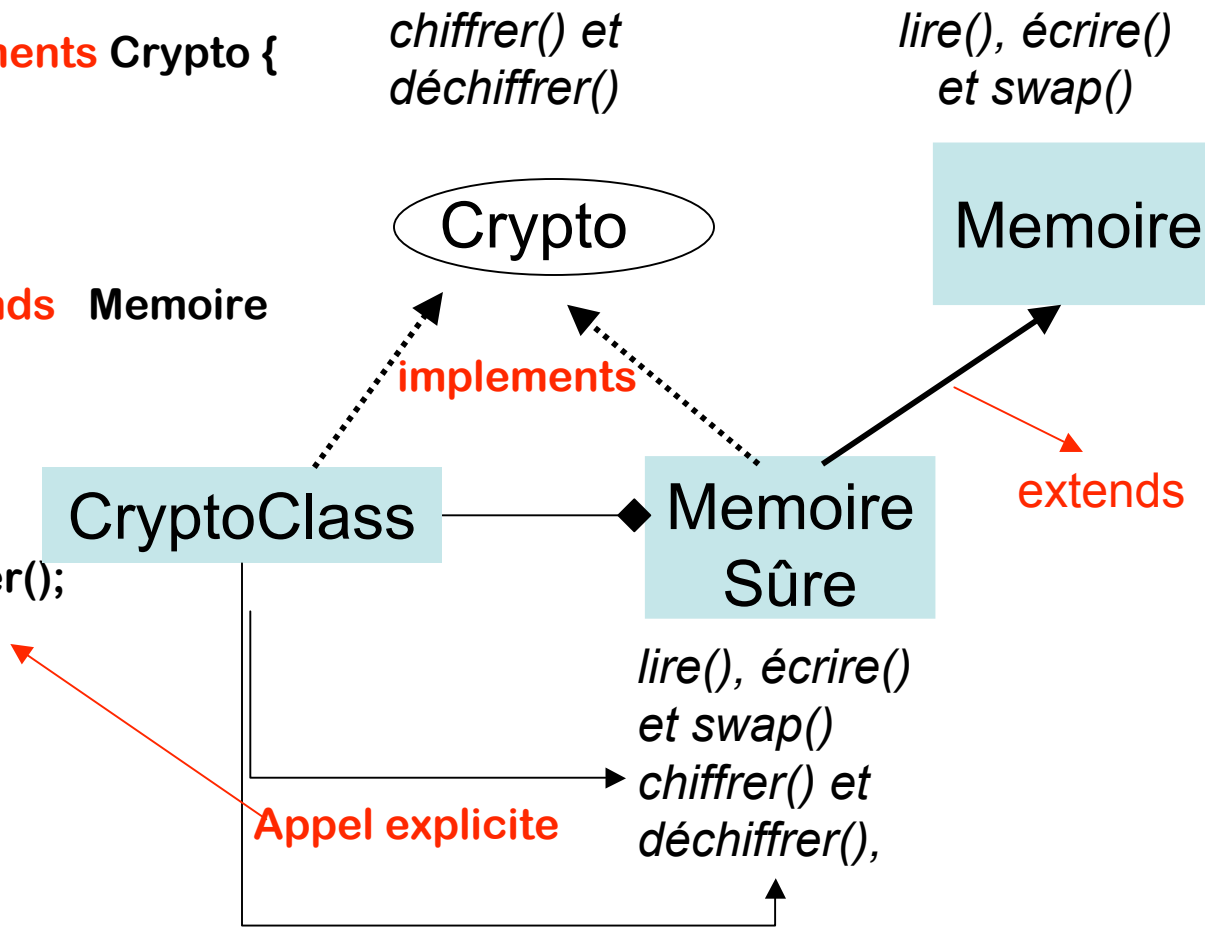
- Implémenter explicitement cet héritage en utilisant la composition et l'interfaçage

Héritage Multiple: Implémentation en Java

- **Class CryptoClass implements Crypto {**
  chiffrer(){..};
  dechiffrer() {..};
  }

- **Class MemoireSure extends Memoire**
  implements Crypto
  { CryptoClass C;

  chiffrer() { C. chiffrer();
  déchiffrer() { C.déchiffrer();
  }



Héritage Multiple: Implémentation en Java

1. Partage
   - Réutiliser du code de plusieurs sources dans une nouvelle classe
2. Typage
   - Garantir la transmission de typage entre sous classe est super classe
   - Une sous classe peut être utilisée partout ou sa superclasse est utilisée
   - P.ex: un objet MemoireSure peut être utilisé à la place d'un objet Crypto, ou bien à la place d'un objet Memoire

Exception

- Définition
  - Une exception est un signal qui indique que quelque chose d'exceptionnel (comme une erreur) s'est produit. Elle interrompt le flot d'exécution normal du programme
- A quoi ça sert
  - Gérer les erreurs est indispensable :
    - Mauvaise gestion peut avoir des conséquences catastrophiques (Ariane 5)
  - Mécanisme simple et lisible :
    - Regroupement du code réservé au traitement des erreurs (pas de « mélange » avec l'algorithme)
    - Possibilité de « récupérer » une erreur à plusieurs niveaux d'une application (propagation dans la pile des appels de méthodes)
- Vocabulaire
  - Lancer ou déclencher (**throw**) une exception consiste à signaler les erreurs
  - Capturer ou attraper (**catch**) une exception permet de traiter les erreurs

www.serli.com
serli informatique

Exception

Premier exemple : lancer et capturer une exception

```
public class Point {
    ... // Déclaration des attributs

    ... // Autre méthodes et constructeurs

    public Point(int x, int y) throws ErrConst {
        if ((x < 0) || (y < 0)) throw new ErrConst();
        this.x = x ; this.y = y;
    }

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

```
public class Test {
    public static void main(String[] argv) {
        try {
            Point a = new Point(1,4);
            a.affiche();
            a = new Point(-2, 4);
            a.affiche();
        } catch (ErrConst e) {
            System.out.println("Erreur Construction");
            System.exit(-1);
        }
    }
}
```

La classe *ErrConst* n'est pas encore défini. A voir plus tard

```
Coordonnées : 1 4
Erreur Construction
```

www.serli.com
serli informatique

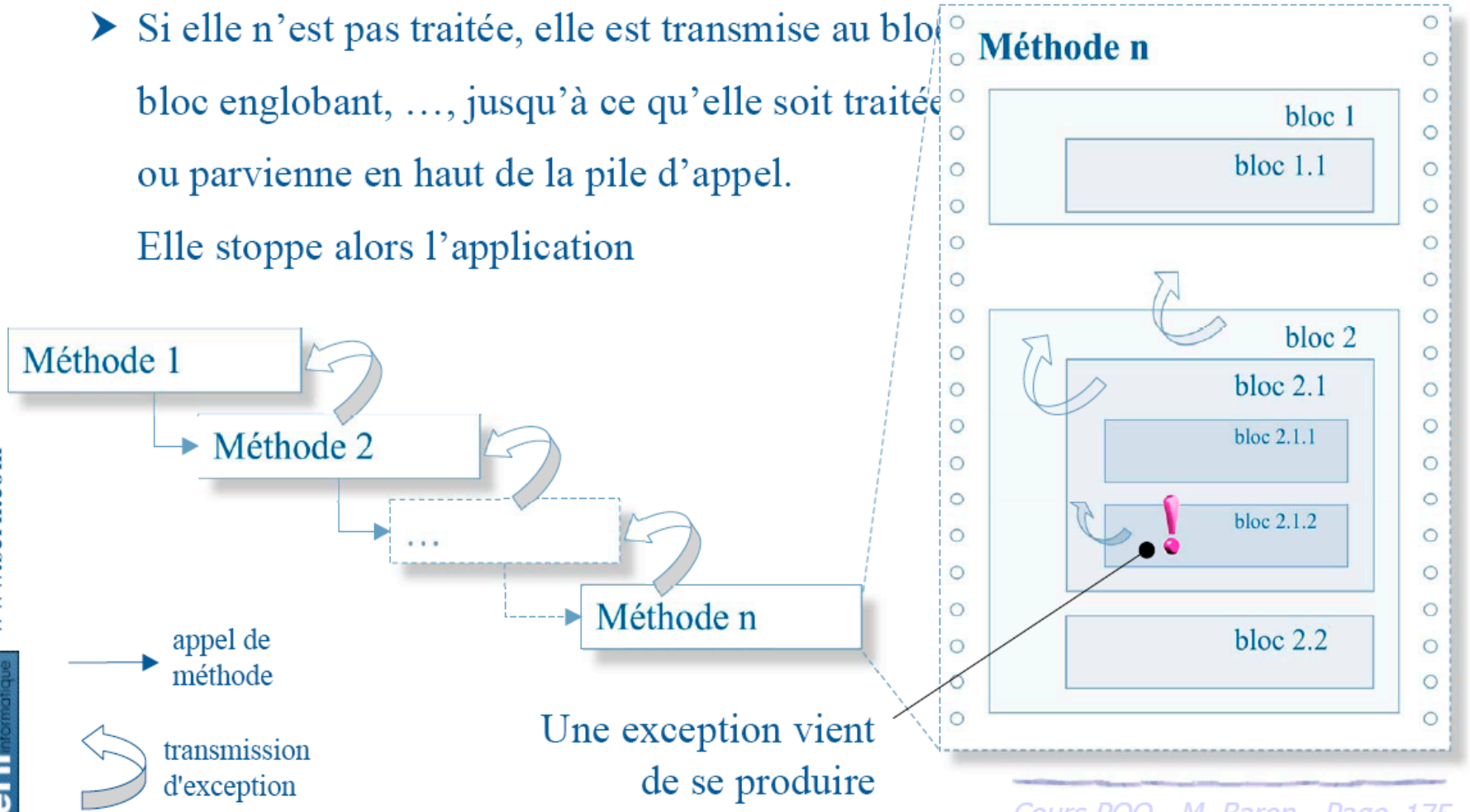
Exception : mécanisme

- Explication
  - Lorsqu'une situation exceptionnelle est rencontrée, une exception est lancée
  - Si elle n'est pas traitée, elle est transmise au bloc

    bloc englobant, …, jusqu'à ce qu'elle soit traitée

    ou parvienne en haut de la pile d'appel.

    Elle stoppe alors l'application



www.serli.com
serli informatique

Exception : lancer ou déclencher

- Une méthode déclare qu'elle peut lancer une exception par le mot clé **throws**

```
public Point(int x, int y) throws ErrConst {
    ...
}
```

Permet au constructeur *Point* de lancer une exception *ErrConst*

- Soit la méthode lance une exception, en créant une nouvelle valeur (un objet) d'exception en utilisant le mot clé **throw**

```
public Point(int x, int y) throws ErrConst {
    if ((x < 0) || (y < 0)) throw new ErrConst();
    this.x = x ; this.y = y;
}
```

Création d'une nouvelle valeur d'exception

- Soit la méthode appelle du code qui lance une exception

```
public Point(int x, int y) throws ErrConst {
    checkXYValue(x,y);
    this.x = x ; this.y = y;
}
```

```
private void checkXYValue(in x, int y)
throws ErrConst {
    if ((x < 0) || (y < 0))
        throw new ErrConst();
}
```

www.serli.com
serli

Exception : capturer ou attraper

- On parle ici de gestionnaire d'exception. Il s'agit de traiter par des actions la situation exceptionnelle
- On délimite un ensemble d'instructions susceptibles de déclencher une exception par des blocs **try {…}**

```
try {
    Point a = new Point(1,4);
    a.affiche();
    a = new Point(-2, 4);
    a.affiche();
}
```

Méthodes à risques. Elles sont « surveillées »

- La gestion des risques est obtenue par des blocs **catch(TypeException e) {…}**

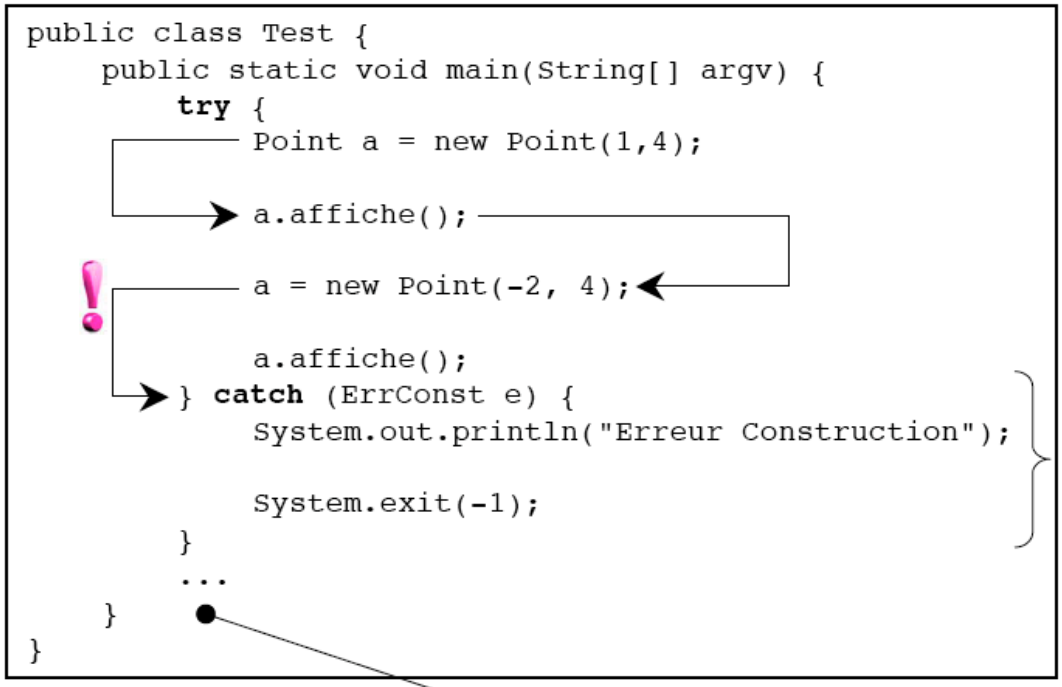
```
} catch (ErrConst e) {
    System.out.println("Erreur Construction");
    System.exit(-1);
}
```

- Ces blocs permettent de capturer les exceptions dont le type est spécifié et d'exécuter des actions adéquates

www.serli.com
serli informatique

Exception : capturer ou attraper

- Compréhension du mécanisme de capture



L'erreur exceptionnelle est traitée par le bloc **catch**

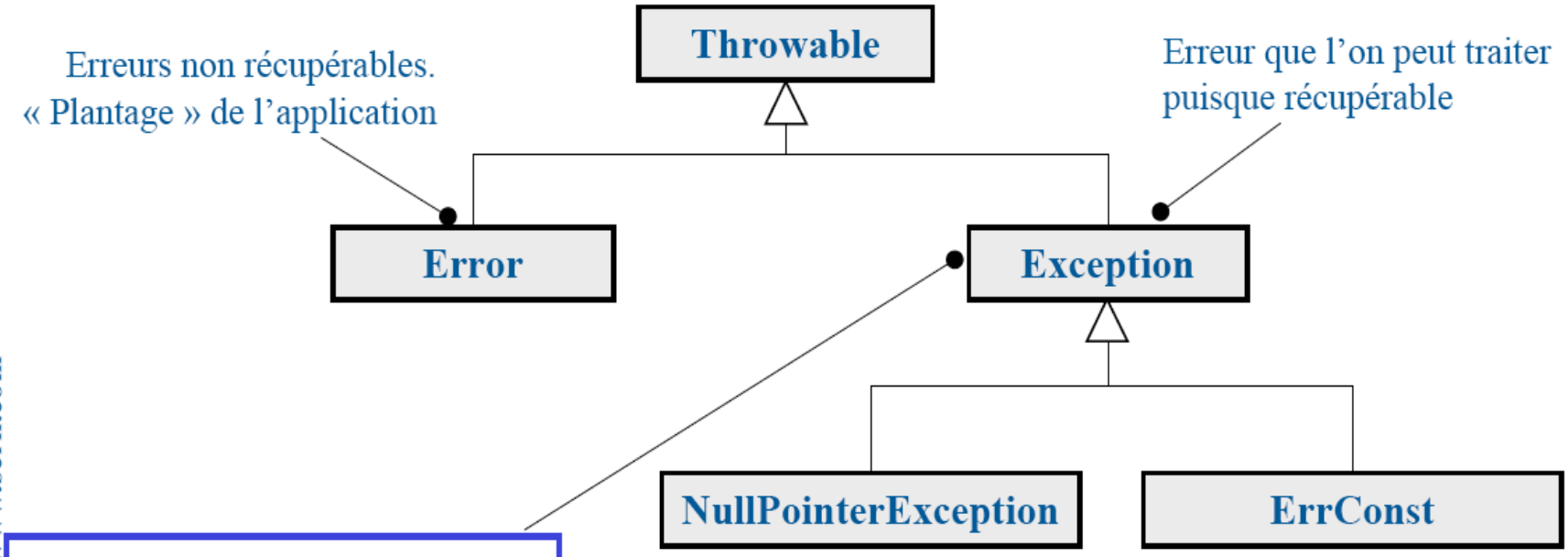
Puis, il y a poursuite de l'exécution en dehors du bloc **try catch**

Remarque : si erreur le programme s'arrête (System.exit(-1))

www.serli.com
serli informatique

Exception : modélisation

- Les exceptions en Java sont considérées comme des objets
- Toute exception doit être une instance d'une sous-classe de la classe *java.lang.Throwable*

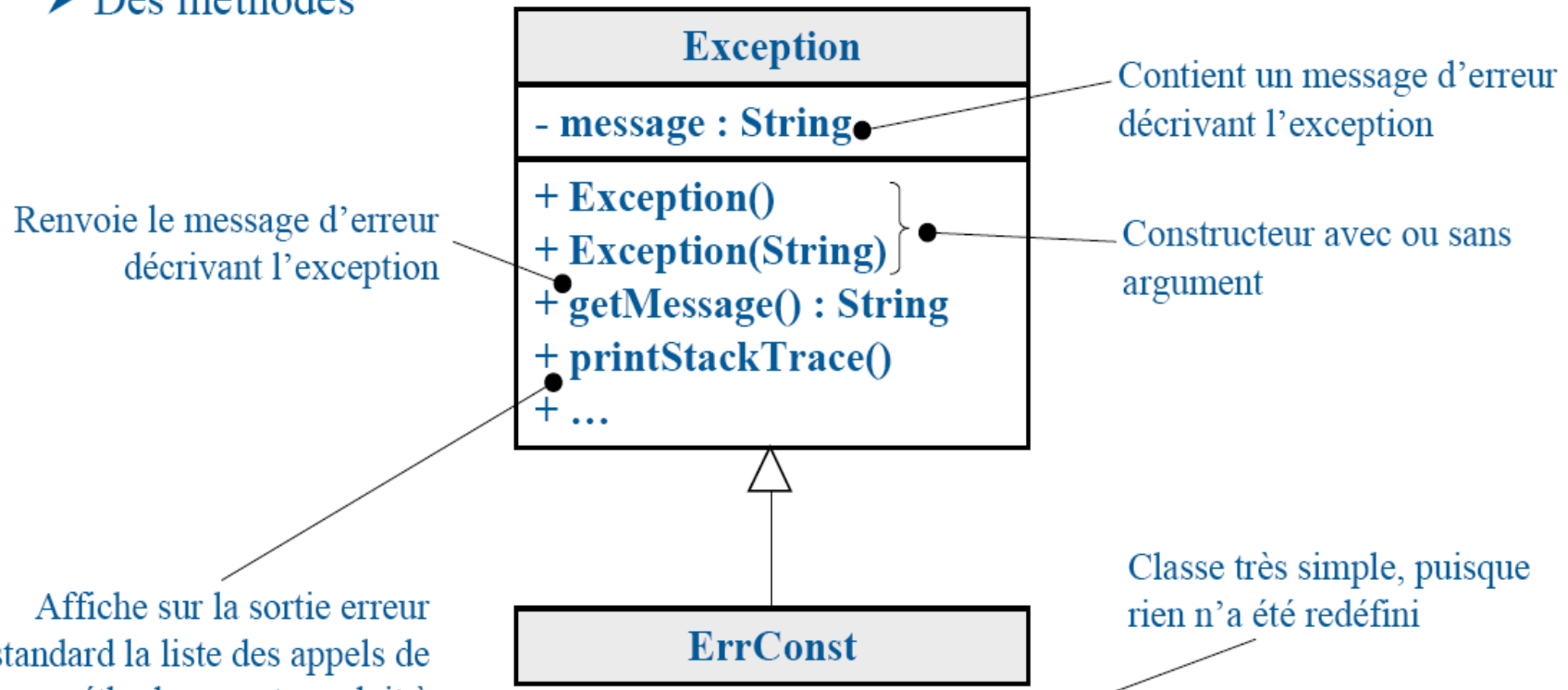


Pour définir de nouveaux types d'exception, on sous-classera la classe *Exception*

www.serli.com

Exception : modélisation

- Les exceptions sont des objets nous pouvons donc définir
  - Des attributs particuliers
  - Des méthodes



www.serli.com
serli informatique

Exception : modélisation

- Utilisation de l'objet *ErrConst*

Erreur de type *ErrConst* qui hérite de Exception

```
public class Test {
    public static void main(String[] argv) {
        try {
            ...
        } catch (ErrConst e) {
            System.out.println("Erreur Construction");

            System.out.println(e.getMessage());

            e.printStackTrace();

            System.exit(-1);
        }
        ...
    }
}
```

Affichage de l'erreur

Affichage de la liste des méthodes

```
Console [<arrêté> C:\Program F...n\javaw.exe (02/08/04 22:01)]
Coordonnées : 1 4
Erreur Construction
ErrConst
    at Point.<init>(Point.java:18)
    at Test.main(Test.java:20)
```

www.serli.com
serli informatique

Exception : attraper les tous ...

- Il est possible de capturer plus d'une exception. Un bloc **try** et plusieurs blocs **catch**

```
public class Point {
    public void deplace(int dx, int dy) throws ErrDepl {
        if (((x+dx) < 0) || ((y+dy) <0)) throw new ErrDepl();
            x += dx ; y +=dy;
    }

    public Point(int x, int y) throws ErrConst {
        if ((x < 0) || (y < 0)) throw new ErrConst();
        this.x = x ; this.y = y;
    }
    .
}
```

```
public class Test {
    public static void main(String[] argv) {
        try {
            ... // Bloc dans lequel on souhaite détecter
                    les exceptions ErrConst et ErrDepl
        } catch (ErrConst e) {
            System.out.println("Erreur Construction");
            System.exit(-1);
        } catch (ErrDepl e) {
            System.out.println("Erreur Déplacement");
            System.exit(-1);
        }
    }
}
```

Définition d'une nouvelle méthode qui lance une exception

Attrape la nouvelle exception de type *ErrDepl*

www.serli.com
serli informatique

Exception : attraper les tous ...

- Toute méthode susceptible de lever une exception doit :
  - Soit l'attraper (bloc **try catch**)
  - Soit déclare explicitement qu'elle peut lancer une exception (mot clé **throws)**
- Les exceptions déclarées dans la clause **throws** d'une méthode sont :

Les exceptions levées dans la méthode (*Point*) et non attrapées par celle-ci

```
public Point(int x, int y) throws ErrConst {
     if ((x < 0) || (y < 0)) throw new ErrConst();
     this.x = x ; this.y = y;
}
```

Les exceptions levées dans des méthodes (*checkXYValue*) appelées par la méthode (*Point*) et non attrapées par celle-ci

```
public Point(int x, int y) throws ErrConst {
     checkXYValue(x,y);
     this.x = x ; this.y = y;
}
```

```
private void checkXYValue(in x, int y) throws
ErrConst {
      if ((x < 0) || (y < 0))
          throw new ErrConst();
}
```

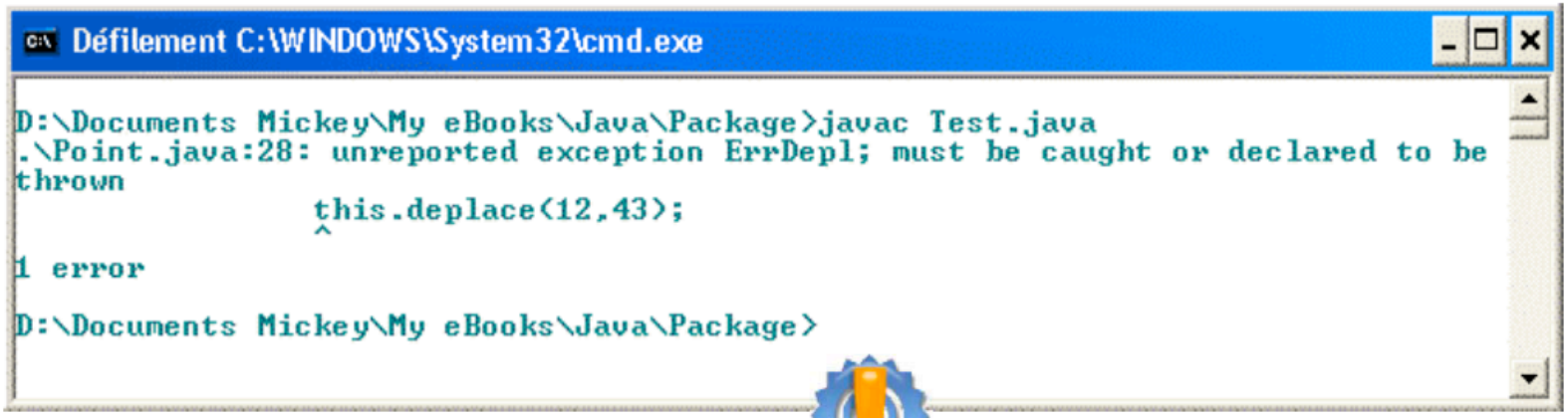
www.serli.com
serli informatique

Exception : attraper les tous ...

- Il faut s'assurer que les exceptions sont sous contrôles

```
public class Point {
    public void deplace(int dx, int dy) throws ErrDepl {
        if (((x+dx) < 0) || ((y+dy) <0)) throw new ErrDepl();
            x += dx ; y +=dy;
    }

    public void transformer() {
        ...
        this.deplace(...);
    }
}
```

```
public class ErrDepl extends Exception {
    public ErrDepl() {
        ...
    }
}
```



Ne pas oublier de traiter une exception sinon le compilateur ne vous loupe pas!!!!

www.serli.com
serli informatique

Exception : attraper les tous ...

- Pour garantir une bonne compilation deux solutions

```
public class Point {
    public void deplace(int dx, int dy) throws ErrDepl {
        if (((x+dx) < 0) || ((y+dy) <0)) throw new ErrDepl();
            x += dx ; y +=dy;
    }

    public void transformer() {
        ...
        this.deplace(...);
    }
}
```

Soit en ajoutant explicitement l'instruction **throws** à la méthode *transformer* de façon à rediriger l'erreur

```
public void transformer()
    throws ErrDepl {
    ...
    this.deplace(...);
}
```

Soit en entourant d'un bloc **try … catch** la méthode qui peut poser problème

```
public void transformer() {
    try {
        ...
        this.deplace(...);
    } catch (ErrDepl e) {
        e.printStackTrace();
    }
}
```

www.serli.com
serli informatique

Exception : transmission d'information

- Possibilité d'enrichir la classe *ErrConst* en ajoutant des attributs et des méthodes de façon à communiquer

```
public class Point {
    public Point(int x, int y) throws ErrConst {
        if ((x < 0) || (y < 0)) throw new ErrConst(x,y);
        this.x = x ; this.y = y;
    }
    ...
}
```

```
public class ErrConst extends Exception {
    private int abs, ord;

    public ErrConst(int x, int y) {
        this.abs = x;
        this.ord = y;
    }

    public int getAbs() { return this.abs; }
    public int getOrd() { return this.ord; }
}
```

ErrConst
- abs, ord : int
+ ErrConst(x,y) + getAbs : int + getOrd : int

```
public class Test {
    public static void main(String[] argv) {
        try {
            ...
            a = new Point(-2, 4);
        } catch (ErrConst e) {
            System.out.println("Erreur Construction point");
            System.out.println("Coordonnées souhaitées : "
              + e.getAbs() + " " + e.getOrd());
            System.exit(-1);
        }
    ...
```

ErrConst permet de connaître les valeurs qui ont fait échouer la construction de *Point*

www.serli.com
serli informatique

Exception : finally

- Bloc finally : c'est une instruction optionnelle qui peut servir de « nettoyage ». Elle est exécutée quelle que soit le résultat du bloc try (c'est-à-dire qu'il ait déclenché une exception ou non)
- Permet de spécifier du code dont l'exécution est garantie quoi qu'il arrive
- Intérêt double :
  - Rassembler dans un seul bloc un ensemble d'instructions qui autrement auraient du être dupliquées
  - Effectuer des traitements après le bloc try, même si une exception a été levée et non attrapée par les blocs catch

www.serli.com
serli informatique

Exception : finally

```
public class Test {
    public static void main(String[] argv) {
        try {
            ... // Bloc dans lequel on souhaite détecter
                    les exceptions ErrConst et ErrDepl
        } catch (ErrConst e) {
            System.out.println("Erreur Construction");
            System.out.println("Fin du programme");
            System.exit(-1);
        } catch (ErrDepl e) {
            System.out.println("Erreur Déplacement");
            System.out.println("Fin du programme");
            System.exit(-1);
        }
    }
}
```

Ces instructions sont rappelées plusieurs fois

Au moyen du mot clé **finally**, il est possible de factoriser

```
public class Test {
    public static void main(String[] argv) {
        try {
            ... // Bloc dans lequel on souhaite détecter
                    les exceptions ErrConst et ErrDepl
        } catch (ErrConst e) {
            System.out.println("Erreur Construction");
        } catch (ErrDepl e) {
            System.out.println("Erreur Déplacement");
        } finally {
            System.out.println("Fin du programme");
            System.exit(-1);
        }
    }
}
```

www.serli.com
serli informatique

Exception : pour ou contre

```
erreurType lireFichier() {
    int codeErreur = 0;
    // Ouvrir le fichier
    if (isFileIsOpen()) {
        // Détermine la longueur du fichier
        if (getFileSize()) {
            // Vérification de l'allocation de la mémoire
            if (getEnoughMemory()) {
                // Lire le fichier en mémoire
                if (readFailed()) {
                    codeErreur = -1;
                }
            } else {
                codeErreur = -2;
            }
        } else {
            codeErreur = -3;
        }

        // Fermeture du fichier
        if (closeTheFileFailed()) {
            codeErreur = - 4;
        }
    } else {
        codeErreur = - 5;
    }
}
```

La gestion des erreurs devient très difficile

Difficile de gérer les retours de fonctions

Le code devient de plus en plus conséquent

www.serli.com
serli

Exception : pour ou contre

- Le mécanisme d'exception permet
  - La concision
  - La lisibilité

```
void lireFichier() {
    try {
        // Ouvrir le fichier
        // Détermine la longueur du fichier
        // Vérification de l'allocation de la mémoire
        // Lire le fichier en mémoire
        // Fermer le fichier
    } catch (FileOpenFailed) {
        ...
    } catch (FileSizeFailed) {
        ...
    } catch (MemoryAllocFailed) {
        ...
    } catch (FileReadFailed) {
        ...
    } catch (FileCloseFailed) {
        ...
    }
}
```

Préférer cette solution à la précédente. Programmation propre et professionnelle

www.serli.com
serli informatique

Exception : les exceptions courantes

- Java fournit de nombreuses classes prédéfinies dérivées de la class *Exception*
- Ces exceptions standards se classent en deux catégories
  - Les exceptions explicites (celles que nous avons étudiées), mentionnées par mot clé **throws**
  - Les exceptions implicites qui ne sont pas mentionnées par le mot clé **throw**
- Liste de quelques exceptions
  - *ArithmeticException* (division par zéro)
  - *NullPointerException* (référence non construite)
  - *ClassCastException* (problème de cast)
  - *IndexOutOfBoundsException* (problème de dépassement d'index dans table

www.serli.com

Les collections : les génériques Java

- Avec la version 5 de Java possibilité d'exploiter les génériques dans les collections et pour d'autres aspects du langage également
- Une syntaxe particulière a été rajoutée de manière à prendre en considération les génériques
  - < ? > : signale qu'il faut préciser le type de la classe
  - < ? , ? > : signale qu'il faut préciser deux types
- Avec les génériques il va être possible de fixer à la construction de la collection le type du contenu stocké dans les collections
- Avanages
  - Toutes les méthodes accesseurs et modifieurs qui manipulent les éléments d'une collection sont *signés* suivant le type défini à la construction de la collection
  - Vérification des types pendant le développement (avant problème de CastClassException)

www.serli.com
serli technologique

Les collections : Iterator

- Les itérateurs permettent de parcourir les éléments d'une collection sans connaissance précise du type de la collection : Polymorphisme

- Il existe deux familles d'itérateurs :
  - *monodirectionnels*

    Le parcours de la collection se fait d'un début vers une fin; on ne passe qu'une seule fois sur chacun des éléments
  - *bidirectionnels*

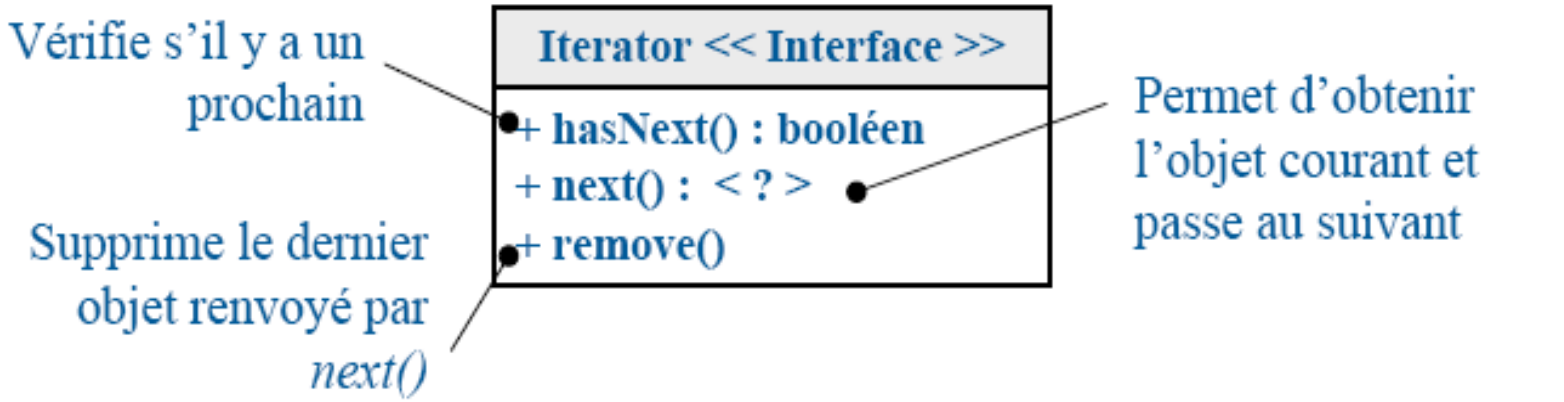
    Le parcours de la collection peut se faire dans les deux sens ; on peut avancer et reculer à sa guise dans la collection

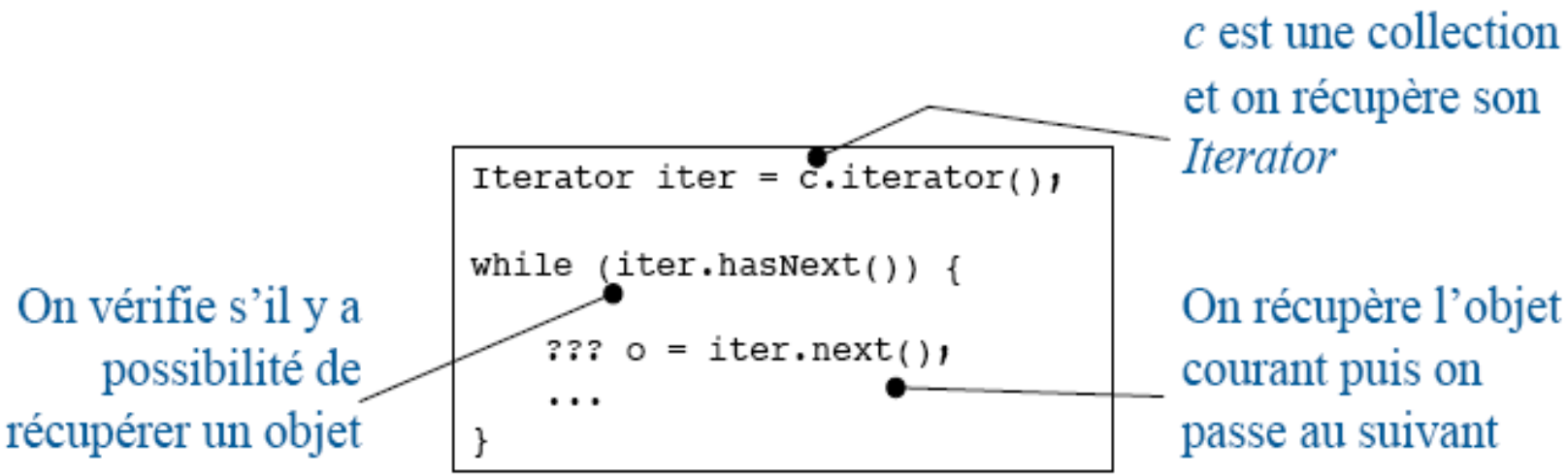
La notion d'Iterator fait partie de l'ensemble des Design Patterns

www.serli.com
serli

Les collections : Iterator

- Itérateur monodirectionnels : interface *Iterator*
  - Par défaut, toutes collections possèdent une attribut de type *Iterator*

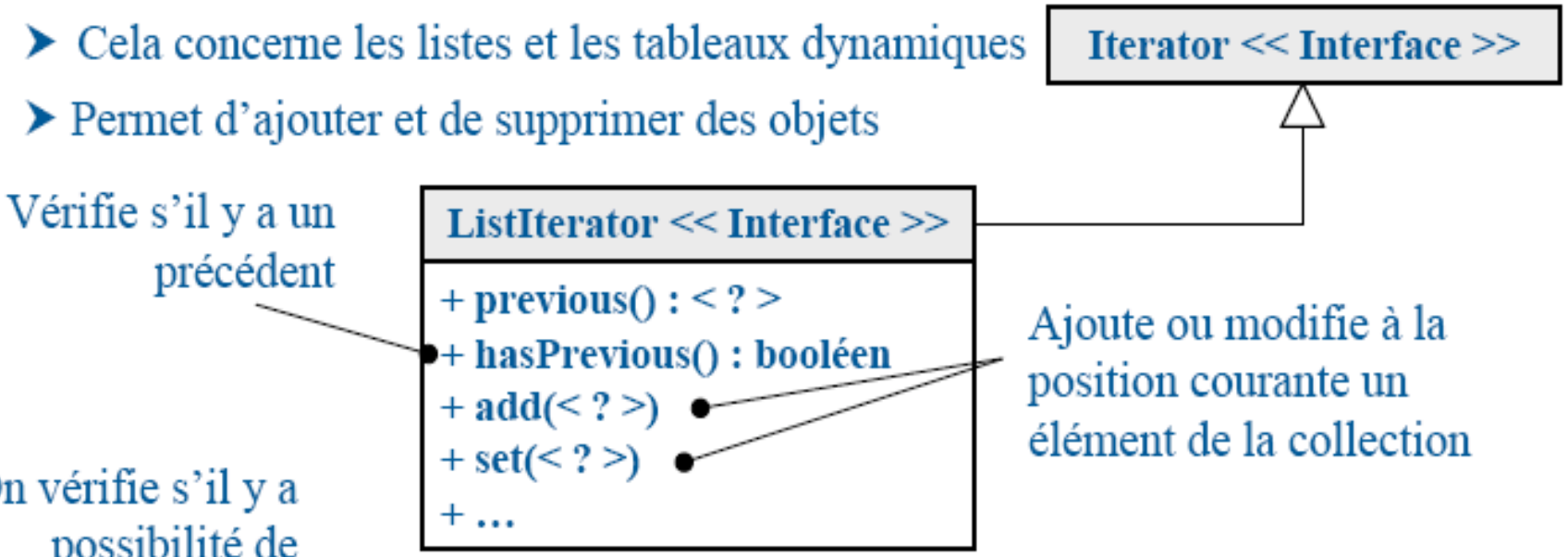




www.serli.com
serli informatique

Les collections : Iterator

- Itérateur bidirectionnels : interface *ListIterator*
  - Cela concerne les listes et les tableaux dynamiques
  - Permet d'ajouter et de supprimer des objets



On vérifie s'il y a possibilité de récupérer un objet précédemment

c est une collection et on récupère son *ListIterator* Initialise en début de liste

```
Iterator iter = c.listIterator();

while (iter.hasPrevious()) {

   ??? o = iter.previous();
    ...
}
```

Récupère l'objet précédemment puis on passe au précédent

www.serli.com
serli

Les collections : LinkedList

- Cette classe permet de manipuler des listes dites « doublement chaînées ».
  A chaque élément de collection, on associe implicitement deux informations qui sont les références à l'élément précédent et suivant

```
LinkedList<string> l1 = new LinkedList<string>();
ListIterator iter = l1.listIterator();

iter.add("Bonjour");
iter.add("Coucou");

while(iter.hasPrevious()) {
    String o = iter.previous();
    System.out.println(o);
}
```

Plus rien après ses éléments, on fait un retour en arrière

Ajout des éléments au travers de l'itérateur

L'utilisation de la *LinkedList* est transparente

www.serli.com
serli informatique

Les collections : LinkedList

- Possibilité d'utiliser les collections (ici LinkedList est un exemple) sans les itérateurs mais moins performant !!!

```
LinkedList<String> l1 = new LinkedList<String>();

l1.add("Bonjour");
l1.add("Coucou");

for (int i = 0; i < l1.size(); i++) {
    String o = l1.get(i);
    System.out.println(o);
}
```

L'utilisation de la *LinkedList* n'est pas transparente. Connaissance obligatoire de ces méthodes

Utilisation de la méthode *add* de la classe *LinkedList*

Ne pas modifier la collection (*add* de *LinkedList*) pendant qu'on utilise l'itérateur (*next()*)

www.serli.com
serli

Les collections : ArrayList

- La classe *ArrayList* est une encapsulation du tableau avec la possibilité de le rendre dynamique en taille
- Possibilité d'utiliser des *ListIterator* mais on préfère son utilisation à un élément de rang donné

```
ArrayList<Object> myArrayList = new ArrayList<Object>();

myArrayList.add("Coucou");
myArrayList.add(34);

for (int i = 0; i < myArrayList.size(); i++) {
    Object myObject = myArrayList.get(i);
    if (myObject instanceof String) {
        System.out.println("Chaîne:" + ((String)myObject));
    }

    if (my_object instanceof Integer) {
        System.out.println("Integer:" + ((Integer)myObject));
    }
```

Préférer l'utilisation de la classe *ArrayList* au lieu de la classe *Vector*

www.serli.com

Les collections : HashSet

- La classe *HashSet* permet de gérer les ensembles. Deux éléments ne peuvent être identiques
- Il faut prévoir deux choses dans vos classes :
  - La redéfinition de la méthode *hashCode()* qui est utilisée pour ordonnancer les éléments d'un ensemble (calcul la table de hachage d'un objet)
  - La redéfinition de la méthode *equals(Object)* qui compare des objets de même classe pour connaître l'appartenance d'un élément à l'ensemble

```
public class TestHashSet {
     public static void main(String[] argv) {
          Point p1 = new Point(1,3), p2 = new Point(2,2);
          Point p3 = new Point(4,5), p4 = new Point(1,8);
          Point p[] = {p1, p2, p1, p3, p4, p3}

          HashSet<Point> ens = new HashSet<Point>();
          for (int i = 0; i<p.length; i++) {
               System.out.println("Le Point ") ; p[i].affiche();
               boolean ajoute = ens.add(p[i]);
               if (ajoute) System.out.println(" a été ajouté");
                    else System.out.println("est déjà présent");
               System.out.print("Ensemble = "); affiche(ens);
          }
     }
}
```

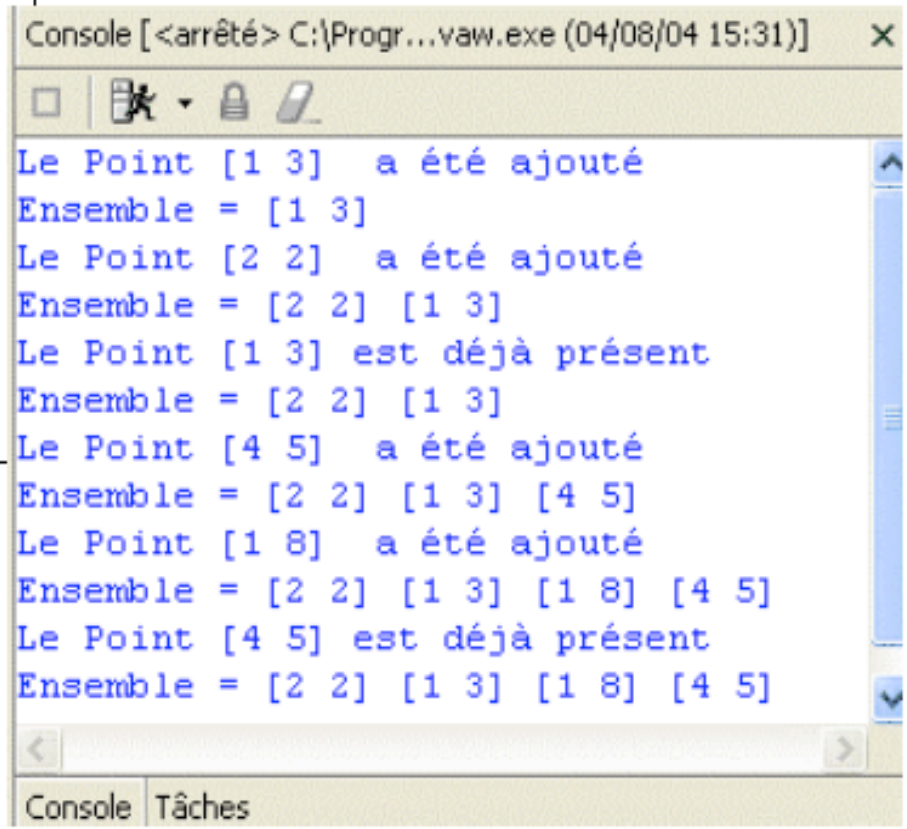
```
public static void affiche(HashSet ens) {
     Iterator iter = ens.iterator();
     while(iter.hasNext()) {
          Point p = iter.next();
          p.affiche();
     }
     System.out.println();
}
```

www.serli.com
serli

Les collections : HashSet

```
public class Point {
    private int x,y;

    Point(int x, int y) {
        this.x = x; this.y = y;
    }
    public int hashCode() {
        return x+y;
    }
    public boolean equals(Object pp) {
        Point p = (Point)pp;
        return ((this.x == p.x) &
            (this.y == p.y));
    }
    public void affiche() {
        System.out.print("[" + x + " "
            + y + "] ");
    }
}
```

Redéfinition des méthodes *hashCode()* et *equals(Object)*



www.serli.com
serli informatique

Classes enveloppes des types primitifs

- Certaines manipulations nécessitent de travailler sur des objets (instances de classes) et pas sur des types primitifs (int, char, double, …)
- Le paquetage **`java.lang`** fournit des classes pour "envelopper" les types primitifs (les objets sont constants) :
  - **`Byte, Short, Integer, Long, Float, Double, Boolean, Character`**

Méthodes utilitaires des classes enveloppes

- Ces classes "enveloppes" :
  - permettent la manipulation des valeurs des types primitifs comme des objets
  - offre des méthodes utilitaires (le plus souvent `static`) pour faire des conversions entre types primitifs (et `String`)

Méthodes de conversions de Integer

- **`int` → `Integer`** (constructeur de **`Integer`**) :
  ```
  Integer(int i)
  ```

- **`Integer` → `int`** :
  ```
  int intValue()
  ```

- **`String` → `Integer`** :
  ```
  static Integer valueOf(String ch [,int base])
  ```

- **`Integer` → `String`** :
  ```
  String toString()
  ```

- **`String` → `int`** :
  ```
  static int parseInt(String ch [,int base])
  ```

- **`int` → `String`** (méthode de la classe **`String`**) :
  ```
  static String valueOf(int i)
  ```

Conversion en entier d'un paramètre

```
/** Afficher le double du nombre passé en paramètre
   */
public class AfficheParam {
  public static void main(String[] args) {
    int i = Integer.parseInt(args[0]);
    System.out.println(i*2);
  }
}
```