Programmation orientée objet et temps réelle avec Java

Interfaces de programmation

Dominique Blouin
Maître de conférence
Télécom Paris, Institut Polytechnique de Paris
dominique.blouin@telecom-paris.fr

Objectifs d'apprentissage

- Notion d'interface
- Héritage d'interfaces
- Interfaces de constantes et marquage
- Les principales interfaces de collections en Java
- Programmer à l'interface

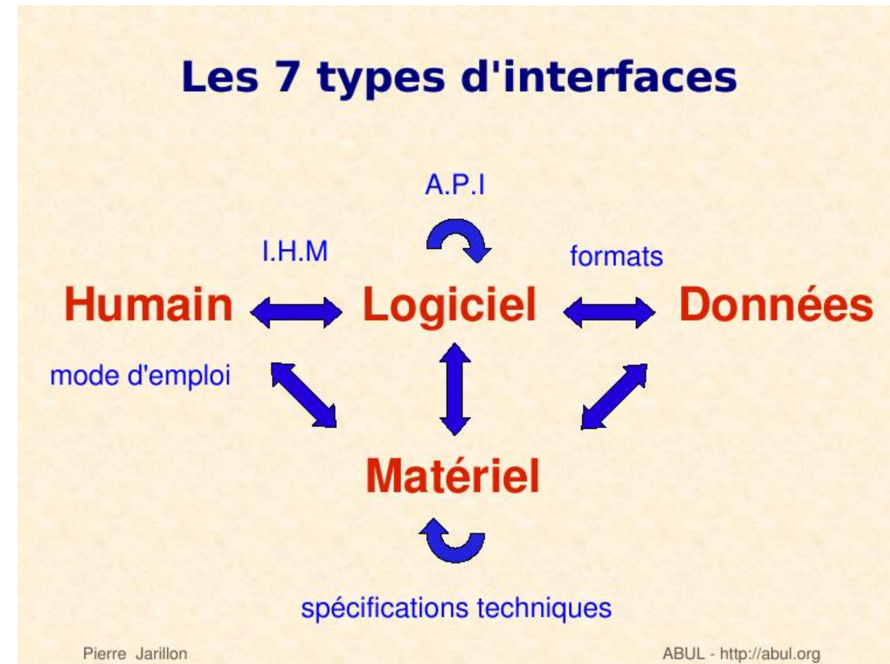
TELECOM Paris IP PARIS

Qu’est-ce qu’une interface ? Dictionnaire Robert

- Limite commune à deux systèmes, deux ensembles, deux appareils.

- Informatique : dispositif qui permet la **communication** entre deux éléments d'un système informatique.
  - Exemple : une interface graphique permet la communication entre un utilisateur et un système informatique.

- Sens figuré : relation.
  - Exemple : assurer **l'interface** avec un client.



TELECOM Paris
IP PARIS

Rappels

- Les objets sont des **entités** qui communiquent par envois de **messages**.
- Les objets contiennent des valeurs appelées **attributs**.
- Parmi les attributs, certains sont de type **primitif** (nombres, caractères) et d'autres de type **référence** sur d'autres objets.
- Une **référence** sur un objet permet de lui envoyer un **message**.
- Pour chaque type de message que l'objet peut recevoir, la clase déclare une **méthode** associée au type de message.
- Cette méthode est une procédure qui est **exécutée** par l'objet lorsqu'il reçoit le type de message associé.

L’interface d’un objet

- La signature des méthodes d’une classe décrivent comment d’autres objets peuvent **communiquer** avec lui.
- Pour interagir avec un objet, il faut avoir une **référence** sur l'objet et connaître son **interface**.
- Connaître l'interface d'un objet, c'est connaître les **signatures** de ses méthodes et bien sûr la documentation associée.
- En Java, il est possible de matérialiser l'interface d'un objet en utilisant des **déclarations d'interface**.
- Ne pas confondre les interfaces de programmation, objets de ce cours, avec les interfaces graphiques (IHM).

Exemple: un canevas pour dessiner des figures

```java
public interface Canvas {

    int getWidth();

    int getHeigth();

    Collection<Figure> getFigures();

    String getName();
}
```


TELECOM Paris IP PARIS

Une interface n’est pas une classe

- Une classe peut déclarer **mettre en œuvre** ou **implémenter une ou plusieurs** interfaces.

- Une interface peut être vue comme un **engagement** à mettre en œuvre les **méthodes** déclarées dans l'interface.

- L'interface **ne dit rien** sur la mise en œuvre de ses méthodes.

Exemple d'implantation de l'interface Canvas

```java
public class BasicCanvas implements Canvas {

    private final int width;
    private final int heigth;
    private final String name;
    private final List<Figure> figures;

    public BasicCanvas( final int width,
                        final int heigth,
                        final String name ) {
        this.width = width;
        this.heigth = heigth;
        this.name = name;

        figures = new ArrayList<>();
    }

    @Override
    public int getWidth() {
        return width;
    }

    @Override
    public int getHeigth() {
        return heigth;
    }

    @Override
    public Collection<Figure> getFigures() {
        return figures;
    }

    @Override
    public String getName() {
        return name;
    }
}
```

Mise en œuvre d'une interface

- Si une classe déclare qu'elle implémente une interface, elle doit proposer une implémentation de toutes méthodes déclarées dans l'interface.
  - Sauf si la classe est **abstraite**.

- La classe peut bien sûr proposer d'autres méthodes qui ne sont pas déclarées dans l'interface.

- Le compilateur est vigilant !

TELECOM Paris IP PARIS

Nommage des interfaces

- Nous avons déjà introduit une interface nommée **`Canvas`**.
- En Java, il y aura une collision de noms si ces deux éléments sont déclarés dans un même package. Afin d'éviter cela, nous avons donc nommé différemment notre classe implémentant l'interface (**`BasicCanvas`**).
- Nous aurions également pu nommer notre interface **`CanvasInterface`** (ou encore **`ICanvas`** pour faire plus court).
- Une bonne pratique consiste cependant à utiliser un nom générique pour l'interface dont le niveau d'abstraction est plus **élevé** que celui de la classe.
- On donnera alors des noms **plus spécifiques** aux différentes implémentations en fonction de leurs caractéristiques.
- Exemple:

```
public class BasicCanvas implements Canvas {
    ...
    /* This implementation optimizes resources consumption...*/
    public class OptimizedCanvas implements Canvas {
    ...
```

Interfaces comme outil de conception

- Les interfaces sont un outil de conception :
  - Elles doivent être utilisées lors de la phase de conception du logiciel.
  - Lors de cette phase, on identifie les différents types d'objets du problème.
  - Chaque objet est alors caractérisé par son interface documentée.

- On pense l'interface avant de penser la classe :
  - Si elle est bien commentée, l'interface apparaît comme une **spécification des méthodes** que les objets doivent implémenter.

TELECOM Paris IP PARIS

Exemple : l'interface `List`

- Recherche : JAVA SE List

- Exemples de deux implémentations :
  - `ArrayList`
  - `LinkedList`

- **`ArrayList`** : Implémentation stockant les éléments sous forme d'un **tableau** (**`Array`**).

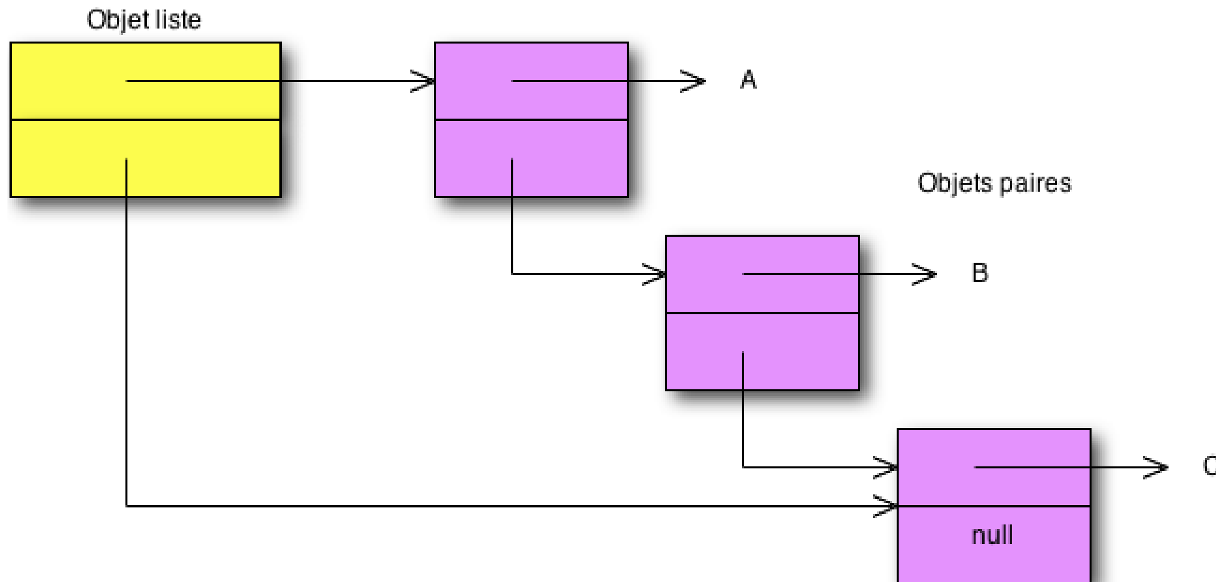
- **`LinkedList`** : Implémentation stockant les éléments sous forme d'une **liste chaînée**.

TELECOM Paris IP PARIS

Liste chaînée

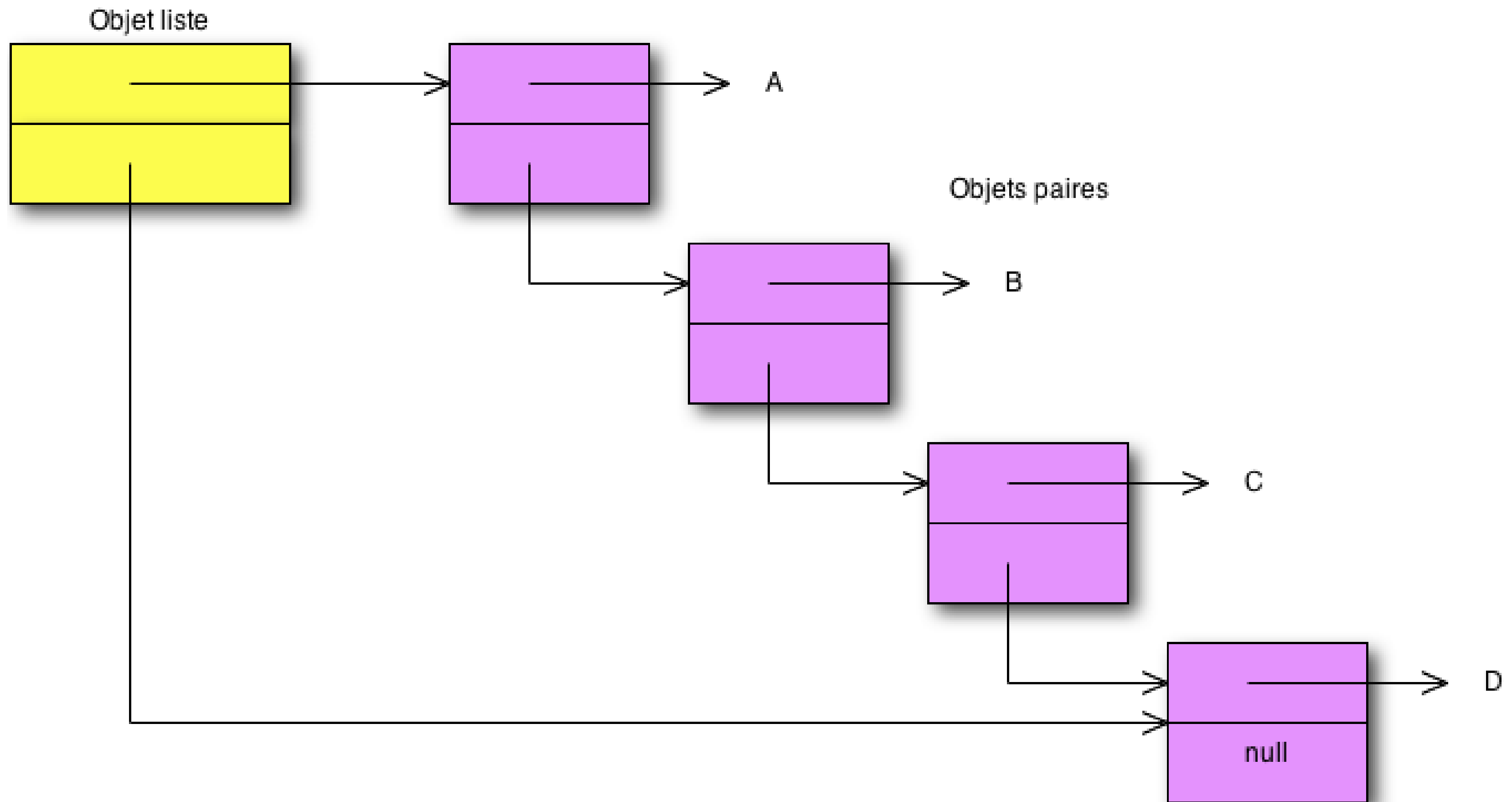
- Structure de données permettant de représenter une liste d'objets.

- Elle est réalisée à l'aide d'**objets auxiliaires** (les paires) liés par des **références** :
  - Une paire possède une **référence** sur la paire suivante.



TELECOM Paris
IP PARIS

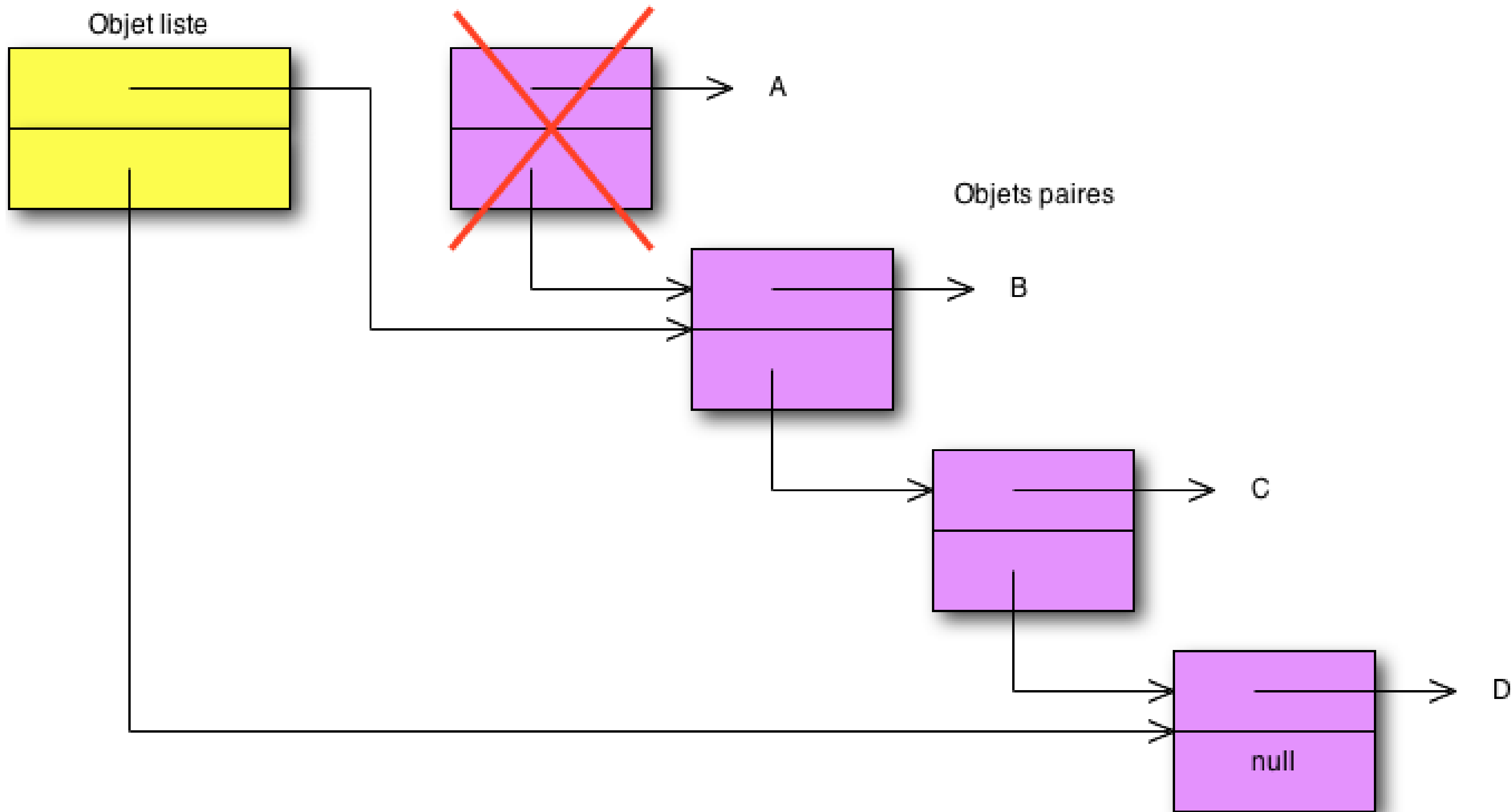
Ajouter un élément D en fin de la liste chaînée d'objets A, B, C…



TELECOM Paris

IP PARIS

Retirer le premier élément de la liste chaînée d'objets A, B, C, D…



TELECOM Paris

IP PARIS

Interfaces comme outil contractuel

- On voit donc que l'on peut utiliser une structure de liste chaînée pour implémenter une liste :
  - Ajouter un objet à la liste revient à ajouter cet objet en fin de liste.
  - Prendre le premier objet dans la liste, c'est prendre le premier objet de la liste et le retirer de la chaîne.
  - Les concepteurs de la classe `LinkedList` ont donc déclaré que cette classe remplissait le contrat défini par l'interface **`List`**.
  - Notons que cela n'empêche pas la classe `LinkedList` de proposer bien d'autres méthodes spécifiques aux files telles que celles de l’interface `Queue`.

- On peut également utiliser une autre structure de données telle que les tableaux (**`Array`**) pour implémenter une liste :
  - Ajouter un objet à la liste revient à ajouter cet objet en fin du tablau sous-jacent et d’augmenter sa taille au besoin.
  - Prendre le premier objet dans la liste, c'est prendre le premier objet du tableau et le retirer du tableau.
  - Les concepteurs de la classe `ArrayList` ont donc déclaré que cette classe remplissait le contrat défini par l'interface **`List`**.

- L’interface **`List`** sert donc de **contrat**.

TELECOM Paris IP PARIS

Interfaces comme outil d'encapsulation

- En utilisant l'interface comme un **type** pour une variable, on **restreint** l'accès aux méthodes de l'objet.

- Exemple:

```
Queue<E> myQueue = new LinkedList<E>();
E myElement = myQueue.get(0);
myQueue.add(myElement);
```

Est-ce que ça compile?

Est-ce que ça compile?

- Réponse : seules les méthodes de q déclarées dans l'interface **`Queue`** sont accessibles.
  - Les autres méthodes de la classe **`LinkedList`** sont cachées.

- Alors que l'objet **`myQueue`** est en réalité une **`LinkedList`**, toutes les opérations possibles sur une liste chaînée sont inaccessibles.

- **`myQueue`** est devenue une file d'attente de type **`Queue`** et le programmeur est **obligé** de s'en servir comme d'une file d'attente.
  - Il ne peut plus s'en servir comme d'une **`LinkedList`**.

TELECOM Paris IP PARIS

Interfaces pour présenter une vue particulière

- Exemple:

```
Queue<E> myQueue = new LinkedList<E>();
```

- L'interface **Queue** sert également à **voir** la liste chaînée comme étant une file.

- Elle présente donc une **vue particulière** sur une liste chaînée.

- L'objet fourni est le même mais les possibilités d'accès aux méthodes de l'objet ne sont pas les mêmes :
  - Seulement celles de la vue sont offertes.

Propriétés en commun d'objets d'identité différentes

- Des classes sans lien entre elles peuvent implémenter la **même interface.**

- Dans ce cas, on dira que l'interface décrit une **propriété** que des classes ont en commun et peuvent implémenter **différemment.**

- Prenons l'exemple d'un éditeur graphique permettant de dessiner en utilisant des formes prédéfinies : carré, rectangle, cercle, etc.

- Il y aura donc différentes classes pour différents types de figures telles que `Square`, `Rectangle`, `Circle`, etc.

- L'éditeur comprendra une zone de dessin sur laquelle seront dessinées les figures.
  - Celle-ci sera représentée par un objet de la classe `Graphics`.

TELECOM Paris IP PARIS

Propriétés en commun d’objets d’identité différentes

- Toutes les classes de figures devront implémenter l'interface **`Drawable`** :

```
public interface Drawable {

    void paint(Graphics graphics);
}
```

- Ainsi le programme de l'éditeur pourra fournir un objet **`graphics`** de type **`Graphics`** et **demander** à tous les objets en édition de se dessiner en envoyant le message **`paint(graphics)`** à chacun d'eux.

TELECOM Paris IP PARIS

Propriétés en commun d'objets d'identité différentes

- Si l'utilisateur de l'éditeur graphique a demandé l'affichage d'une grille pour l'aider à positionner les objets, celle-ci devra s'afficher en fond dans la zone d'édition.

- On aura donc une classe **`Grid`** :

```
public class Grid implements Drawable {

    @Override
    public void paint(Graphics graphics) {
        ...
    }

    ...
}
```

- Ainsi, le programme de l'éditeur traitera la grille de type **`Grid`** comme les figures au moment de dessiner, bien que la grille et les figures ne soient **pas de même nature**.

TELECOM Paris IP PARIS

Héritage d'interfaces

- Une interface peut **hériter** d'autres interfaces.

- Cela signifie que cette interface contient ses propres déclarations de méthodes mais aussi, implicitement, les **déclarations de méthodes des interfaces dont elle hérite**.

- On peut bien sûr se passer de cette possibilité puisqu'une classe peut implémenter plusieurs interfaces.

- Cependant, regrouper ces interfaces sous une seule interface peut simplifier la programmation et expliciter le modèle OO utilisé.

Exemple d'héritage d'interfaces

```
public interface Whistling {
    void whistle();
}

public interface Flying {
    void fly();
}

public interface Bird extends Whistling, Flying {
}

public class BasicBird implements Bird {
    ...
}

public interface Walking {
    void walk();
}

public interface Human extends Whistling, Walking {
}

public class BasicHuman implements Human {
    ...
}
```


TELECOM Paris IP PARIS

Conflits de noms

- Lorsqu'une interface hérite de plusieurs autres interfaces, il peut apparaître des déclarations de méthodes de **même nom**.
- Si les deux déclarations de méthodes héritées ont des en-têtes identiques, il n'y a pas de problème. La classe implémentant l'interface devra implémenter cette méthode une seule fois.
- Si les deux déclarations ont un même nom de méthode mais des paramètres différents, en types ou en nombre, il n'y a pas de problème. La classe implémentant l'interface devra implémenter les deux méthodes.
- Le problème apparaît lorsque deux déclarations de méthodes ont même nom et mêmes paramètres mais un **type de retour différent**. Dans ce cas, le compilateur Java refuse de compiler le programme car il y a un **conflit de nom**.
- N.B. Le conflit de noms s'applique également lors de l'héritage de classes. On ne pourra déclarer deux méthodes de même nom, de mêmes paramètres mais de type de retours différents que dans le cas où le type de retour de la sous-classe **hérite** du type de retour de la classe mère.

Exemple

```
public class RectangularDimension extends Dimension {

    private final int width;
    private final int heigth;

public abstract class Component implements Figure {
    …

    public Dimension getDimension() {
        return dimension;
    }
    …
}

public class Area extends Component {
   …
    @Override
    public RectangularDimension getDimension() {
        return (RectangularDimension) super.getDimension();
    }
```

Pas de conflit car `RectangularDimension` hérite de `Dimension`

TELECOM Paris IP PARIS

Constantes dans les interfaces

- Une interface contient des déclarations de méthodes.
  - Elle ne contient pas d'implémentation de ces méthodes, sauf les méthodes dites de type **`default`** (à partir de Java 8, pas au programme de ce cours).

- On peut également déclarer des **constantes** dans une interface.
  - Elles se déclarent avec les mots clés **`static`** et **`final`** :

```
public interface MathConstants {
    static final double PI = 3.1416;
}
```

- Si des classes doivent partager les mêmes constantes, le mieux est de les déclarer dans une interface.

- Dans n'importe quelle classe, on peut accéder à cette constante en écrivant **`MathConstants.PI`** car la constante est déclarée **`public`**.

- Dans une classe donnée, on peut **directement** accéder à cette constante (en écrivant simplement **`PI`**) à condition que cette classe implémente l'interface **`MathConstants`**.

TELECOM Paris
IP PARIS

Interfaces comme marqueurs

- Java propose de nombreuses structures de données de type collections d'objets : **`ArrayList`**, **`LinkedList`**, etc.

- Un objet de type **`ArrayList`** permet un accès à ses éléments par un index numérique. Cet accès se fait en **temps constant**.
  - Le temps d'accès à l'élément d'index **n** est majoré par une **constante** ne dépendant pas de **n**.

- Un objet de type **`LinkedList`** permet également d'accéder à ses éléments par un index numérique. Cependant, pour accéder à un élément, il faut partir de la première paire de la liste et suivre le chaînage des paires jusqu'à l'élément voulu. Cela ne se fait plus en temps constant mais en **temps linéaire**.
  - Cela signifie que le temps d'accès à l'élément d'index **n** est majoré par une fonction affine de **n**.

TELECOM Paris IP PARIS

Interfaces comme marqueurs

- La classe **`Collections`** du JDK propose différents algorithmes de tri pour réordonner une collection d'objets en fonction d'un ordre que l'on spécifie.

- Or l'algorithme de tri optimal ne sera pas le même selon que l'on peut accéder à un élément en temps constant ou pas.

- Il existe des algorithmes spécialisés pour les tables (accès par index en temps constant) et d'autres spécialisés pour les listes chaînées (accès par index en temps linéaire).

- Comment l'algorithme de tri proposé par la classe **`Collections`** peut-il savoir si l'accès par index aux éléments de la liste est en temps constant ou linéaire ?

- Java propose que le programmeur l'indique explicitement. Mais comment ?

Interfaces comme marqueurs

- Java introduit une interface **vide** nommée **`RandomAccess`** :

```
public interface RandomAccess {
}
```

- Le programmeur qui veut indiquer que sa classe de collection d'objets a un accès par index en temps constant déclare simplement que sa classe implémente l'interface **`RandomAccess`**.

- L'algorithme de tri générique de la classe **`Collections`** peut savoir si la collection qu'on lui soumet permet un accès par index en temps constant en testant si l'objet implémente l'interface **`RandomAccess`**. Pour cela, il utilise l'expression suivante :

```
if (collection instanceof RandomAccess) {
    ...
}
else {
    ...
}
```

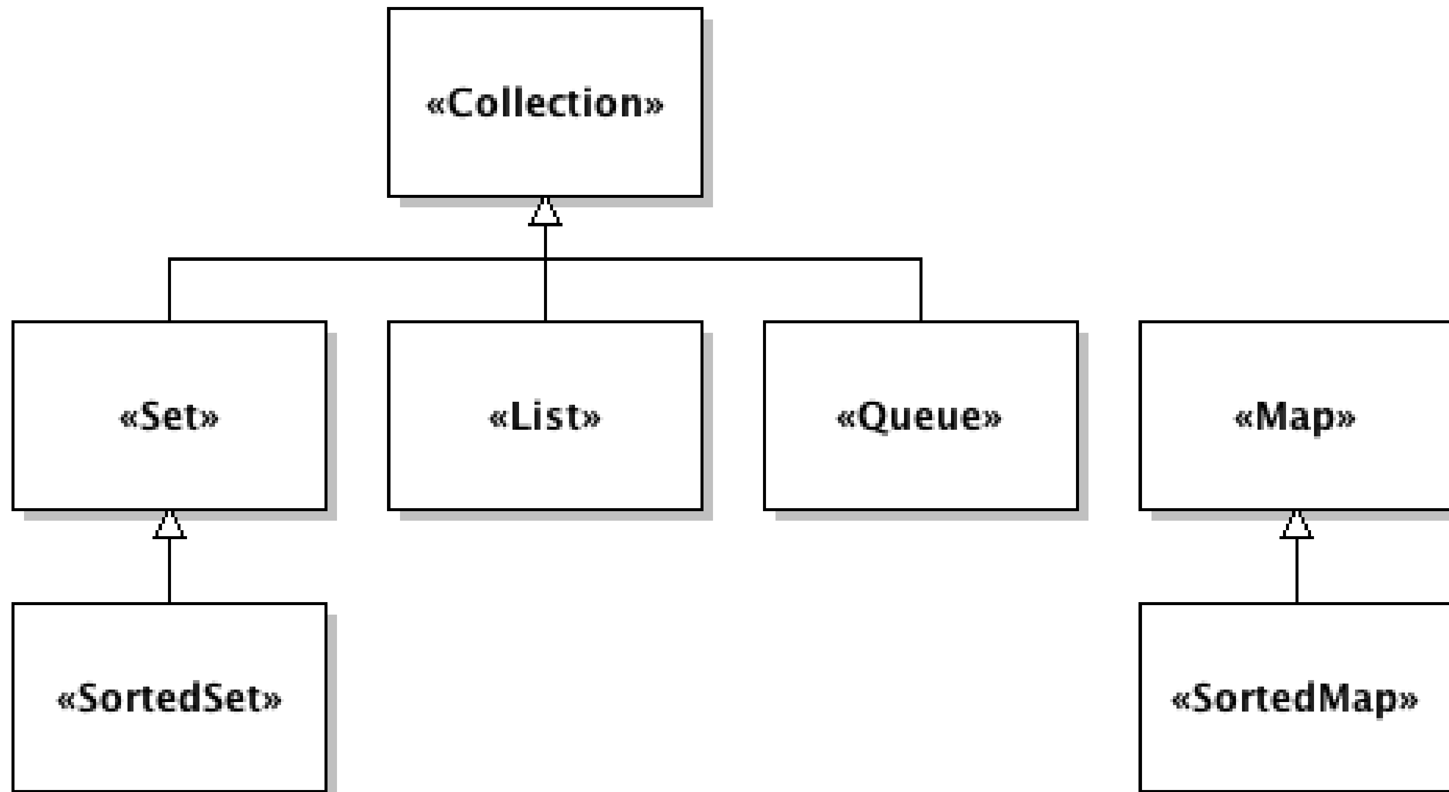
- Ceci explique l'existence de nombreuses **interfaces vides** dans le JDK.

TELECOM Paris IP PARIS

Interfaces de collections

- Une collection est une structure de données qui regroupe un nombre variable d'objets en un seul objet.
- Les collections sont représentées en Java par des classes de mise en œuvre. Il existe différentes manières de regrouper des objets selon l'usage que l'on compte faire de la collection.
- Les collections sont également représentées par des interfaces qui définissent les vues que l'on peut avoir sur les objets qui les implémentent.
- Les collections sont enfin représentées par un ensemble d'algorithmes qui permettent de les manipuler.

Diagramme des interfaces de collections



TELECOM Paris
IP PARIS

L'interface `Collection`

- L'interface **`Collection`** est la racine de l'arbre.
  - Elle contient ce qui est commun à toutes les collections.

- Recherche : JAVA SE **`Collection`**

TELECOM Paris IP PARIS

L'interface Set

- L'interface **`Set`** représente un ensemble fini d'objets.

- Elle ne peut pas contenir deux fois le même élément :
  - Un appel à la méthode **`add(element)`** retournera **`false`** et l'élément ne sera pas ajouté.

- L'ordre des objets n'est pas garanti :
  - Les objets ne seront pas nécessairement récupérés dans le même ordre que celui dans lequel ils ont été insérés.

- Recherche : JAVA SE **`Set`**

TELECOM Paris IP PARIS

L’interface `SortedSet`

- L'interface **`SortedSet`** représente un ensemble fini d'objets ordonnés.

- Elle ne peut pas contenir deux fois le même élément :
  - Un appel à la méthode **`add(element)`** retournera **`false`** et l’élément ne sera pas ajouté.

- L’ordre des objets est garanti :
  - Une **fonction d'ordre** peut être fournie.

- Recherche : JAVA SE **`SortedSet`**

TELECOM Paris IP PARIS

L’interface `Map`

- L'interface **`Map`** représente une table d'associations **clé - valeur**.
  - Également appelée **table de hachage**.

- L’ordre des paires clé – valeur n’est pas garanti.

- Recherche : JAVA SE `Map`

TELECOM Paris
IP PARIS

L’interface `SortedMap`

- L'interface **`SortedMap`** représente une table d'associations clé – valeur mais avec un ordre sur les clés.
  - Fonction d'ordre sur les clés à fournir.
  - Eléments maintenus en ordre ascendant des clés.

- Peut servir à modéliser un annuaire ou un répertoire téléphonique.

- Recherche : JAVA SE **`SortedMap`**

TELECOM Paris IP PARIS

Comment choisir les implémentations des interfaces de collection ?

- Exemple pour l'interface `Map` :

- `Hashtable` :
  - L'accès aux éléments est **synchronisé**, ce qui permet de gérer du parallélisme.
  - La synchronisation introduit un surcoût en temps d'exécution.

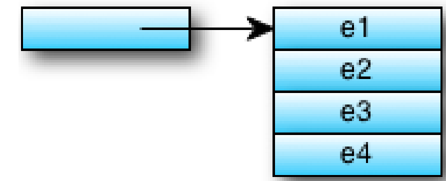
- `HashMap` :
  - L'accès aux éléments n'est pas **synchronisé**, ce qui ne permet pas de gérer du parallélisme.
  - Pas de surcoût en temps d'exécution.

TELECOM Paris IP PARIS

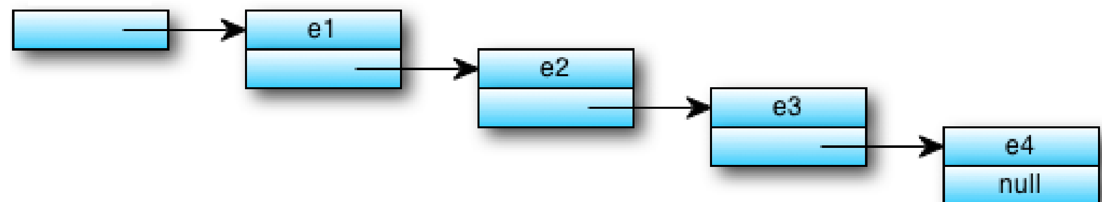
Comment choisir les implémentations des interfaces de collection ?

- Exemple pour l'interface `List` :

- `ArrayList` :
  - L'insertion ou l'enlèvement d'éléments est **coûteux**.
  - L'accès à un élément se fait en **temps constant**.



- `LinkedList` :
  - L'insertion ou l'enlèvement d'éléments est **peu couteux**.
  - L'accès à un élément se fait en temps **linéaire**.



- En résumé, il faut bien connaître le **besoin** de l'application, bien **étudier** les différentes options pour choisir la **bonne implémentation**.

TELECOM Paris
IP PARIS

Programmer à l'interface

- Jusqu'ici, lorsque nous avons eu besoin d'une variable de type liste d’objets, nous l'avons toujours déclarée en tant que **`ArrayList`** :

```
public class Factory extends Component {
    private final ArrayList<Component> components;

    public Factory() {
        components = new ArrayList<Component>();
    }

    public ArrayList<Component> getComponents() {
        return components;
    }
    ...
}
```


TELECOM Paris IP PARIS

Programmer à l'interface

- Cependant, que se passera-t-il si on découvre que dans certains contextes d'utilisation, l'implémentation **ArrayList** n'est pas assez performante?

- Il faudra alors retrouver dans le code **toutes les déclarations** où la classe **ArrayList** a été utilisée (attributs, variables locales, paramètres de méthode, etc.) et **changer** toutes ces déclarations afin de pouvoir utiliser la bonne implémentation.

- Cela risque d'être très fastidieux, en particulier pour les applications riches pouvant contenir des **milliers de classes** et devant être maintenues pendant de très **nombreuses années**...

- Ainsi, une bonne pratique pour éviter ce problème consiste à **programmer à l'interface**.

Programmer à l'interface

- A cette fin nous utiliserons plutôt l'interface **`List`** pour déclarer nos attributs, variables locales, paramètres de méthode, etc.

```
public class Factory extends Component {
    private final List<Component> components;

    public Factory() {
        components = new ArrayList<Component>();
    }

    public List<Component> getComponents() {
        return components;
    }
    ...
}
```


TELECOM Paris

IP PARIS

Programmer à l'interface

- Ainsi, si nous devons changer l'implémentation de la liste, il suffira alors de ne changer que le code **d'instanciation** de la liste, et tout le reste du code où la liste sera utilisée compilera peu importe l'implémentation choisie :

```
public class Factory extends Component {
    private final List<Component> components;

    public Factory() {
        components = new LinkedList<Component>();
    }

    public List<Component> getComponents() {
        return components;
    }
    ...
}
```


TELECOM Paris
IP PARIS

Programmer à l'interface

- Programmer à l'interface (c'est-à-dire déclarer l'**interface** plutôt que la **classe**) permet donc d'améliorer grandement la **maintenabilité** du code en faisant **abstraction** de l'implémentation qui sera concrètement utilisée à l'exécution du programme.

- De plus, l'interface n'expose que les méthodes **communes** des classes implémentant l'interface, ce qui empêche les développeurs d'utiliser des méthodes **spécifiques à l'implémentation** qui rendraient le code plus difficile à maintenir lorsque l'implémentation change.

TELECOM Paris IP PARIS

Collections itérables

- Toutes les collections supportent la boucle **`for`** généralisée.

- Si **`coll`** est une collection d'objets de type **`Data`** et que cette collection implémente l'interface **`Iterable`**, on peut parcourir cette collection la boucle **`for`** généralisée :

```
Collection<Data> coll = new ArrayList<>();

for (Data data : coll) {
    data.doSomething();
    ...
}
```

- **Toutes** les collections supportent la création d'**itérateurs** qui permettent également de les parcourir.

TELECOM Paris IP PARIS

Les algorithmes de collection

- La classe **`Collections`** contient plusieurs méthodes de classe (**`static`**) mettant en œuvre des algorithmes sur les collections.

- Recherche : JAVA SE **`Collections`**

TELECOM Paris IP PARIS

Conclusion

- Nous avons vu différents usages des interfaces Java :
  - L'interface comme **outil de conception**. On décrit les interfaces de chacun des objets d'un problème avant d'implémenter ces interfaces par des classes. On pense l'interface avant la classe.
  - L'interface comme **engagement contractuel**. L'interface est vue comme un **contrat** que la classe doit **respecter**.
  - L'interface comme **outil d'encapsulation**. L'interface permet de ne montrer qu'une certaine **facette** d'un objet en dissimulant le reste.
  - L'interface comme **descriptif de propriétés**. L'interface permet de décrire une propriété **partagée** par **différentes classes** qui n'ont a priori aucun lien entre elles.

- Les interfaces peuvent être **composées** via l'héritage.

- **Programmer à l'interface** est une excellente pratique :
  - Favorise la **maintenabilité** du code en facilitant le **changement** des implémentations, même au runtime…

- Dans un bon programme on déclare à l'interface et on **centralise l'instanciation** dans une méthode (ou classe) dont c'est le **rôle**.
  - Voir le design pattern Factory Method.

TELECOM Paris IP PARIS