Programmation orientée objet en Java

Les interfaces de programmation

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

Objectifs d'apprentissage

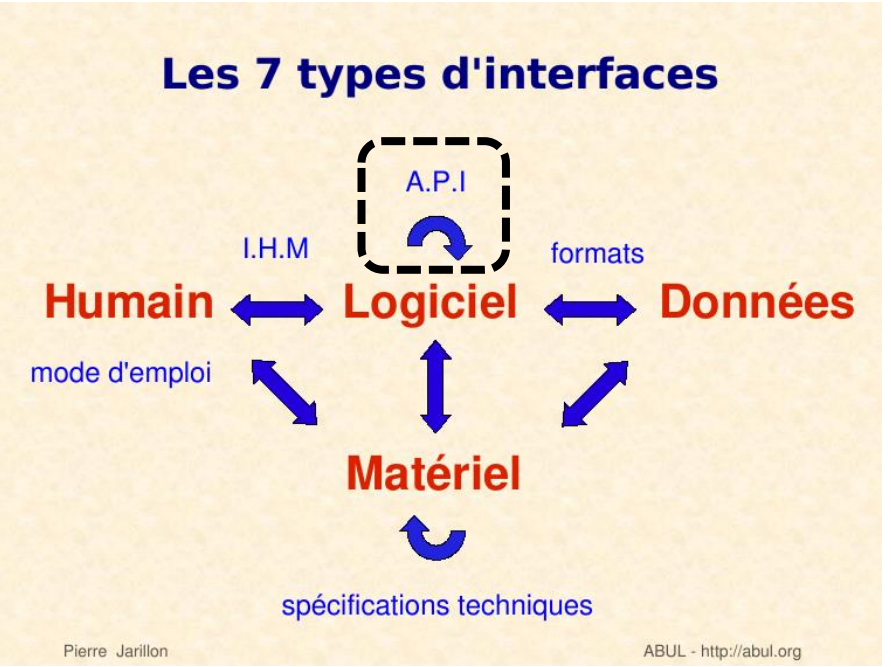
- Notion d'interface de programmation
- Composition (héritage) d'interfaces
- Interfaces de constantes et de marquage
- Les principales interfaces des collections du JDK
- Programmer à l'interface

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.

- Pour ce cours, on s’intéresse aux interfaces de type **logiciel - logiciel**.

- Également appelées API :
  - Application Programming Interface
  - Interface de programmation d'application



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

```
public interface Canvas {

    String getName();

    int getWidth();

    int getHeight();

    ArrayList<Figure> getFigures();
}
```

Une interface n’est pas une classe

- Une classe peut déclarer **mettre en œuvre** ou **implémenter** une ou **plusieurs interfaces** définies par ses méthodes **visibles**.
- 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 manière dont sont mises en œuvre ses méthodes.

Exemple de mise en œuvre d'une interface

```
public class BasicCanvas implements Canvas {

    private final int width;
    private final int height;
    private final String name;
    private final ArrayList<Figure> figures;

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

        figures = new ArrayList<>();
    }

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

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

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

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

```
public interface Canvas {

    String getName();

    int getWidth();

    int getHeight();

    ArrayList<Figure> getFigures();
}
```

- Si une classe déclare qu'elle **implémente** une interface, elle doit proposer une implémentation de **toutes** les 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 !

Remarques sur le nommage des interfaces

- Nous avons introduit une interface nommée **`Canvas`**. En Java, il y aura une **collision de noms** si la classe et l'interface ont le **même nom** et 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 la class **`Canvas`** et son 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** :

```
/* This is a default basic implementation...*/
public class BasicCanvas implements Canvas {
    ...

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

Interfaces comme *outil de conception*

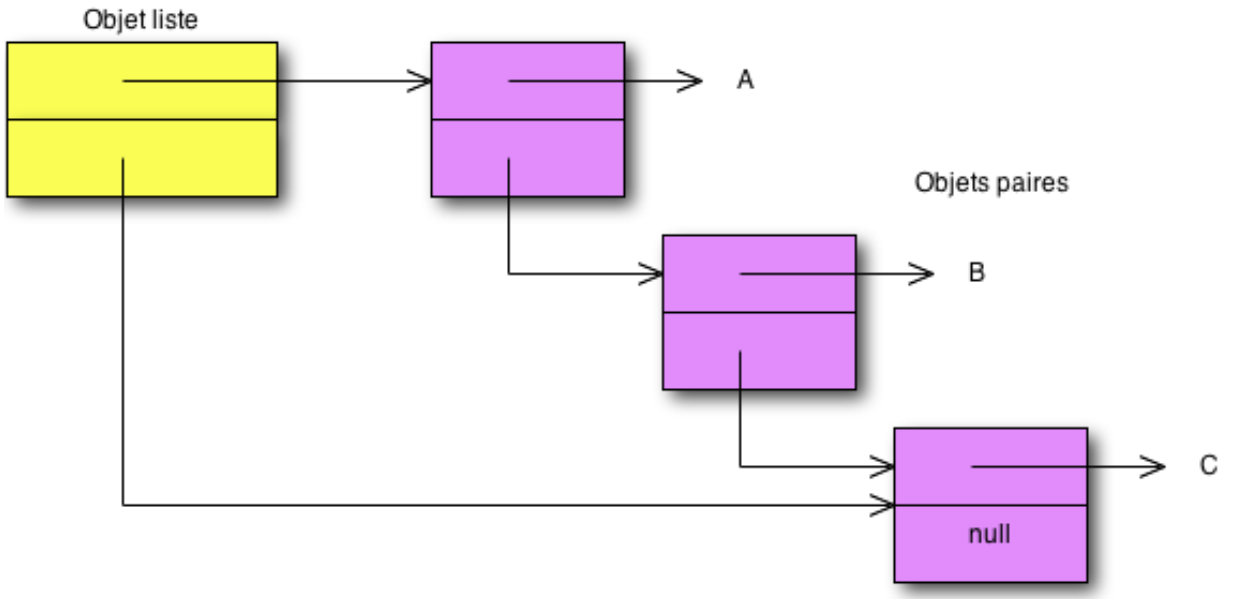
- Les interfaces sont un outil de conception :
  - Elles doivent être utilisées lors de la **phase de conception** du logiciel.
- Lors de cette phase, on identifie les différents **types** d'objets du problème.
  - Chaque objet est alors caractérisé par son **interface documentée**.
- On pense l'interface avant de penser la classe :
  - Si elle est bien commentée, l'interface apparaît comme une **spécification des méthodes** que les objets doivent implémenter.

Exemple : l'interface `List`

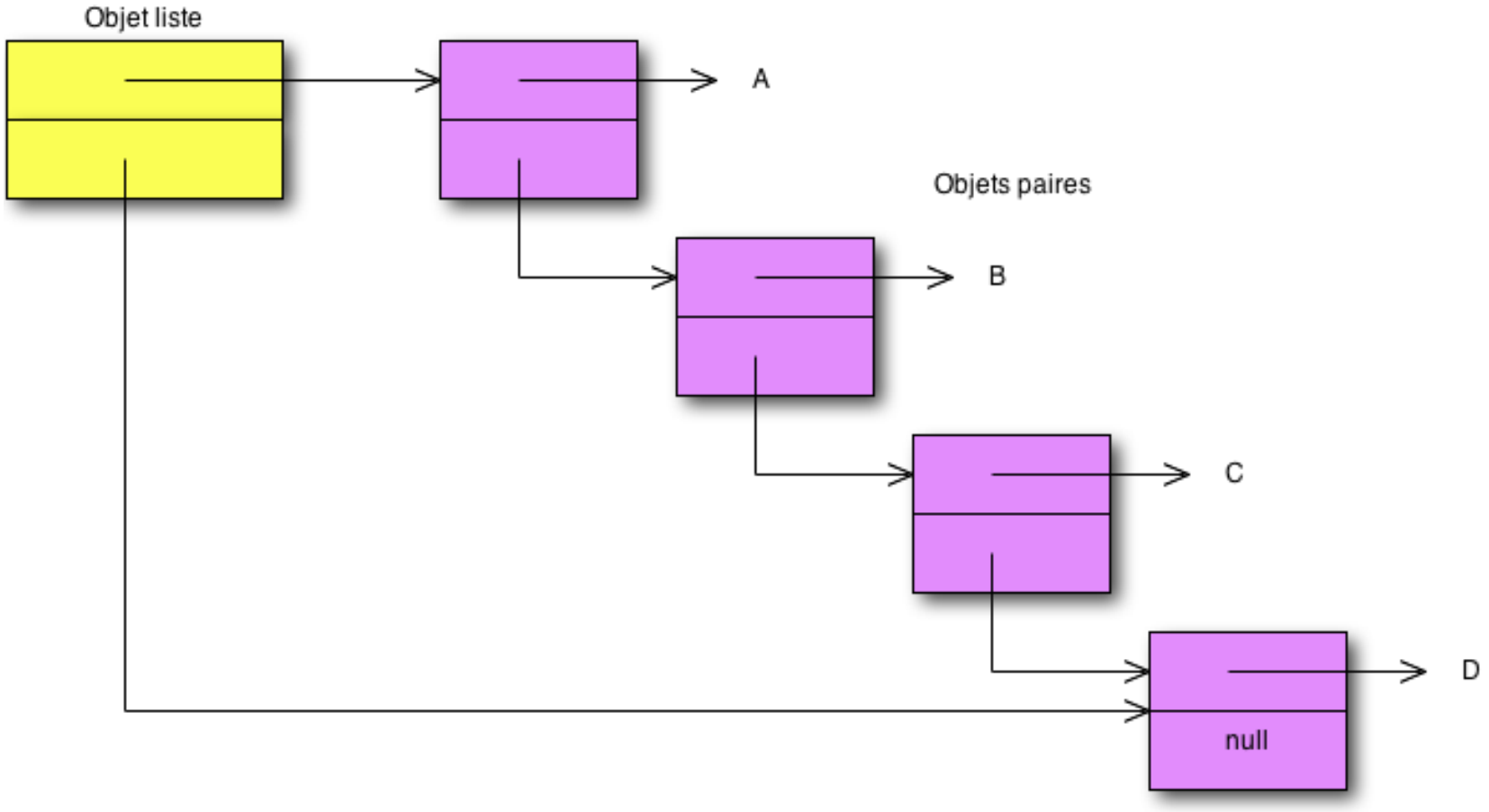
- Recherche : JAVA SE List
- Exemples pour deux de ses implémentations :
  - **`ArrayList`**
  - **`LinkedList`**
- **`ArrayList`** : Implémentation stockant les éléments sous forme d'un **tableau** (**`Array`**).
- **`LinkedList`** : Implémentation stockant les éléments sous forme d'une **liste chaînée**.

Structure d'une liste chaînée

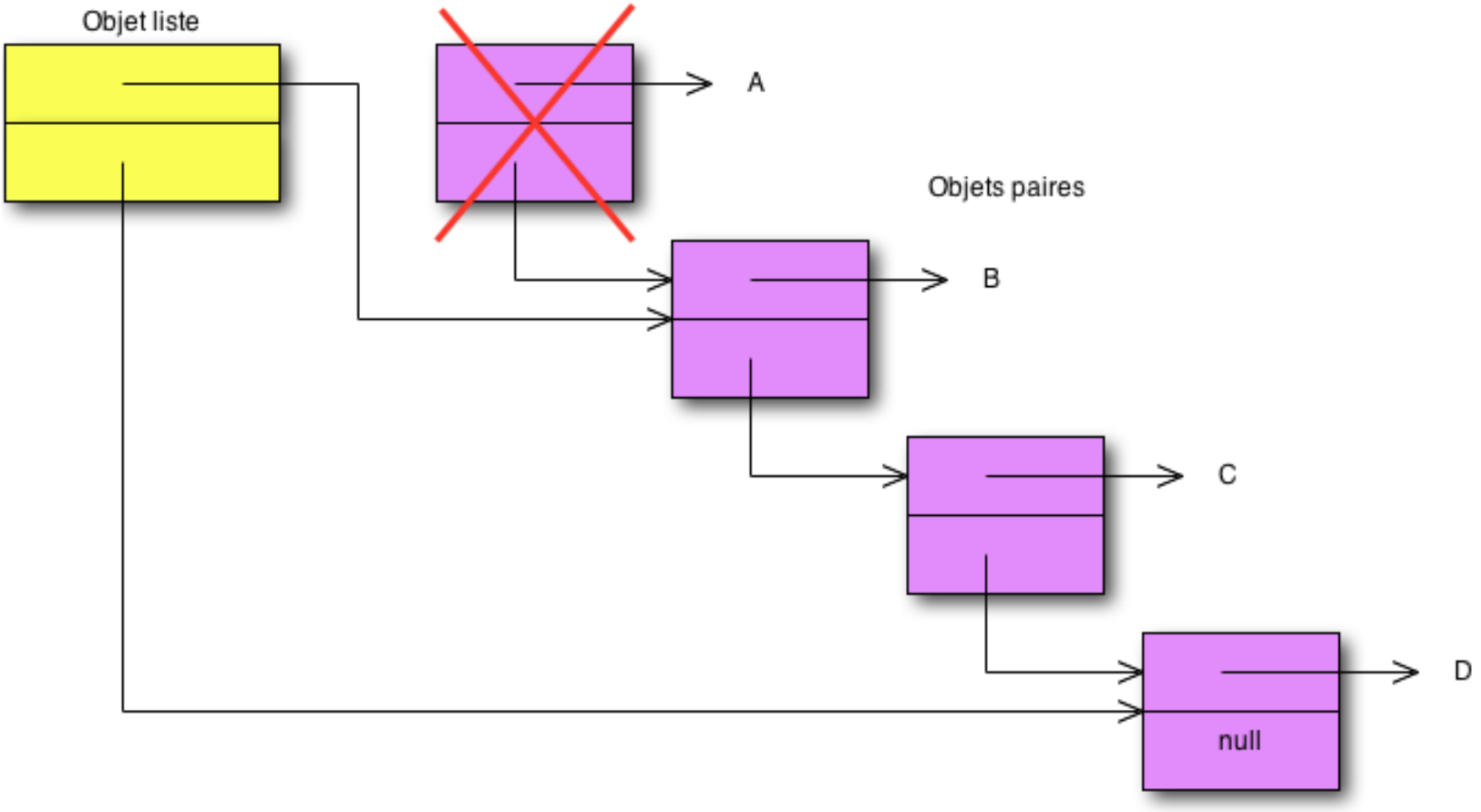
- Elle est réalisée à l'aide d'**objets auxiliaires** (les paires) liés par des **références**.
- Une paire possède une **référence** sur la paire suivante.
- Le **dernier objet** de la liste possède une référence **nulle** sur la paire suivante.



Ajouter un élément en fin de la liste



Retirer le premier élément de la liste



Interfaces comme *outil contractuel*

- Une structure de liste chaînée peut être utilisée 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 cette classe de proposer plusieurs autres méthodes spécifiques aux files telles que celles de l'interface Queue.

- Un tableau (**`Array`**) peut également être utilisé pour implémenter une liste :
  - Ajouter un objet à la liste revient à ajouter cet objet en fin du tableau 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**.

Interfaces comme *outil d'encapsulation*

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

- Question :

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

Est-ce que ça compile?

Est-ce que ça compile?

- Allez voir la documentation…

Interfaces comme outil d'encapsulation

```
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 **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 instance de **`LinkedList`**, les opérations spécifiques à cette classe 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`**.

Interfaces pour *présenter une vue particulière*

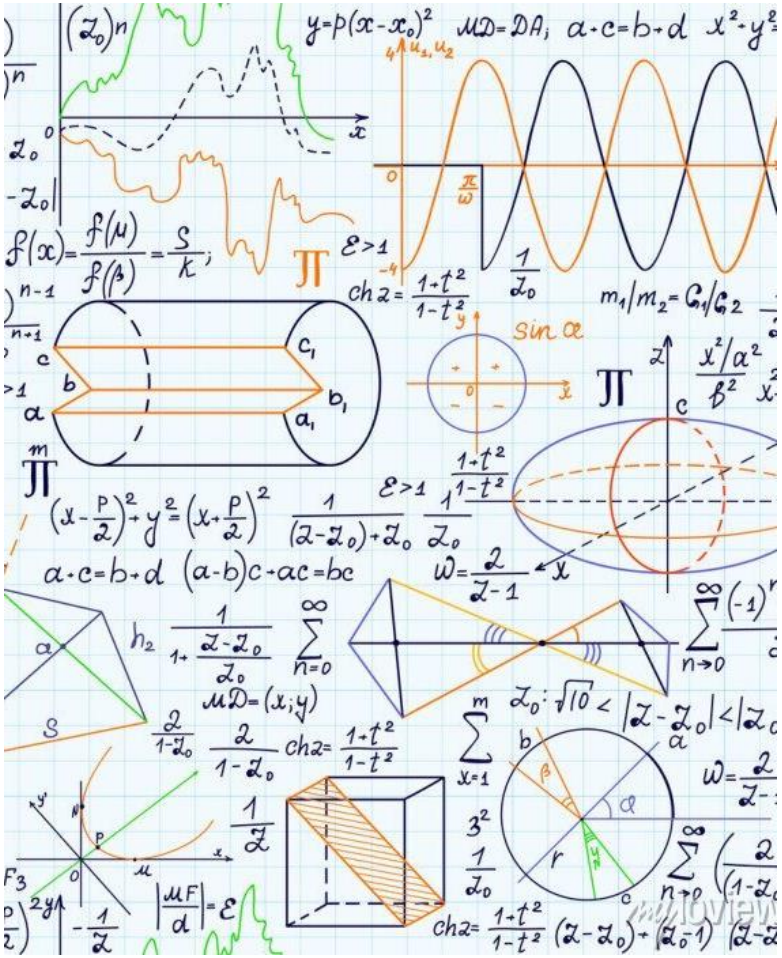
- Exemple :

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

- L'interface **`Queue`** sert à **voir** la liste chaînée comme étant une **file**.
- Elle présente donc une **vue particulière** sur la 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'*identités différentes*

- Des classes **sans lien entre elles** peuvent implémenter la **même interface**.
- Dans ce cas, on dira que l'interface décrit une **propriété** que ces classes **ont en commun** et peuvent **implémenter différemment**.
- Exemple d'un éditeur graphique permettant de dessiner en utilisant des formes prédéfinies :
  - Carré, rectangle, cercle, etc.
- Il y aura différentes classes pour différents types de formes telles que **Square**, **Rectangle**, **Circle**, etc.
- L'éditeur comporte une zone de dessin (canevas) quadrillée sur laquelle sont dessinées les formes.
  - Celle-ci sera représentée par un objet de la classe **Canvas**.



Propriétés en commun d'objets d'identité différentes

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

```
public interface Drawable {
    void paint(Graphics graphics);
}
```

- Pour chaque figure à dessiner, l'éditeur graphique demandera à la figure de se dessiner en envoyant le message **`paint(graphics)`** et en passant un objet de type **`Graphics`**.

- Pour s'afficher, chaque figure saura comment utiliser cet objet **`graphics`**.

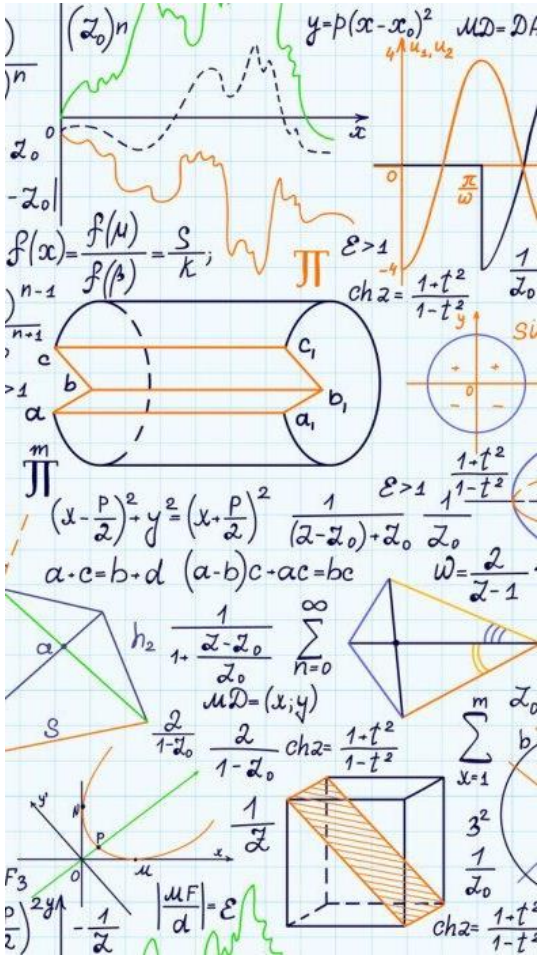
Propriétés en commun d'objets d'identité différentes

- Qu'en est-il de la grille du canevas ?
- On définira une classe **`Grid`** qui elle devra implémenter **`Drawable`** :

```
public class Grid implements Drawable {

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

- Ainsi, le programme de l'éditeur traitera la grille de type **`Grid`** comme toute autre forme au moment de l'affichage, bien que la grille et les autres formes ne soient **pas de même nature (identité)**.
- L'interface **`Drawable`** représente alors la **propriété de pouvoir se dessiner** qui est commune à ces deux classes d'identités différentes.



Composition (via l'héritage) d'interfaces

- Des interfaces peuvent être **composées** afin de définir une nouvelle interface.
- Cela se réalise via l'**héritage** d'une ou **plusieurs** 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 de composition, puisqu'une classe peut implémenter plusieurs interfaces.
- Cependant, composer ces interfaces en une seule interface peut simplifier la programmation et **expliciter** le modèle OO utilisé.

Exemple de composition d'interfaces via l'héritage

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

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 {
    ...
}
```

```
public interface Flying {
    void fly();
}
```

Conflits de noms

- Lorsqu'une interface hérite de plusieurs autres interfaces, des déclarations de ces différentes interfaces pourraient avoir des méthodes de **même nom**.

- Si deux déclarations de méthodes ont des en-têtes identiques, il n'y a pas de problème.
  - La classe implémentant l'interface ne devra implémenter cette méthode qu'**une seule fois**.

- Si les deux déclarations de méthodes 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**.

- Un problème se manifeste lorsque deux déclarations de méthodes ont un même nom de méthode, des mêmes paramètres de **signature identiques**, mais un **type de retour différent**.
  - Dans ce cas, le compilateur refuse de compiler le programme car il y a un **conflit de noms**.

- Ce type de conflit de noms s'observe également pour l'héritage de classes.
  - On ne pourra déclarer deux méthodes de même nom, même signature de paramètres mais de type de retours différents que dans le cas où le type de retour de la méthode de la sous-classe **hérite** du type de retour de la méthode de la classe mère.

Exemple

```
public class RectangularShape extends Shape {
    private final int width;
    private final int height;
    …
}

public abstract class Component {
    …
    public Shape getShape() {
        return shape;
    }
    …
}

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

Pas de conflit car `RectangularShape` *hérite* de `Shape`

Constantes dans les interfaces

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

- On peut également déclarer des **constantes** dans une interface.

```
public interface MathConstants {
    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`**.

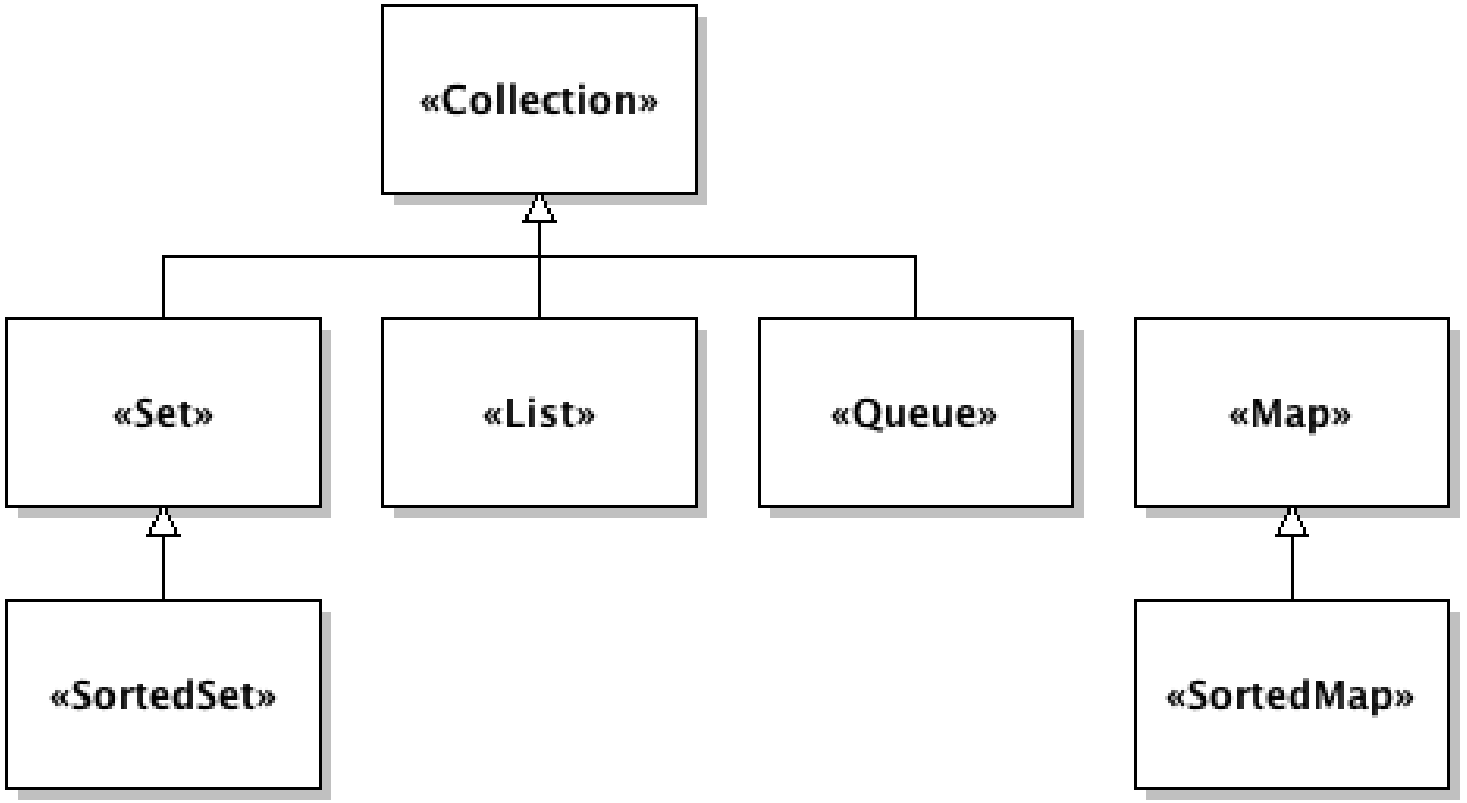
- Dans toute classe qui implémente l'interface **`MathConstants`**, on peut **directement** accéder à cette constante (en écrivant simplement **`PI`**).

```
public class Circle implements MathConstants {
    …
    public double getArea() {
        return PI * getRadius() * getRadius();
    }
    …
}
```

Interfaces des collections du JDK

- 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 également représentées par un **ensemble d'algorithmes** qui permettent de les **manipuler**.

Interfaces des collections du JDK



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`**

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`**

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`**

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`**

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`**

Comment choisir une implémentation adéquate d'une interface de collection ?

- Exemple pour l'interface **`Map`** :

- **`Hashtable`** :
  - L'accès aux éléments est **synchronisé**, ce qui permet de gérer des accès **concurrents** aux donnés.
  - En revanche, 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.
  - En revanche, il n'y a pas de surcoût en temps d'exécution.

Temps d'accès aux éléments des collections

- Java propose de nombreuses structures de données de type **collections d'objets** : **`ArrayList`**, **`LinkedList`**, etc.
  - Le temps d'accès aux éléments peut **dépendre** fortement de **l'implémentation**.

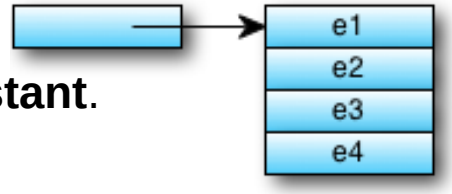
- Un objet de type **`ArrayList`** permet un accès à ses éléments par un index numérique. Cet accès se fait en **temps constant**.
  - Le temps d'accès à l'élément d'index **`n`** est majoré par une **constante** ne **dépendant pas** de **`n`**.

- Un objet de type **`LinkedList`** permet également d'accéder à ses éléments par un index numérique. 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. L'accès aux éléments ne se fait **plus** en temps constant.
  - Le temps d'accès à l'élément d'index **`n`** est **majoré** par une fonction affine de **`n`**
    - $f(x) = ax + b$.

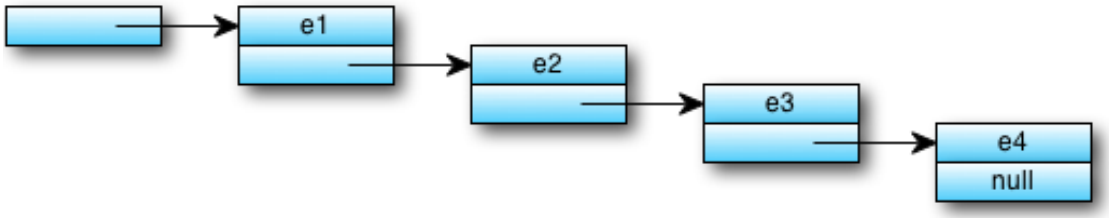
Comment choisir une implémentation adéquate d'une interface de collection ?

- Exemple pour l'interface **`List`** :

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



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



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

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;
    }
    ...
}
```

Programmer à l'interface

- Cependant, que se passera-t-il si on découvre que dans certains contextes d'utilisation du programme, l'implémentation **ArrayList** n'est pas adéquate au besoin ?
  - Par exemple, 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;
    }
    ...
}
```

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.
  - 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;
    }
    ...
}
```

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 le programmeur d'utiliser des méthodes **spécifiques à l'implémentation** qui rendraient le code plus difficile à maintenir lorsque l'implémentation doit changer.

- Cela permet également de changer d'implémentation **en cours d'exécution**, en fonction de changements des caractéristiques des données ou de l'environnement par exemple.
  - Self-adaptive software…

Collections itérables

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

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

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

for (Data myData : myCollection) {
    myData.doSomething();
     ...
}
```

- **Toutes** les collections du JDK supportent également la création d'**itérateurs** qui permettent également de les parcourir.
  - Voir le patron de conception Iterator.

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`**

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 peut spécifier.

- L'algorithme de tri optimal ne sera pas le même selon que l'on peut accéder à un élément en temps constant ou pas.
  - Par exemple, les temps d'accès de **`ArrayList`** et de **`LinkedList`** sont très différents…

- Il existe des algorithmes spécialisés pour les tableaux (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 permet au programmeur de l'indiquer explicitement. Mais comment ?

Interfaces comme marqueurs

- Java introduit une interface **vide** (ne déclarant aucune méthode) nommée **`RandomAccess`** :

```
public interface RandomAccess {
}
```

- Le programmeur qui veut indiquer que sa classe de collection 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`** pourra 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.

Conclusion

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

- 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 pendant l'exécution du programme…

- 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 patron de conception Factory Method.