

R1.01

INITIATION AU DÉVELOPPEMENT

Cours 9 : Liste chaînée













une structure de donnée dynamique

Hervé Blanchon & Anne Lejeune

Université Grenoble Alpes

IUT 2 – Département Informatique




Plan du cours

-  Type abstrait de données : Liste (classe `ListeInterface`)
-  Classe `Cellule`
-  Liste chaînée (classe `ListeChaine`)
-  Illustration
-  Algorithmes récursifs de parcours
 -  deux familles de parcours
 -  parcours de gauche à droite
 -  parcours de droite à gauche
-  Algorithmes itératifs de parcours
 -  cas de la récursivité terminale
 -  cas de la récursivité non terminale
-  Introduction aux exceptions







TYPE ABSTRAIT DE DONNÉE LISTE

Notion de Type Abstrait de Données

 Un **TAD** est une collection d'éléments munie d'opérations sur ces éléments

-  il ne définit pas la façon dont les données sont stockées
 -  i.e. il est défini indépendamment de la manière dont on choisit de représenter les données en mémoire
-  il ne définit pas la façon dont les opérations sont implantées ; en effet, on ne sait pas comment les données sont stockées !

 Des **TAD** que vous connaissez sans le savoir

-  Collection séquentielle indexée (CSI)
 -  une collection d'éléments tous du même type numérotés sur un intervalle [borne_inf, borne_sup]
 -  implémentée par exemple avec un vecteur (on a choisi `ArrayList<E>`)
-  Collection séquentielle indexée triée
 -  une collection d'éléments comparables tous du même type numérotés en respectant leur ordre relatif sur un intervalle [borne_inf, borne_sup]
 -  implémentée par exemple avec un vecteur trié

Notion de Type Abstrait de Données

 Un **TAD** est défini par :

 Nom : un nom du type abstrait

 Utilise : les types abstraits utilisés par celui que l'on définit

 Opérations : prototype de toutes les opérations que l'on peut faire sur le type abstrait

 parmi ces opérations on trouve un constructeur, des transformateurs (mutateurs, setters) et des observateurs (accesseurs, getters)

 exemples :

 insérer un élément à la position n dans une coll. seq. indexée (CSI)

 consulter le $n^{\text{ième}}$ dans une CSI

 Les opérations sont dotées de :

 d'une pré-condition

 qui doit être vérifiée pour que l'opération produise le résultat attendu

 d'une post-condition (ou axiome)






 qui décrit le comportement, l'effet, de chaque opération

Le TAD liste
















 Une liste L est une suite finie d'éléments

$$l_1, l_2, \dots, l_n ; n \geq 0$$

 On appelle :

-  tête de L : premier élément l_1 de L
-  reste de L : la liste l_2, \dots, l_n {le reste est une liste}
-  longueur de L : nombre d'éléments de L
-  liste vide : liste sans élément (de longueur 0), ici notée null
-  successeur de l'élément l_i : l'élément l_{i+1} ($1 \leq i < n$)








Opérations sur le TAD liste

-  `estVide() : boolean;`
-  `getLongueur() : int;`
-  `getInfoAtPosit(posit) : info;`
 -  `// pas toujours possible`
-  `insertete(nouvelleInfo);`
 -  `// on pourrait s'en passer`
-  `supprimeTete();`
 -  `// on pourrait s'en passer`
-  `insereAtPosit(position, nouvelleInfo) : bool;`
 -  `// vrai si réussite`
-  `supprimeAtPosit(position) : bool;`
 -  `// vrai si réussite`
-  `setInfoAtPosit(position, nouvelleInfo) : bool;`
 -  `// vrai si réussite`
-  `vide();`

NOTE



-  pour des raisons pédagogiques certaines opérations seront implantées avec plusieurs algorithmes

Le TAD liste

-  Le **TAD** liste sera décrit (implanté) au moyen d'une interface générique qui ...
 -  définira uniquement les opérations (services)
-  La classe ListeChaine implantera une liste en implantant cette interface, et ...
 -  ... définira la structure de données utilisée pour représenter la liste
 -  nous utiliserons une liste dynamique d'objet (des Cellules)
 -  ... implantera les opérations décrites dans l'interface
 -  nous implanterons des méthodes sur une liste de Cellules

Une Interface ?






Contenu

-  une interface définit un ensemble de méthodes abstraites (services) et éventuellement des constantes de classe
-  les méthodes sont implicitement publiques (public) et abstraites (abstract)

Mise en place en Java

```
public interface Mon_Interface {  
    // déclaration de constantes  
    type nomDeConstante = valeur ;  
    // signature de méthodes publiques abstraites  
    type maMethode(type et nom des paramètres formels) ;  
}
```

Implantation

-  n'importe quelle classe peut implanter une interface ...
-  `public class Ma_Classe implements Une_Interface {...}`
-  ... en implantant toutes les méthodes que l'interface propose
-  une classe peut implanter plusieurs interfaces
-  **Comparable** est une interface qui définit le service `int compareTo()`

Le TAD liste : `ListeInterface`

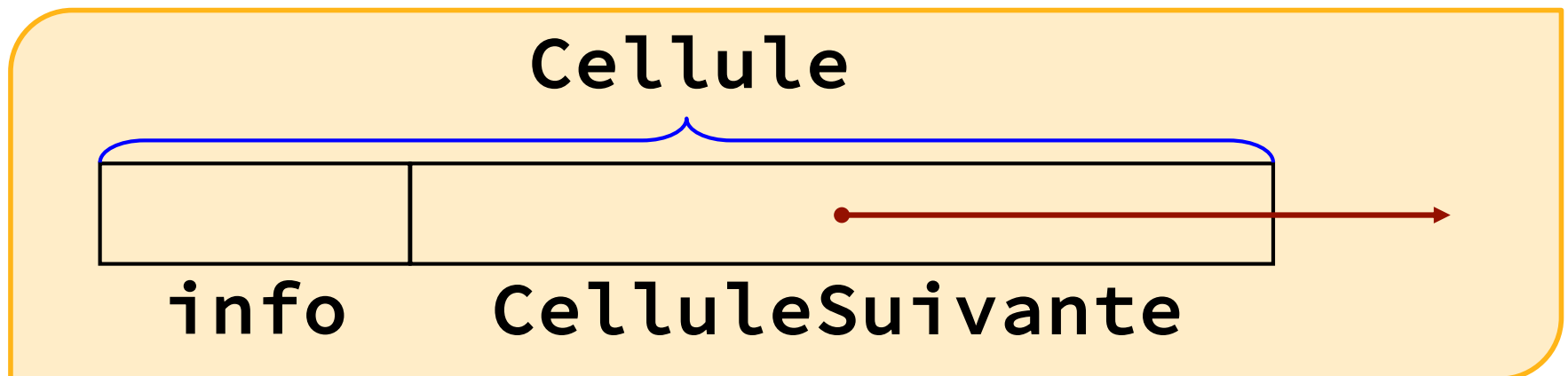
```
public interface ListeInterface<TypeInfo> {  
    // TypeInfo : type des informations que l'on peut mettre dans la liste  
    // exemple : Integer, Rectangle, Pays, Polar  
  
    // prototype des services minimum que doit rendre une liste  
    boolean estVide();  
  
    int getLongueur();  
  
    void insereTete(TypeInfo uneInfo);  
    void supprimeTete();  
  
    // les positions dans une liste sont numérotées à partir de 1  
    boolean insereAtPosit(int position, TypeInfo nouvelleInfo);  
    boolean supprimeAtPosit(int position);  
  
    TypeInfo getInfoAtPosit(int position) throws ExceptionMauvaisIndice;  
    boolean setInfoAtPosit(int position, TypeInfo nouvelleInfo);  
  
    void vide();  
}
```

CLASSE CELLULE

La classe Cellule

On va définir la classe Cellule pour représenter un élément d'une liste chaînée

Une Cellule à deux attributs :



On propose une classe modèle (template) pour manipuler des listes chaînées contenant des infos de n'importe quel type comparable

La classe Cellule

```
public class Cellule<TypeInfo> {  
    private TypeInfo info;           // information de cette Cellule  
    private Cellule<TypeInfo> celluleSuivante; // référence vers suivante  
  
    /** Constructeur de Cellule sans suivante */  
    public Cellule(TypeInfo info) {  
        this.info = info;  
        celluleSuivante = null;  
    }  
  
    /** Constructeur de Cellule avec celluleSuivante comme suivante */  
    public Cellule(TypeInfo info, Cellule<TypeInfo> celluleSuivante) {  
        this.info = info;  
        this.celluleSuivante = celluleSuivante;  
    }  
  
    /** getter et setter de info */  
    public TypeInfo getInfo() { return info; }  
    public void setInfo(TypeInfo info) { this.info = info; }  
  
    /** getter et setter de info */  
    public Cellule<TypeInfo> getCelluleSuivante() { return celluleSuivante; }  
    public void setCelluleSuivante(Cellule<TypeInfo> celluleSuivante) {  
        this.celluleSuivante = celluleSuivante;  
    }  
}
```

Illustration



Soit le programme Java suivant :

```
01 public class testCellule {
02     public static void main(String[] args) {
03         // construction de cell1, avec l'information 5, sans successeur
04         Cellule<Integer> cell1 = new Cellule<>(5);
05         System.out.println("cell 1 à l'adresse : " + cell1 + " ; info : "
06                             + cell1.getInfo() + ", suivante à l'adresse : " + cell1.getCelluleSuivante());
07         // construction de cell2, avec l'information 8, sans successeur
08         Cellule<Integer> cell2 = new Cellule<>(8);
09         System.out.println("cell 2 à l'adresse : " + cell2 + " ; info : "
10                             + cell2.getInfo() + ", suivante à l'adresse : " + cell2.getCelluleSuivante());
11         // mise à jour suivante de cell1 avec cell2 (son adresse)
12         cell1.setCelluleSuivante(cell2);
13         System.out.println("cell 1 mise à jour à l'adresse : " + cell1 + " ; info : "
14                             + cell1.getInfo() + ", suivante à l'adresse : " + cell1.getCelluleSuivante());
15         // construction cell3 avec cell1 comme suivante cell3
16         Cellule<Integer> cell3 = new Cellule<>(3, cell1 );
17         System.out.println("cell 3 à l'adresse : " + cell3 + " ; info : "
18                             + cell3.getInfo() + ", suivante à l'adresse : " + cell3.getCelluleSuivante());
19     }
20 }
```

Illustration

 Trace obtenue :

```
05 // cell1 construite sans suivante (null) contient 5
    cell1 à l'adresse : Cellule@eed1f14 ; info : 5, suivante à l'adresse : null

08 // cell2 construite sans suivante (null) contient 8
    cell2 à l'adresse : Cellule@7229724f ; info : 8, suivante à l'adresse : null

    // mise à jour de la suivante de cell1
    // cell1 contient toujours 5 et a maintenant pour suivante cell2 à l'adresse @7229724f
11 cell1 mise à jour à l'adresse : Cellule@eed1f14 ; info : 5, suivante à l'adresse : Cellule@7229724f

    // cell3 construite avec comme suivante cell1 à l'adresse @eed1f14 contient 3
14 cell3 à l'adresse : Cellule@4c873330 ; info : 3, suivante à l'adresse : Cellule@eed1f14
```

```

public class testCellule {
    public static void main(String[] args) {
        // construction de cell1, avec l'information 5, sans successeur
        Cellule<Integer> cell1 = new Cellule<>(5);
        System.out.println("cell 1 à l'adresse : " + cell1 + " ; info : "
            + cell1.getInfo() + ", suivante à l'adresse : " + cell1.getCelluleSuivante());
        // construction de cell2, avec l'information 8, sans successeur
        Cellule<Integer> cell2 = new Cellule<>(8);
        System.out.println("cell 2 à l'adresse : " + cell2 + " ; info : "
            + cell2.getInfo() + ", suivante à l'adresse : " + cell2.getCelluleSuivante());
        // mise à jour suivante de cell1 avec cell2 (son adresse)
        cell1.setCelluleSuivante(cell2);
        System.out.println("cell 1 mise à jour à l'adresse : " + cell1 + " ; info : "
            + cell1.getInfo() + ", suivante à l'adresse : " + cell1.getCelluleSuivante());
        // construction cell3 avec cell1 comme suivante cell3
        Cellule<Integer> cell3 = new Cellule<>(3, cell1 );
        System.out.println("cell 3 à l'adresse : " + cell3 + " ; info : "
            + cell3.getInfo() + ", suivante à l'adresse : " + cell3.getCelluleSuivante());
    }
}

```

variable	adresse	contenu mémoire	
----------	---------	-----------------	--

cell1	@eed1f14	5	null
-------	----------	---	------


```

public class testCellule {
    public static void main(String[] args) {
        // construction de cell1, avec l'information 5, sans successeur
        Cellule<Integer> cell1 = new Cellule<>(5);
        System.out.println("cell 1 à l'adresse : " + cell1 + " ; info : "
            + cell1.getInfo() + ", suivante à l'adresse : " + cell1.getCelluleSuivante());
        // construction de cell2, avec l'information 8, sans successeur
        Cellule<Integer> cell2 = new Cellule<>(8);
        System.out.println("cell 2 à l'adresse : " + cell2 + " ; info : "
            + cell2.getInfo() + ", suivante à l'adresse : " + cell2.getCelluleSuivante());
        // mise à jour suivante de cell1 avec cell2 (son adresse)
        cell1.setCelluleSuivante(cell2);
        System.out.println("cell 1 mise à jour à l'adresse : " + cell1 + " ; info : "
            + cell1.getInfo() + ", suivante à l'adresse : " + cell1.getCelluleSuivante());
        // construction cell3 avec cell1 comme suivante cell3
        Cellule<Integer> cell3 = new Cellule<>(3, cell1 );
        System.out.println("cell 3 à l'adresse : " + cell3 + " ; info : "
            + cell3.getInfo() + ", suivante à l'adresse : " + cell3.getCelluleSuivante());
    }
}

```

variable	adresse	contenu mémoire	
----------	---------	-----------------	--

cell1	@eed1f14	5	null
-------	----------	---	------

cell2	@7229724f	8	null
-------	-----------	---	------

```

public class testCellule {
    public static void main(String[] args) {
        // construction de cell1, avec l'information 5, sans successeur
        Cellule<Integer> cell1 = new Cellule<>(5);
        System.out.println("cell 1 à l'adresse : " + cell1 + " ; info : "
            + cell1.getInfo() + ", suivante à l'adresse : " + cell1.getCelluleSuivante());
        // construction de cell2, avec l'information 8, sans successeur
        Cellule<Integer> cell2 = new Cellule<>(8);
        System.out.println("cell 2 à l'adresse : " + cell2 + " ; info : "
            + cell2.getInfo() + ", suivante à l'adresse : " + cell2.getCelluleSuivante());
        // mise à jour suivante de cell1 avec cell2 (son adresse)
        cell1.setCelluleSuivante(cell2);
        System.out.println("cell 1 mise à jour à l'adresse : " + cell1 + " ; info : "
            + cell1.getInfo() + ", suivante à l'adresse : " + cell1.getCelluleSuivante());
        // construction cell3 avec cell1 comme suivante cell3
        Cellule<Integer> cell3 = new Cellule<>(3, cell1 );
        System.out.println("cell 3 à l'adresse : " + cell3 + " ; info : "
            + cell3.getInfo() + ", suivante à l'adresse : " + cell3.getCelluleSuivante());
    }
}

```

variable	adresse	contenu mémoire	
----------	---------	-----------------	--

cell1	@eed1f14	5	@7229724f
-------	----------	---	-----------

cell2	@7229724f	8	null
-------	-----------	---	------

```

public class testCellule {
    public static void main(String[] args) {
        // construction de cell1, avec l'information 5, sans successeur
        Cellule<Integer> cell1 = new Cellule<>(5);
        System.out.println("cell 1 à l'adresse : " + cell1 + " ; info : "
            + cell1.getInfo() + ", suivante à l'adresse : " + cell1.getCelluleSuivante());
        // construction de cell2, avec l'information 8, sans successeur
        Cellule<Integer> cell2 = new Cellule<>(8);
        System.out.println("cell 2 à l'adresse : " + cell2 + " ; info : "
            + cell2.getInfo() + ", suivante à l'adresse : " + cell2.getCelluleSuivante());
        // mise à jour suivante de cell1 avec cell2 (son adresse)
        cell1.setCelluleSuivante(cell2);
        System.out.println("cell 1 mise à jour à l'adresse : " + cell1 + " ; info : "
            + cell1.getInfo() + ", suivante à l'adresse : " + cell1.getCelluleSuivante());
        // construction cell3 avec cell1 comme suivante cell3
        Cellule<Integer> cell3 = new Cellule<>(3, cell1 );
        System.out.println("cell 3 à l'adresse : " + cell3 + " ; info : "
            + cell3.getInfo() + ", suivante à l'adresse : " + cell3.getCelluleSuivante());
    }
}

```

variable	adresse	contenu mémoire	
----------	---------	-----------------	--

cell1	@eed1f14	5	@7229724f
-------	----------	---	-----------

cell2	@7229724f	8	null
-------	-----------	---	------

cell3	@4c873330	3	@eed1f14
-------	-----------	---	----------

Où je comprends la définition inductive d'un type de données !

CLASSE LISTECHAINEE

La classe ListeChaine

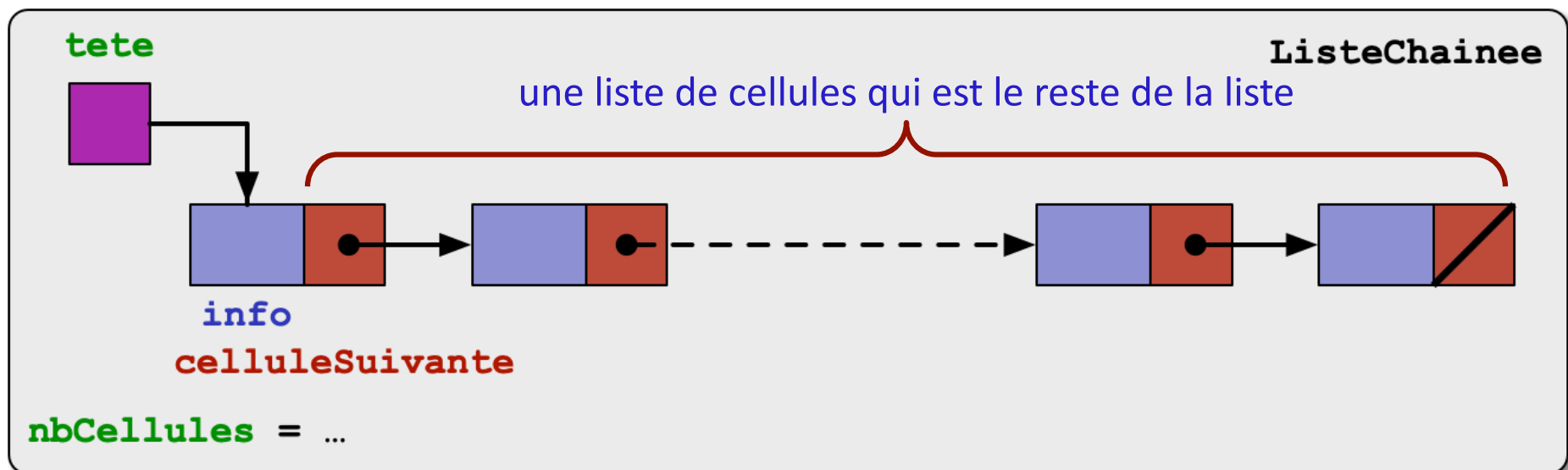
■ Implantation du TAD liste au moyen d'une liste chaînée de **Cellules**

■ **tete** est un pointeur sur la première Cellule

■ chaque Cellule pointe sur son successeur

■ la dernière Cellule n'a pas de successeur (**null**)

■ **nbCellules** est le nombre de Cellules de la liste



```
public class ListeChaine<TypeInfo extends Comparable<TypeInfo>> implements
ListeInterface<TypeInfo> {

    private Cellule<TypeInfo> tete;
    private int nbCellules;

    /**
     * Getter de la tête présent uniquement pour des raisons pédagogiques
     * utilisé seulement dans la classe Utilitaire
     */
    public Cellule<TypeInfo> getTete() {
        return tete;
    }

    /**
     * Construction d'une liste vide
     */
    public ListeChaine() {
        tete = null;
        nbCellules = 0;
    }

    @Override
    public boolean estVide() {
        return nbCellules == 0;
    }

    @Override
    public void vide() {
        tete = null;
        nbCellules = 0;
    }
}
```

La classe ListeChaine

```
public class ListeChaine<TypeInfo extends Comparable<TypeInfo>>
    implements ListeInterface<TypeInfo> {
    /*
    * Pour utiliser l'ordre naturel de la classe TypeInfo dans les méthodes de la classe
    * (ici ListeChaine), il faut préciser TypeInfo extends Comparable<TypeInfo>
    *
    * Les méthodes @Override sont les méthodes de l'interface ListeInterface
    * La classe ListeInterface doit obligatoirement implanter tous les services de
    * ListeInterface
    * Certaines méthodes ont un worker récursif
    */

    private Cellule<TypeInfo> tete;           // pointeur sur la première Cellule
    private int nbCellules;                  // nombre de Cellules

    public ListeChaine() {...}               // construction d'une liste vide avec 0 Cellules

    @Override
    public boolean estVide() {...}

    @Override
    public void vide() {...}

    @Override
    public int getLongueur() {...}
```

La classe ListeChaine

```
private Cellule<TypeInfo> insereTeteWorker(Cellule<TypeInfo> celluleCourante,  
                                           TypeInfo uneInfo) {...}  
  
@Override  
public void insereTete(TypeInfo uneInfo) {...}
```

```
private Cellule<TypeInfo> supprimeTeteWorker(Cellule<TypeInfo> celluleCourante) {...}  
  
@Override  
public void supprimeTete() {...}
```

```
private Cellule<TypeInfo> insereAtPositRecWorker(Cellule<TypeInfo> celluleCourante,  
                                                  int position,  
                                                  TypeInfo nouvelleInfo) {...}  
  
@Override  
public boolean insereAtPosit(int position, TypeInfo nouvelleInfo) { ...}
```

```
private Cellule<TypeInfo> supprimeAtPositRecWorker(Cellule<TypeInfo> celluleCourante,  
                                                    int position) {...}  
  
@Override  
public boolean supprimeAtPosit(int position) {...}
```

```
private TypeInfo getInfoAtPositRecWorker(Cellule<TypeInfo> celluleCourante,  
                                          int position) {...}  
  
@Override  
public TypeInfo getInfoAtPosit(int position) throws ExceptionMauvaisIndice {...}
```


La classe ListeChaine

```
private boolean setInfoAtPositRecWorker(Cellule<TypeInfo> celluleCourante,  
                                         int position,  
                                         TypeInfo nouvelleInfo) {... }  
  
@Override  
public boolean setInfoAtPosit(int position, TypeInfo nouvelleInfo) {...}
```

```
private int compterNbOccValWorker(Cellule<TypeInfo> celluleCourante, TypeInfo val) {...}  
public int compterNbOccVal(TypeInfo val) {...}
```

```
private void afficheGaucheDroiteRecWorker(Cellule<TypeInfo> celluleCourante) {...}  
public void afficheGaucheDroiteRec() {...}
```

```
public void afficheGraucheDroiteIter() {...}
```

```
private void afficheDroiteGaucheRecWorker(Cellule<TypeInfo> celluleCourante) {...}  
public void afficheDroiteGaucheRec() {...}
```

```
}
```

Où j'ai une idée de comment ça marche !

ILLUSTRATION

```

01 public class testListeChaineCours {
02     public static void main(String[] args) {
03         ListeChaine<String> maListe = new ListeChaine<>();
04         maListe.insereAtPosit(1, "un");
05         maListe.insereAtPosit(2, "deux");
06         maListe.insereAtPosit(2, "entreUnEtDeux");
07         maListe.afficheGaucheDroiteRec();
08         maListe.supprimeTete();
09         maListe.afficheGaucheDroiteRec();
10         maListe.insereTete("nouveauUn");
11         maListe.afficheGaucheDroiteRec();
12         try {
13             System.out.println("Élément en position 2 : " + maListe.getInfoAtPosit(2));
14             System.out.println("Élément en position 12 : " + maListe.getInfoAtPosit(12));
15         } catch (ExceptionMauvaisIndice e) {
16             System.out.print(e.getMessage());
17             System.out.println("Exception interceptée, poursuite de l'exécution du programme !");
18         }
19         System.out.println("\nL'exécution est terminée !");
20     }
21 }

```

```

07 affichage récursif de gauche à droite : un -> entreUnEtDeux -> deux ->
09 affichage récursif de gauche à droite : entreUnEtDeux -> deux ->
11 affichage récursif de gauche à droite : nouveauUn -> entreUnEtDeux -> deux ->
13 Élément en position 2 : entreUnEtDeux
14 // la position n'est pas légale, une exception est levée à l'exécution de maListe.getInfoAtPosit(12)
16 Indice de valeur 12 illégal dans getInfoAtPosit, la position doit être dans l'intervalle [1, 3]
17 Exception interceptée, poursuite de l'exécution du programme !
19 L'exécution est terminée !

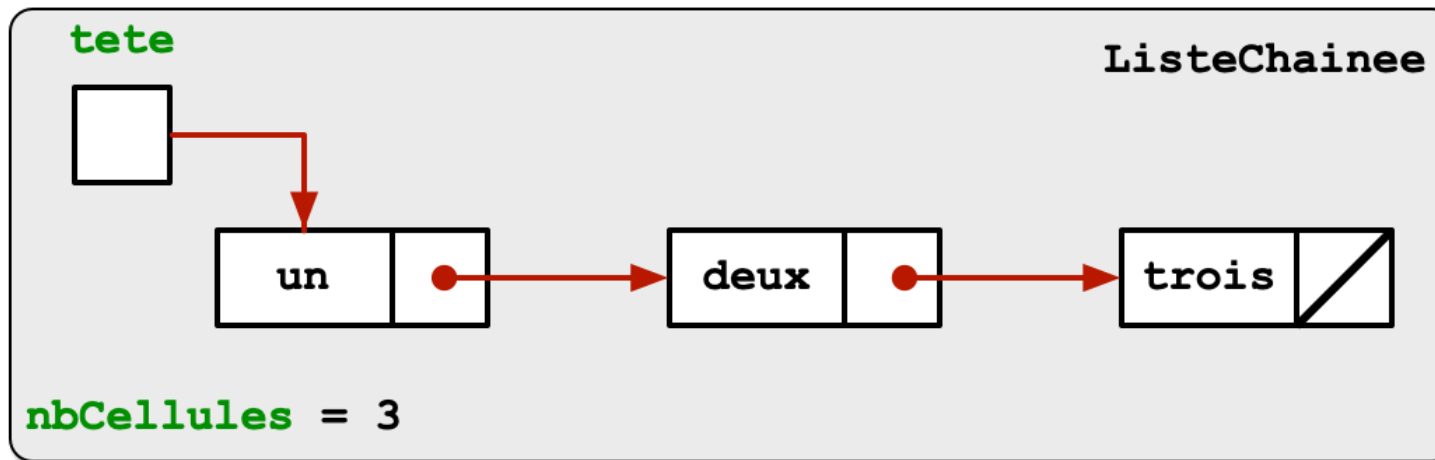
```

Où je profite complètement de la définition inductive !

ALGORITHMES RÉCURSIFS

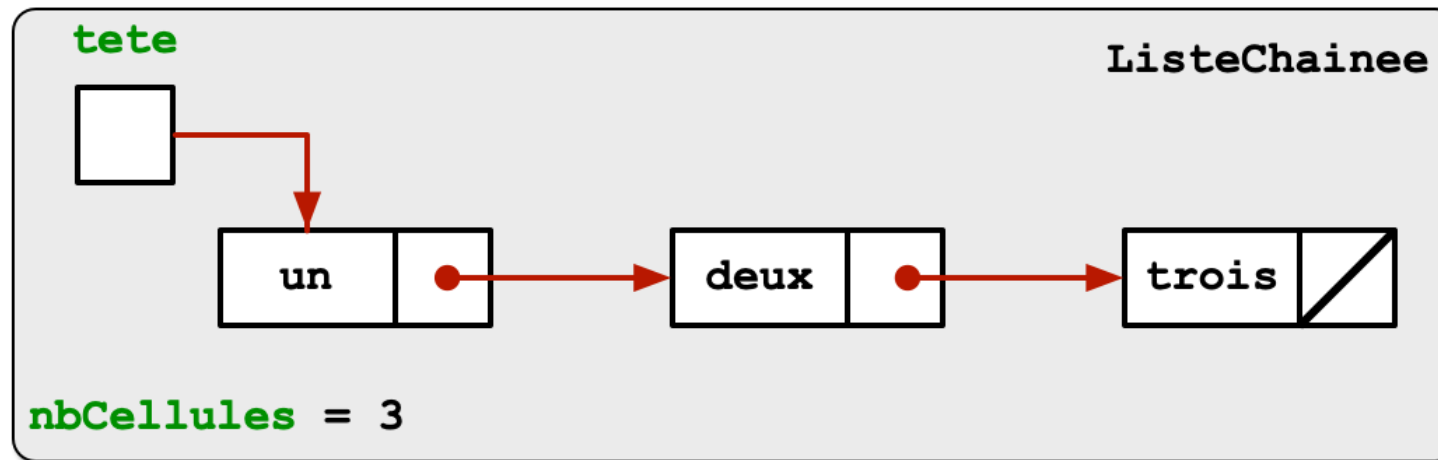
Affichage d'une ListeChaine

- Une **ListeChaine** comporte un pointeur sur sa première **Cellule** ...
- ... il faut donc parcourir toutes les **Cellules** de la **ListeChaine** à partir de la première
 - soit **tete** de type **Cellule<TypeInfo>**
- ... en affichant l'information portée par chacune



Affichage d'une ListeChaine

🧩 Pour la ListeChaine suivante :



🧩 On pourrait produire

🧩 un -> deux -> trois ->

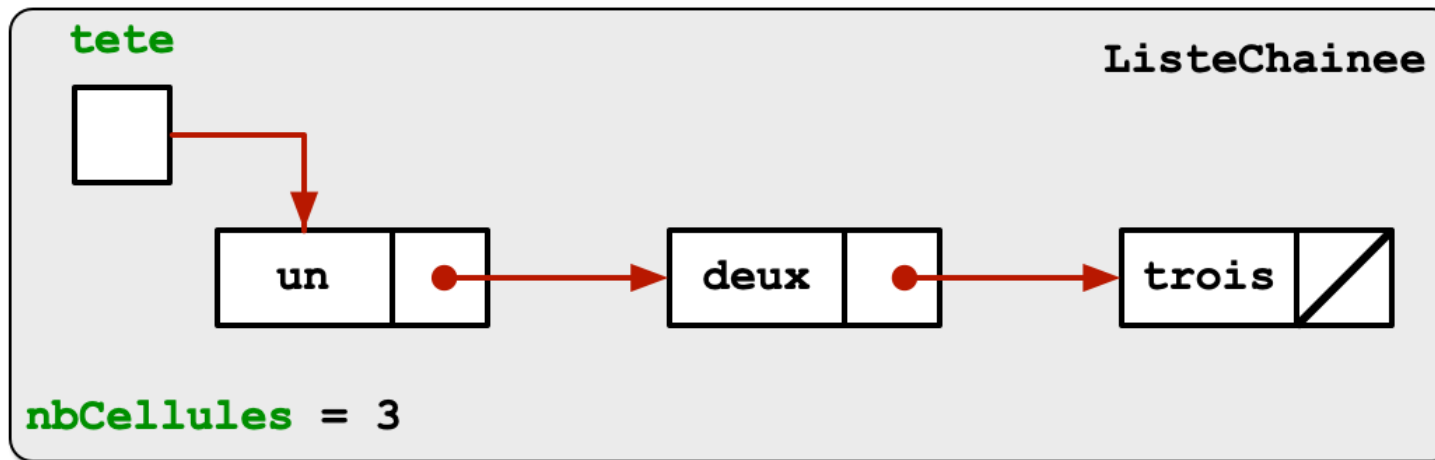
🧩 parcours de gauche à droite

🧩 <- trois <- deux <- un

🧩 parcours de droite à gauche

Affichons une liste de Cellules

On souhaite afficher :



Imaginons que l'on dispose de 2 procédures récursives

```
void afficheGaucheDroiteRecWorker(Cellule<TypeInfo> celluleCourante)
// affiche l'information des Cellules atteignables depuis
// celluleCourante en faisant un parcours de gauche à droite
```


```
void afficheGaucheDroiteRecWorker(Cellule<TypeInfo> celluleCourante)
// affiche le l'information des Cellules atteignables depuis
// celluleCourante en faisant un parcours de droite à gauche
```

Affichage d'une liste de Cellules

 `celluleCourante`, une tête de liste de `Cellules`

 soit la liste ne contient aucune `Cellule`

 `celluleCourante == null`

 soit la liste est composée d'une première `Cellule` chaînée à une liste de `Cellules` (celles qui suivent la première)

 `celluleCourante != null`

 `celluleCourante.getInfo()`

 est l'information portée par la première `Cellule`

 `celluleCourante.getSuivante()`

 est un pointeur sur la première `Cellule` du reste de la liste

Affichage de gauche à droite

Affichage d'une liste de Cellules

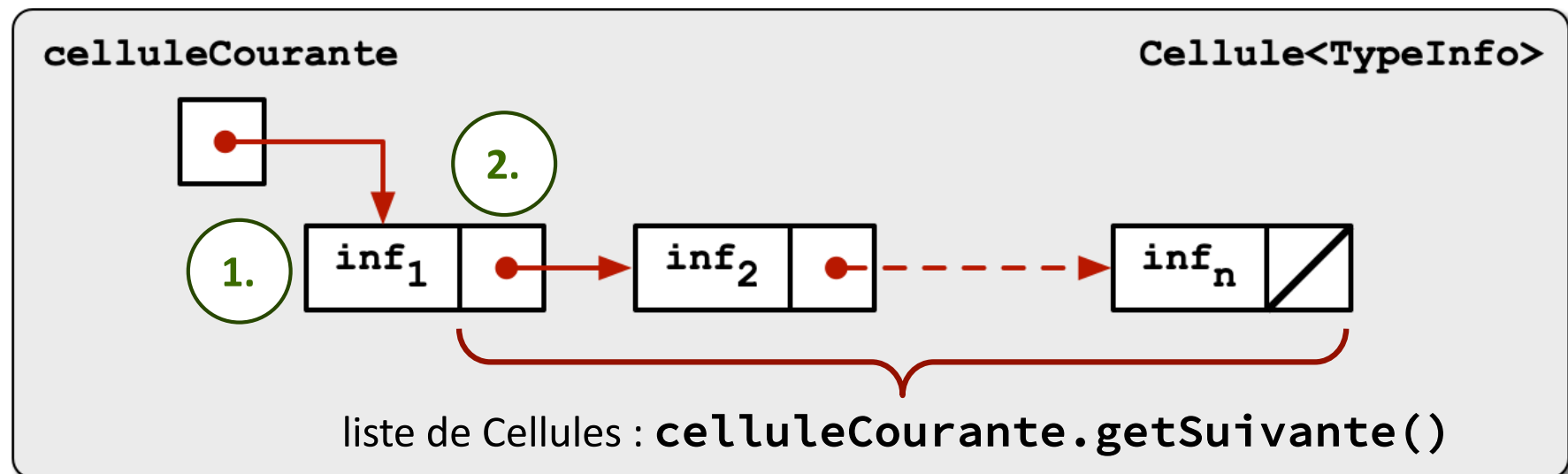
gauche-droite
récursif

■ Si la liste de Cellules est vide :

■ rien à faire !


■ Si la liste de Cellules n'est pas vide :

1. d'abord afficher le contenu de la première Cellule
2. puis afficher la liste de Cellules qui suivent la première



Affichage d'une liste de Cellules

gauche-droite
récuratif

 Réfléchissons à l'écriture de
`afficheGaucheDroite(Cellule<TypeInfo> celluleCourante)`
en supposant qu'elle existe

 Si la liste de `cellules` est vide

 `celluleCourante = null` → il n'y a rien à faire *

 Si la liste de `cellules` n'est pas vide

 `celluleCourante ≠ nullptr` →

 afficher l'info de la 1^{ière} Cellule : `celluleCourante.getInfo()`

 afficher le reste de la liste `celluleCourante.getSuivante()`

 comment ??

 en utilisant `afficheGaucheDroite(...)` faite pour ça !!!

Affichage d'une liste de Cellules

gauche-droite
récuratif

En résumé

```
celluleCourante == null → ne rien faire c'est terminé *  
celluleCourante != null →  
    print(celluleCourante.getInfo() + " -> ");  
    afficheGaucheDroiteRecWorker(celluleCourante.getSuivante());
```

Procédure

procédure réursive

```
void afficheGaucheDroiteRecWorker(Cellule<TypeInfo> celluleCourante) {  
  
    if (celluleCourante != null) {  
        System.out.print(celluleCourante.getInfo() + " -> ");  
  
        afficheGaucheDroiteRecWorker(celluleCourante.getCelluleSuivante());  
    }  
    // si la liste de Cellules est vide, ne rien faire  
}
```

Finalemment ...

gauche-droite
récuratif

 La procédure worker sera privée

```
private void afficheGaucheDroiteRecWorker(Cellule<TypeInfo> celluleCourante) {  
    if (celluleCourante != null) {  
        System.out.print(celluleCourante.getInfo() + " -> ");  
        afficheGaucheDroiteRecWorker(celluleCourante.getCelluleSuivante());  
    }  
    // si la liste de Cellules est vide, ne rien faire  
}
```

 La procédure d'affichage sera publique (sans paramètre)

```
public void afficheGaucheDroiteRec() {  
    System.out.print("affichage récuratif de gauche à droite : ");  
    afficheGaucheDroiteRecWorker(tete);  
    System.out.println();  
}
```

Affichage de droite à gauche

Affichage d'une liste de Cellules

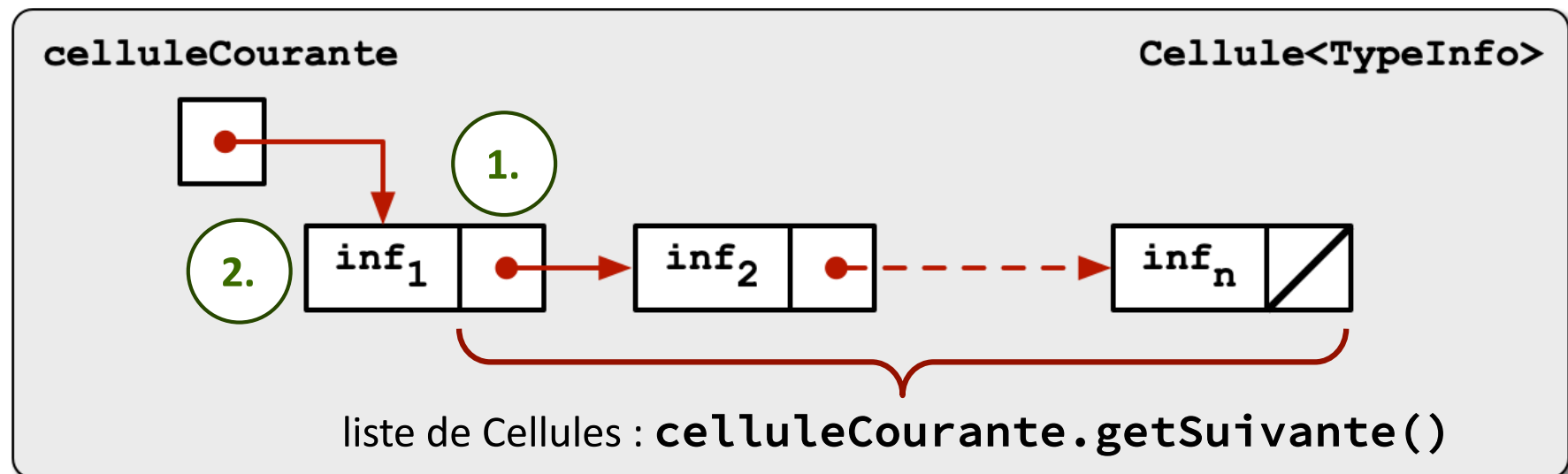
droite-gauche
récursif

■ Si la liste de Cellules est vide :

■ rien à faire !


■ Si la liste de Cellules n'est pas vide :

1. afficher la liste de Cellules qui suivent la première
2. puis afficher le contenu de la première Cellule



Affichage d'une liste de Cellules

droite-gauche
récursif

 Réfléchissons à l'écriture de
`afficheGaucheDroite(Cellule<TypeInfo> celluleCourante)`
en supposant qu'elle existe

 Si la liste de `Cellules` est vide

 `celluleCourante = null` → il n'y a rien à faire *

 Si la liste de `Cellules` n'est pas vide

 `celluleCourante ≠ nullptr` →

 afficher le reste de la liste `celluleCourante.getSuivante()`

 comment ??

 en utilisant `afficheGaucheDroite(...)` faite pour ça !!!

 afficher l'info de la 1^{ière} Cellule : `celluleCourante.getInfo()`

Affichage d'une liste de Cellules

droite-gauche
récursif

En résumé

```
celluleCourante == null → ne rien faire c'est terminé *  
celluleCourante != null →  
    afficheCellulesDroiteGauche(celluleCourante.getSuivante());  
    print(" <- " + celluleCourante.getInfo());
```

Procédure

procédure récursive

```
void afficheDroiteGaucheRecWorker(Cellule<TypeInfo> celluleCourante) {  
    if (celluleCourante != null) {  
        afficheDroiteGaucheRecWorker(celluleCourante.getCelluleSuivante());  
        System.out.print(celluleCourante.getInfo() + " -> ");  
    }  
    // si la liste de Cellules est vide, ne rien faire  
}
```

Finalemment ...

droite-gauche
récuratif

 La procédure worker sera privée

```
private void afficheDroiteGaucheRecWorker(Cellule<TypeInfo> celluleCourante) {  
    if (celluleCourante != null) {  
        afficheDroiteGaucheRecWorker(celluleCourante.getCelluleSuivante());  
        System.out.print(celluleCourante.getInfo() + " -> ");  
    }  
    // si la liste de Cellules est vide, ne rien faire  
}
```

 La procédure d'affichage sera publique (sans paramètre)

```
public void afficheDroiteGaucheRec() {  
    System.out.print("affichage récuratif de gauche à droite : ");  
    afficheDroiteGaucheRecWorker(tete);  
    System.out.println();  
}
```

Où je reviens à ce que je connais, l'itération !

ALGORITHMES ITÉRATIFS

Affichage d'une ListeChaine ...

gauche-droite
Itératif

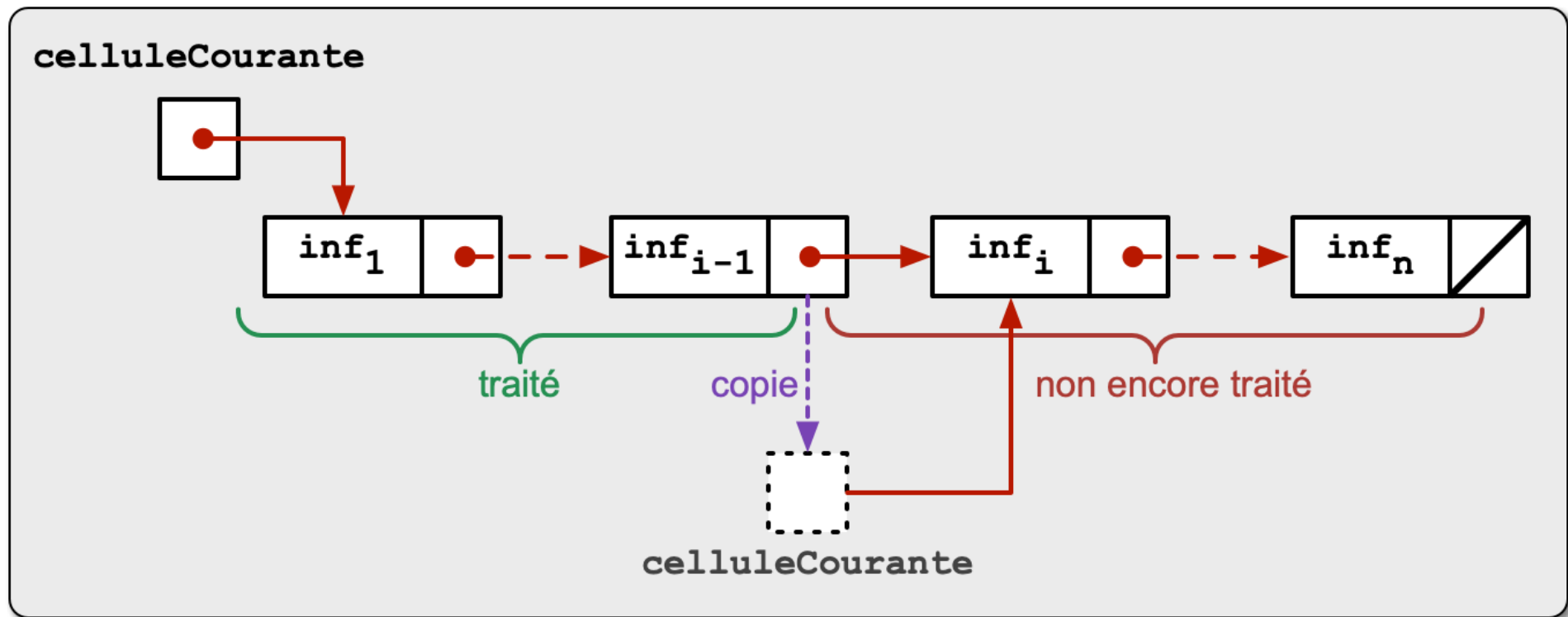
 Une méthode publique de **ListeChaine**

```
public void afficheGraucheDroiteIter() {  
  
    ...  
  
}
```

Affichage d'une liste de Cellules

gauche-droite
Itératif




- Même approche que pour un vecteur
- on a affiché les Cellules qui précèdent celluleCourante
- on n'a pas encore traité les Cellules qui suivent

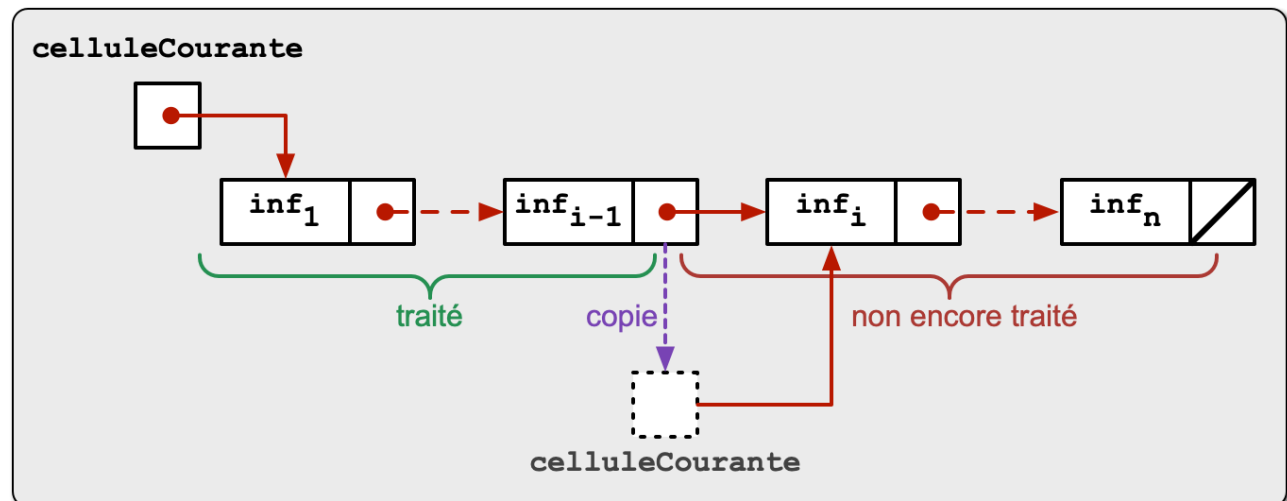


Affichage d'une liste de Cellules

gauche-droite
Itératif

Raisonnement par récurrence

-  Invariant: `celluleCourante+` non traité
- `celluleCourante == null` → {terminé} *
- `celluleCourante != nullptr` →
 `print(celluleCourante.getInfo() + " ");`
 `celluleCourante = celluleCourante.getSuivante();` ➔ I
-  Itération: **while** (`celluleCourante != null`) {...}
-  Initialisation: `celluleCourante = tete;` ➔ I



Affichage d'une liste de Cellules




gauche-droite
Itératif

 Code Java

```
public void afficheGaucheDroiteIter() {  
  
    Cellule<TypeInfo> celluleCourante = tete;  
  
    while (celluleCourante != null) {  
        System.out.print(" <- " + celluleCourante.getInfo());  
        celluleCourante = celluleCourante.getCelluleSuivante();  
    }  
}
```

Affichage d'une liste de Cellules

gauche-droite
Itératif vs Récursif

-  L'écriture de la version itérative de l'affichage de gauche à droite ne présente pas de difficulté majeure
-  parce que dans la version récursive, on ne fait rien (aucune instruction) après l'appel récursif
-  on dit que c'est un appel récursif terminal

```
private void afficheGaucheDroiteRecWorker(Cellule<TypeInfo> celluleCourante) {  
    if (celluleCourante != null) {  
        System.out.print(celluleCourante.getInfo() + " -> ");  
        afficheGaucheDroiteRecWorker(celluleCourante.getCelluleSuivante());  
    }  
    // si la liste de Cellules est vide, ne rien faire  
}
```


Affichage d'une liste de Cellules

droite-gauche
Itératif vs Récursif

- 🧩 L'écriture de la version itérative de l'affichage de droite à gauche est plus difficile
 - 🧩 parce que dans la version récursive, on travaille encore après l'appel récursif (print...)
 - 🧩 dans la version itérative, il faut se rappeler d'où l'on vient
 - 🧩 il faut pour cela une structure de données auxiliaire (pile)
 - 🧩 on dit que c'est un appel récursif non terminal

```
private void afficheDroiteGaucheRecWorker(Cellule<TypeInfo> celluleCourante) {  
    if (celluleCourante != null) {  
        afficheDroiteGaucheRecWorker(celluleCourante.getCelluleSuivante());  
        System.out.print(celluleCourante.getInfo() + " -> ");  
    }  
    // si la liste de Cellules est vide, ne rien faire  
}
```

INTRODUCTION AUX EXCEPTIONS

Introduction

 Un programme peut rencontrer une erreur ou un événement anormal

 Erreur technique

 Fichier non présent, plus de mémoire, etc.

 Erreur métier



 Arguments invalides, etc.

 Prévoir un traitement d'erreur sur les instructions susceptibles de les provoquer




 En Java, ce traitement est intégré dans le langage : traitement des exceptions

Exception : principe

 Une exception est créée :

-  soit par la JVM [environnement d'exécution] (erreur interne/technique)
-  soit par une levée d'exception du programmeur

 Deux solutions

-  attraper (intercepter) l'exception immédiatement pour la traiter
-  relancer (propager) l'exception à la méthode qui a déclenché le traitement erroné
 -  MAIS on devra attraper et traiter au cours de la propagation

Attraper une exception : syntaxe de base

```
try {  
    ... // instructions à contrôler  
  
} catch (MonException e) {  
    ... // traitement de l'exception e  
  
} [catch (AutreException e) {...}]
```

try-catch : exemple

Code	<pre>1 public class TestException { 2 public static void main(String args[]) { 3 int num1, num2; 4 try { // bloc try pour circonscrire l'interception 5 num1 = 0; 6 num2 = 62 / num1; // division par zéro 7 System.out.println("On va quitter un bloc try sans problème"); 8 } catch (ArithmeticException e) { 9 // bloc pour intercepter une division par zéro 10 System.out.println("Erreur : il est interdit de diviser par 0!"); 11 } 12 System.out.println("Sortie du bloc try-catch"); 13 System.out.println("pour une suite d'exécution normale"); 14 } 15 }</pre>
Trace	<pre>6 // levée d'une exception 8 // interception de l'exception 10 Erreur : il est interdit de diviser par 0 ! 12 Sortie du bloc try-catch 14 pour une suite d'exécution normale</pre>

Lever une exception : syntaxe

 `throw new Exception(...);`

 Exception est une classe définie par Java

 `throw new RuntimeException(...);`

 exceptions définies par Java

 `IOException`, `ParseException`, etc.

 exceptions définies dans le code (par le programmeur)

 héritant de la classe `Exception`

(re)lancer une exception : syntaxe

- Ajouter dans la signature de la méthode qui relance le mot-clef `throws` suivi du type d'exception

```
public static int division(int c, int d)
    throws ArithmeticException {
    ...
}
```

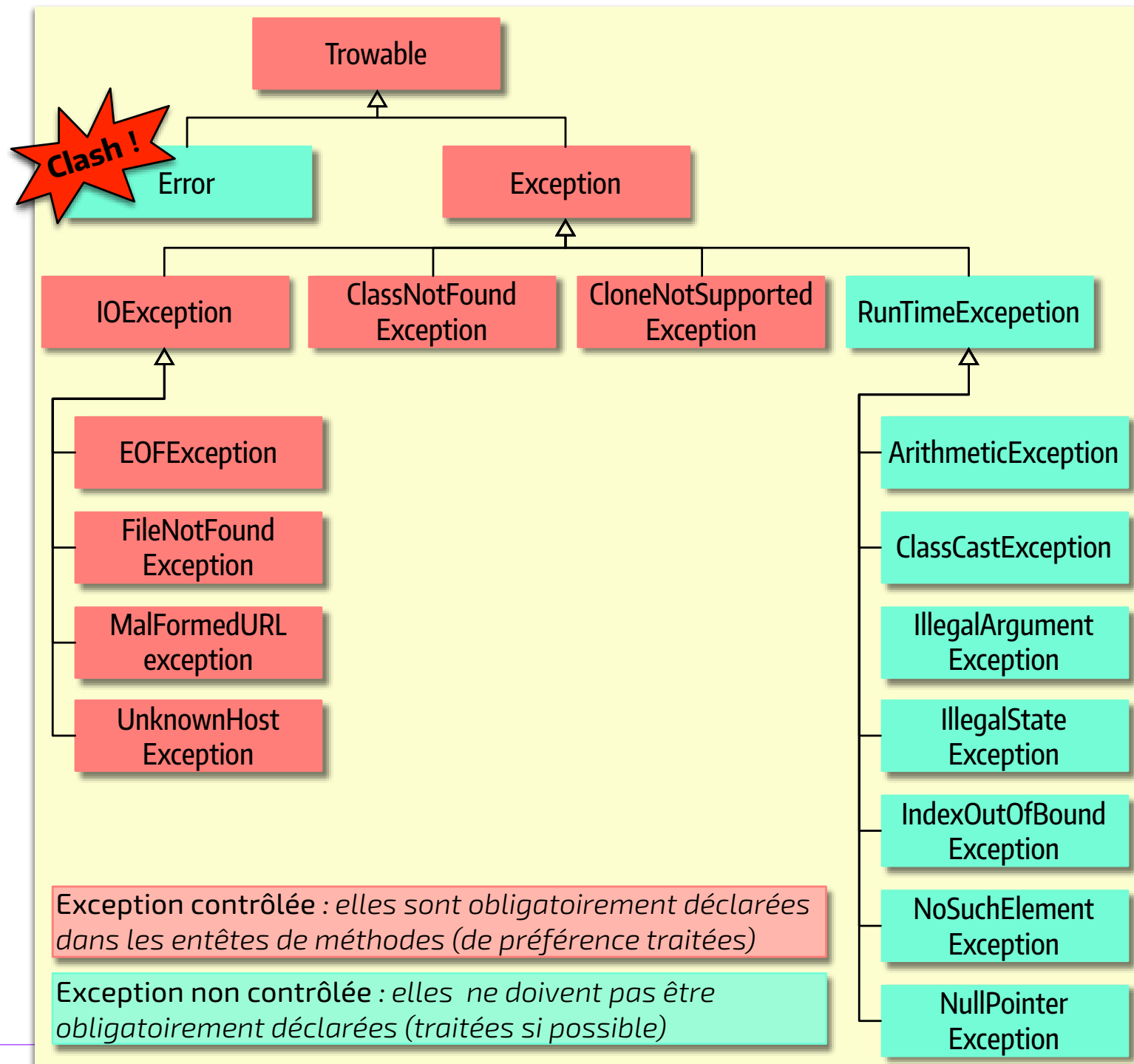
- ATTENTION : il faut normalement attraper l'exception et la traiter à un moment de la « remontée »

Code	<pre>12 public class TestThrow1 { 13 14 static void valider(int age) throws ArithmeticException { 15 if (age < 18) { 16 throw new ArithmeticException("âge invalide"); 17 } else { 18 System.out.println("vous pouvez voter !"); 19 } 20 } 21 22 public static void main(String args[]) { 23 System.out.println("début du code..."); 24 valider(13); 25 System.out.println("suite du code..."); 26 } 27 } 28</pre>
Trace	<pre>23 début du code... Exception in thread "main" java.lang.ArithmeticException: âge invalide at ExceptionPlayGround.TestThrow1.valider(TestThrow1.java:16) at ExceptionPlayGround.TestThrow1.main(TestThrow1.java:24) Java Result: 1</pre>

Lancer une exception : Exception

Code	<pre>12 public class TestThrow2 { 13 14 static void valider(int age) throws Exception { 15 if (age < 18) { 16 throw new Exception("âge invalide"); 17 } else { 18 System.out.println("vous pouvez voter !"); 19 } 20 } 21 22 public static void main(String args[]) throws Exception { 23 // Exception doit obligatoirement être relancée 24 System.out.println("début du code..."); valider(13); 25 System.out.println("suite du code..."); 26 } 27 } 28 29</pre>
Trace	<pre>24 début du code... Exception in thread "main" java.lang.Exception: âge invalide at ExceptionPlayGround.TestThrow1.valider(TestThrow1.java:16) at ExceptionPlayGround.TestThrow1.main(TestThrow1.java:24) Java Result: 1</pre>

Hiérarchie des exceptions (extrait)



Définition d'une exception

 Exemple de la classe `ExceptionMauvaisIndice`

 la syntaxe sera expliquée dans la ressource R2.01, pour l'instant il suffit de respecter ce modèle

```
public class ExceptionMauvaisIndice extends Exception {  
    // il faut utiliser extends Exception et deux constructeurs  
  
    // un constructeur par défaut comme suit  
    public ExceptionMauvaisIndice() {  
        super();  
    }  
  
    // un constructeur avec un paramètre s de type String  
    // s : message d'erreur associé à l'exception (.getMessage())  
    public ExceptionMauvaisIndice(String s) {  
        super(s);  
    }  
}
```

```
// extrait de la classe ListeChaine
@Override
public TypeInfo getInfoAtPosit(int position) throws ExceptionMauvaisIndice {
    if (position >= 1 & position <= nbCellules) { // position légale, appel du worker
        return getInfoAtPositRecWorker(tete, position);
    } else { // position illégale, levée d'exception
        // on utilise le constructeur avec paramètre pour construire un message d'erreur parlant
        throw new ExceptionMauvaisIndice("Indice de valeur " + position + " illégal dans
            getInfoAtPosit, la position doit être dans l'intervalle [1" + ", " + nbCellules + "]\n");
    }
}
```

```
00 // extrait d'une procédure de test
01 maListe.afficheGaucheDroiteRec();
02 try {
03     System.out.println("Élément en position 2 : " + maListe.getInfoAtPosit(2));
04     System.out.println("Élément en position 12 : " + maListe.getInfoAtPosit(12));
05 } catch (ExceptionMauvaisIndice e) {
06     System.out.println(e.getMessage());
07     System.out.println("Exception interceptée, poursuite de l'exécution du programme !");
08 }
09 System.out.println("\nL'exécution est terminée !");
```

```
01 affichage récursif de gauche à droite : nouveauUn -> entreUnEtDeux -> deux ->
03 Élément en position 2 : entreUnEtDeux
04 // getInfoAtPosit(12) déclenche la levée de l'exception ExceptionMauvaisIndice avec un message explicatif
// e.getMessage() permet de récupérer le message d'erreur associé à l'exception e
06 Indice de valeur 12 illégal dans getInfoAtPosit, la position doit être dans l'intervalle [1, 3].
07 Exception interceptée, poursuite de l'exécution du programme !
09 L'exécution est terminée !
```