Programmation orienté objet JAVA

Troisième partie

JAVA

Programmation orientée objet

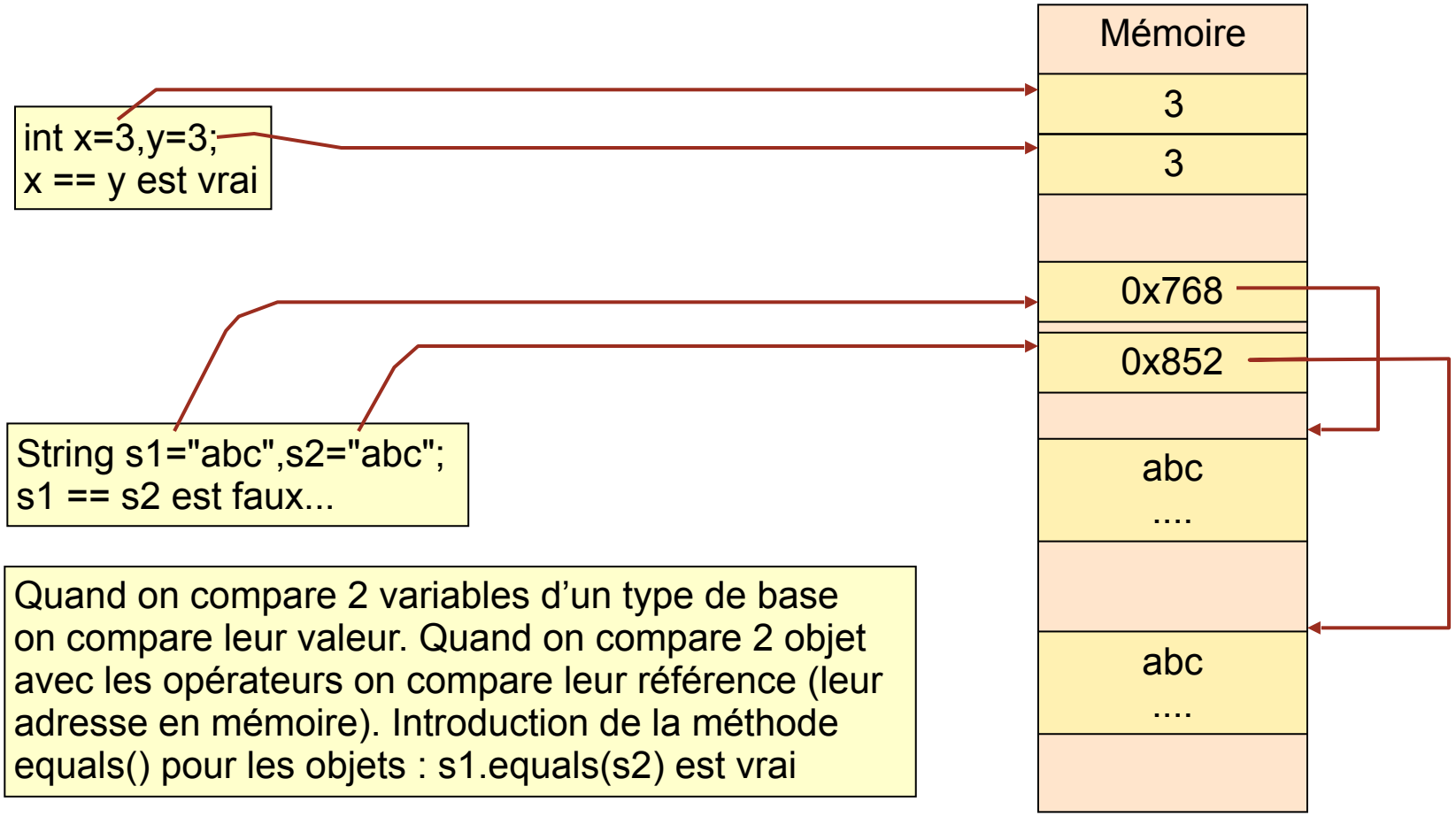
Hafidi Imad **imad.hafidi@gmail.com**

Objets prédéfinis

La classe Object

- Classe mère de toutes les classes.
- Possède des méthodes de base qu'il est possible de redéfinir :
  - **toString**()
  - **equals**() & **hashCode**()
  - **getClass**()
  - **clone**()
  - **finalize**()

Différences entre objets et types de base



Quand on compare 2 variables d'un type de base on compare leur valeur. Quand on compare 2 objet avec les opérateurs on compare leur référence (leur adresse en mémoire). Introduction de la méthode equals() pour les objets : s1.equals(s2) est vrai

La méthode equals

- Il existe déjà une méthode equals(Object) dans Object
- Mais son implantation test les références

```
Point p1=new Point(2,2);
Point p3=new Point(2,2);

p1==p3;          // false
p1.equals(p3); // false
```

- Pour comparer structurellement deux objet, il faut changer (on dit redéfinir) le code de la méthod equals()

Redéfinir equals

- Pourquoi equals ?
  - Car elle sert à cela.
- La plupart des classes de l'API redéfinissent la méthode equals

```
public class Point {
  private final int x,y;
  public Point(int x,int y) {
    this.x=x;
    this.y=y;
  }
  public boolean equals(Point p) {
    return x==p.x && y==p.y;
  }
}
```

CELA NE MARCHE PAS !!!

```
Point p1=new Point(2,2);
Point p3=new Point(2,2);
p1==p3;          // false
p1.equals(p3); // true

ArrayList points=new ArrayList();
points.add(p1);
points.contains(p3); // false
// pourtant contains utilise Object.equals(Object) ??
```

La VM ne fait pas la liaison entre Object.equals(Object) et Point.equals(Point)

- Ce n'est pas le même equals(), car il n'y a pas eut de redéfinition mais une autre définition (on dit surcharge)
- Point possède deux méthodes equals (equals(Object) et equals(Point))

```
Point p1=new Point(2,2);
Point p3=new Point(2,2);
p1.equals(p3); // true : Point.equals(Point)

Object o1=p1;
Object o3=p3;
o1.equals(o3); // false : Object.equals(Object)
```

- Il faut définir equals() dans Point de telle façon qu'elle remplace equals de Object
- Pour cela equals doit avoir la même signature que equals(Object) de Object

```
public class Point {
  private final int x,y;
  public Point(int x,int y) {
    this.x=x;
    this.y=y;
  }
  public boolean equals(Object o) {
    return x==p.x && y==p.y; // ce code ne marche plus
  }
}
```

Utiliser @Override

- @Override est une annotation qui demande au compilateur de vérifier que l'on redéfinie bien une méthode

```
public class Point {
  private final int x,y;
  public Point(int x,int y) {
    this.x=x;
    this.y=y;
  }
  @Override public boolean equals(Point p) {
    ...
  } // the method equals(Point p) in Point must override
    // a superclass method
}
```

- On demande dynamiquement à voir une référence à Object comme à Point (car on le sait que c'est un Point)

```
public class Point {
  ...
  @Override public boolean equals(Object o) {
    Point p=(Point)o; // ClassCastException si pas un Point
    return x==p.x && y==p.y;
  }
}
```

- [illegible] et pas lever une CCE

- On utilise instanceof qui renvoie vrai si une référence est d'un type particulier

```
public class Point {
  ...
  @Override public boolean equals(Object o) {
    if (!(o instanceof Point)) // marche aussi avec null
      return false;
    Point p=(Point)o;
    return x==p.x && y==p.y;
  }
}
```

Int hashCode()

- Renvoie un entier qui peut être utilisé comme valeur de hachage de l'objet
- Permet au objet d'être utiliser dans les tables de hachage

```
public class Point {
  public Point(int x,int y) {
    this.x=x;
    this.y=y;
  }
  @Override public int hashCode() {
    return x ^ Integer.rotateLeft(y,16);
  }
  @Override public boolean equals(Object o) {
    ...
}
```

hashCode et equals

- Tout objet redéfinissant equals doit redéfinit hashcode si l'objet doit être utilisé dans les Collection (donc tout le temps)
- equals et hashcode doivent vérifier
  - equals est symétrique, transitive, réflexive
  - x.equals(y) implique
  - x.hashcode()==y.hashcode()

- Les valeurs de hashcode doivent de préférence être différentes pour les objets du programme
- hashcode doit être rapide à calculer (éventuellement précalculée).

String toString

- Affiche un objet sous-forme textuelle.
- Cette méthode doit être utilisée que pour le deboggage

```
public class MyInteger {
  public MyInteger(int value) {
    this.value=value;
  }
  public String toString() {
    return Integer.toString(value); // ou ""+value;
  }
  private final int value;
  public static void main(String[] args) {
    MyInteger myi=new MyInteger(3);
    System.out.println(myi); // appel toString()
  }
}
```

Class<?> getClass()

- Permet d'obtenir un objet Class représentant la classe d'un objet particulier
- Un objet Class est un objet qui correspond à la classe de l'objet à l'exécution

```
String s="toto";
Object o="tutu";
s.getClass()==o.getClass(); // true
```

- Cette méthode est **final**
- Il existe une règle spéciale du compilateur indiquant le type de retour de getClass().

Clonage

- Permet de dupliquer un objet
- Mécanisme pas super simple à comprendre :
  - clone() a une visibilité protected
  - clone() peut lever une exception CloneNotSupportedException
  - Object.clone() fait par défaut une copie de surface si l'objet implante l'interface Cloneable
  - L'interface Cloneable ne définie pas la méthode clone()

Void Finalize

- Méthode testamentaire qui est appelé juste avant que l'objet soit réclamé par le GC
- Cette méthode est protected et peut lever un Throwable

```
public class FinalizedObject {
  protected @Override void finalize() {
    System.out.println("ahh, je meurs");
  }

  public static void main(String[] args) {
    FinalizedObject o=new FinalizedObject();
    o=null;
    System.gc(); // affiche ahh, je meurs
  }
}
```

Collections

Motivations

- 1, 2, … plusieurs
  - monôme, binôme, … polynôme
  - point, segment, triangle, … polygone
- Importance en conception
  - Relation entre classes
    - «… est une collection de … »
    - « … a pour composant une collection de … »
  - Choisir la meilleure structure collective
    - plus ou moins facile à mettre en œuvre
    - permettant des traitements efficaces
    - selon la taille (prévue) de la collection

Définition d 'une collection

- Une ***collection*** regroupe plusieurs données de même nature
  - Exemples : promotion d'étudiants, sac de billes, ...
- Une ***structure collective*** implante une collection
  - plusieurs implantations possibles
    - ordonnées ou non, avec ou sans doublons, ...
    - accès, recherche, tris (algorithmes) plus ou moins efficaces
- Objectifs
  - adapter la structure collective aux besoins de la collection
  - ne pas re-programmer les traitements répétitifs classiques (affichage, saisie, recherche d'éléments, …)

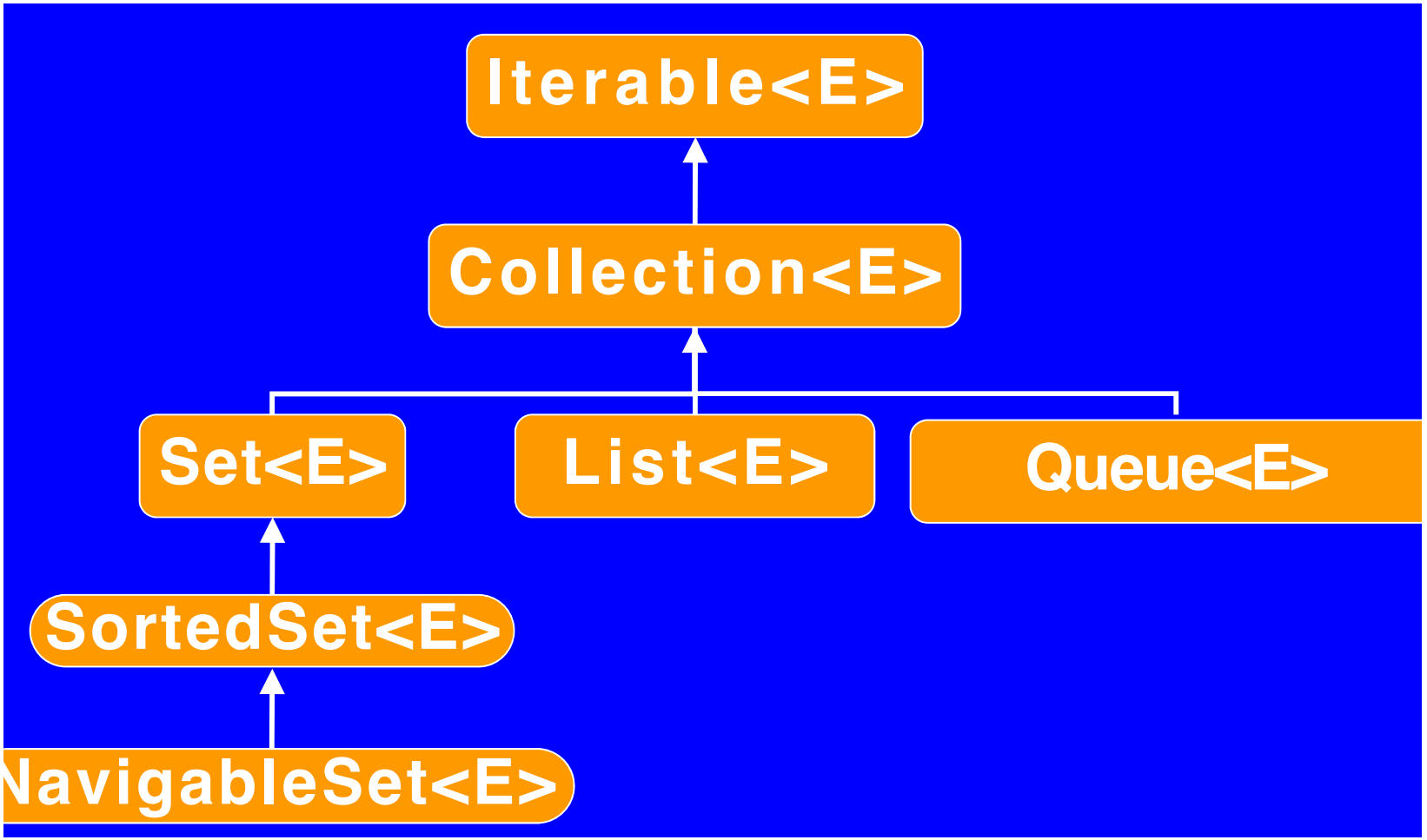
JDK & collections

- Le JDK fournit beaucoup types de collections sous la forme de classes et d'interfaces
- Ces classes et interfaces sont dans le paquetage **java.util**
- Avant le JDK 5.0, il n'était pas possible d'indiquer qu'une collection du JDK ne contenait que des objets d'un certain type ; les objets contenus étaient déclares de type **Object**
- A partir du JDK 5.0, on peut indiquer le type des objets contenus dans une collection grâce à la généricité : **List<Employe>**

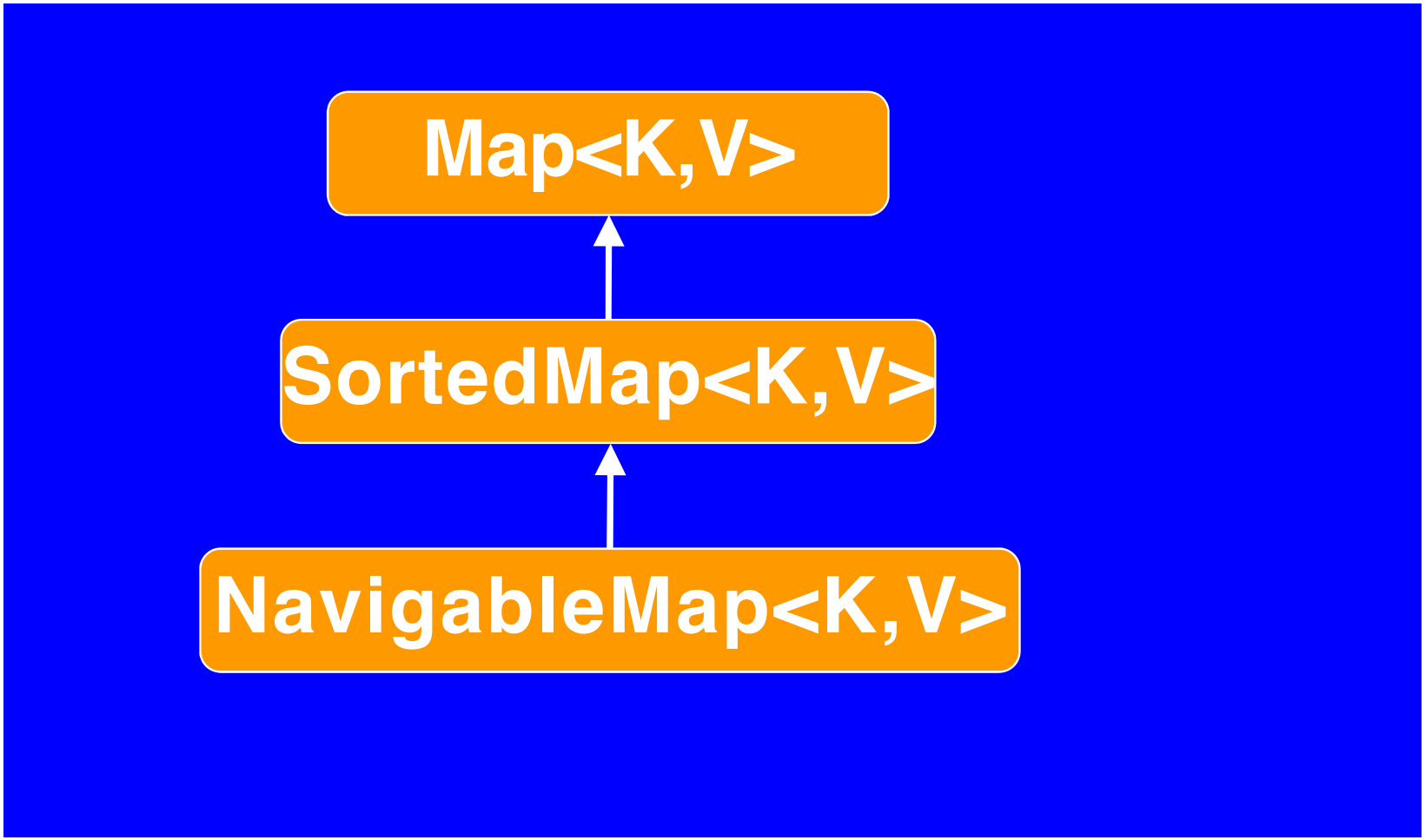
Les interfaces

- Des interfaces dans 2 hiérarchies d'héritage principales :
  - **Collection** correspond aux interfaces des collections proprement dites
  - **Map** correspond aux collections indexées par des clés ; un élément de type **V** d'une map est retrouvé rapidement si on connait sa clé de type **K** (comme les entrées de l'index d'un livre)

Hiérarchie des interfaces Collection



Hiérarchie des interfaces Map



Les classes abstraites

- **AbstractCollection<E>**, **AbstractList<E>**, **AbstractMap<K,V>**,... implantent les méthodes de base communes aux collections (ou *map*)
- Elles permettent de factoriser le code commun à plusieurs types de collections et à fournir une base aux classes concrètes du JDK et aux nouvelles classes de collections ajoutées par les développeurs

Les classes concrètes

- **ArrayList<E>**, **LinkedList<E>**, **HashSet<E>**, **TreeSet<E>**, **HashMap<K,V>**, **TreeMap<K,V>**,...
  héritent des classes abstraites
  - Elles ajoutent les supports concrets qui vont recevoir les objets des collections (tableau, table de hachage, liste chainée,...)
  - Elles implantent ainsi les méthodes d'accès à ces objets (*get*, *put*, *add*,...)

Les classes étudiées

- Nous étudierons essentiellement les classes **ArrayList** et **HashMap** comme classes d'implantation de **Collection** et de **Map**
  - Elles permettent d'introduire des concepts et informations qui sont aussi valables pour les autres classes d'implantation

Classes utilitaires

- **Collections** (avec un *s* final) fournit des méthodes **static** pour, en particulier,
  - trier une collection
  - faire des recherches rapides dans une collection triée
- •**Arrays** fournit des méthodes **static** pour, en particulier,
  - trier
  - faire des recherches rapides dans un tableau trié – transformer un tableau en liste

Collections en JDK1.1

- Les classes et interfaces suivantes, fournies par le JDK 1.1,
  - **Vector**
  - **HashTable – Enumeration**
- existent encore mais il vaut mieux utiliser les nouvelles classes introduites ensuite
- Il est cependant utile de les connaître car elles sont utilisées dans d'autres API du JDK
- Elles ne seront pas étudiées ici

Exemple List

```
List<String> l = new ArrayList<>();
l.add( "Mohammed Jarbi");
l.add(  " Mohammed Argoun");
l.add(" Najib Mahfoud");
l.add(" Jamal AFghani");
Collections.sort(l);
System.out.println(l);
```

Exemple Map

```
Map<String, Integer> frequences = new
HashMap<>();
for (String mot : args) {
        Integer freq = frequences.get(mot);
        if (freq == null)
                freq = 1;
        else
                freq = freq + 1;
                frequences.put(mot, freq);
        }
System.out.println(frequences);
```

Collections et types primitifs

- Les collections de **java.util** ne peuvent contenir de valeurs des types primitifs
  - Avant le JDK 5, il fallait donc utiliser explicitement les classes enveloppantes des types primitifs, **Integer** par exemple
  - A partir du JDK 5, les conversions entre les types primitifs et les classes enveloppantes peuvent être implicites avec le « boxing » / « unboxing »

L'interface collection <E>

- L'interface **Collection<E>**correspond à un objet qui contient un groupe d'objets de type **E**
- Aucune classe du JDK n'implante directement cette interface (les collections vont implanter des sous-interfaces de **Collection**, par exemple **List**)

Collection : méthodes communes

`boolean add(Object)` : ajouter un élément
`boolean addAll(Collection)` : ajouter plusieurs éléments
`void clear()` : tout supprimer
`boolean contains(Object)` : test d'appartenance
`boolean containsAll(Collection)` : appartenance collective
`boolean isEmpty()` : test de l'absence d'éléments
`Iterator iterator()` : pour le parcours (cf Iterator)
`boolean remove(Object)` : retrait d'un élément
`boolean removeAll(Collection)` : retrait de plusieurs éléments
`boolean retainAll(Collection)` : intersection
`int size()` : nombre d'éléments
`Object[] toArray()` : transformation en tableau
`Object[] toArray(Object[] a)` : tableau de même type que a

Interface List

- L'interface **List<E>**correspond à une collection d'objets indexés par des numéros (en commençant par 0)
- Classes qui implantent cette interface :
  - **ArrayList<E>**, tableau à taille variable
  - **LinkedList<E>**, liste chaînée
- On utilise le plus souvent **ArrayList**, sauf si les insertions/suppressions au milieu de la liste sont fréquentes

Nouvelles méthodes de List

```
void add(int indice, E elt)
boolean addAll(int indice, Collection<? extends E> c)
E get(int indice)
E set(int indice, E elt)
E remove(int indice)
int indexOf(Object obj)
int lastIndexOf(Object obj)
ListIterator<E>listIterator()
ListIterator<E> listIterator(int indice)
List<E> subList(int depuis, int jusqua)
```

Exemple : ajout d'éléments

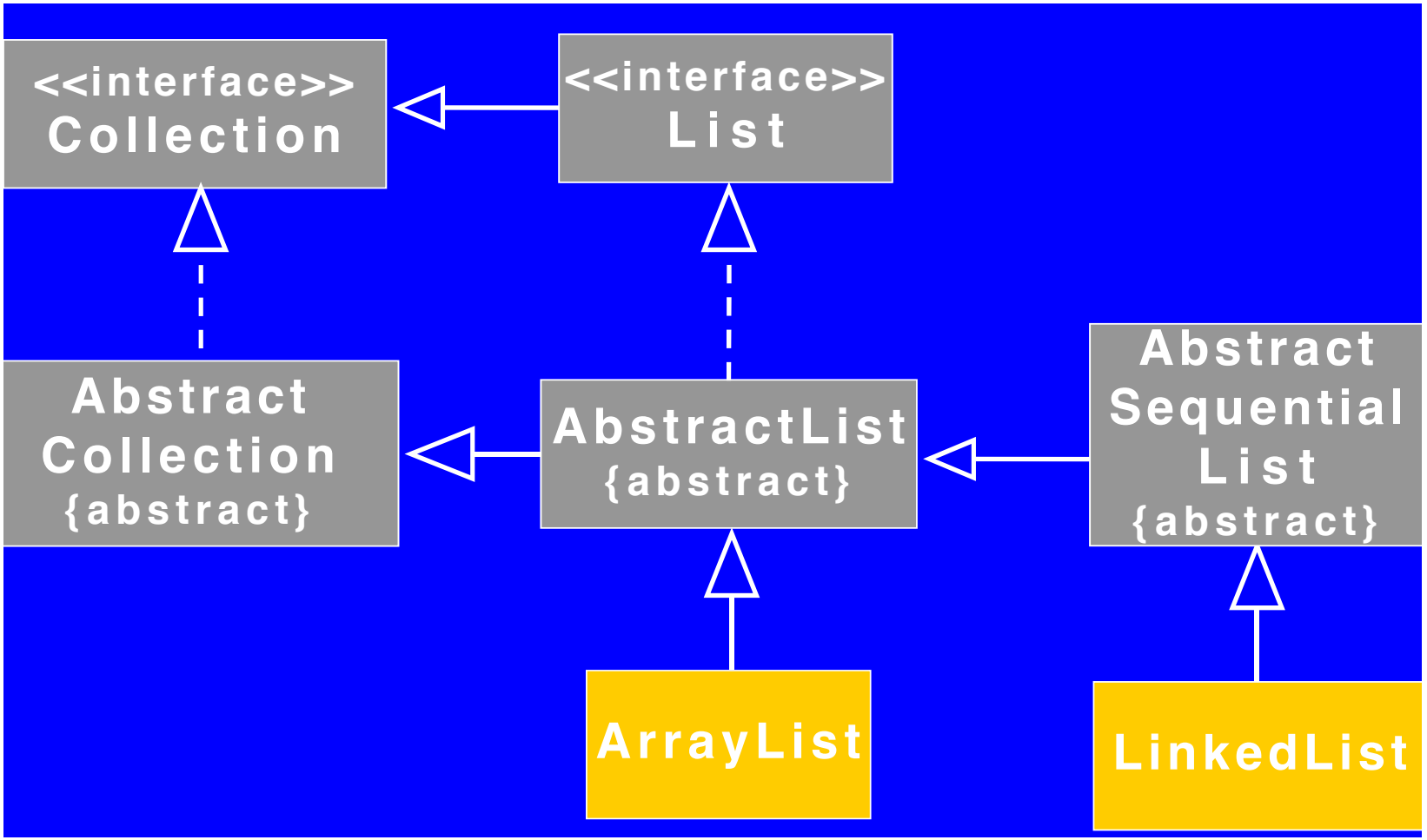
```
import java.util.*;

public class MaCollection {
   static final int N = 25000;
   List listEntier = new ArrayList();

   public static void main(String args[]) {
      MaCollection c = new MaCollection();
      int i;

      for (i = 0; i < N; i++) {
         c.listEntier.add(new Integer(i));
      }
   }
}
```

Interfaces et classes d'implémantation



Classe ArrayList

- Une instance de la classe **ArrayList<E>** est une sorte de tableau qui peut contenir un nombre quelconque d'instances d'une classe E
- Les emplacements sont indexés par des nombres entiers (à partir de 0)
- Constructeurs :
  - **ArrayList**()
  - **ArrayList(int taille initiale)**
  - **ArrayList(Collection<? extends E> c)** : pour l'interopérabilité entre les différents types de collections

Méthodes ArrayList

```
boolean add(E elt)
void add(int indice, E elt)
boolean contains(Object obj)
E get(int indice)
int indexOf(Object obj)
Iterator<E> iterator()
E remove(int indice)
E set(int indice, E elt)
 int size()
```

Recherche d'un élément

- Méthode
  - `public boolean contains(Object o)`
  - interface `Collection`, redéfinie selon les sous-classes
- Utilise l'égalité entre objets
  - égalité définie par `boolean equals(Object o)`
  - par défaut (classe `Object`) : égalité de références
  - à redéfinir dans chaque classe d'éléments
- Cas spéciaux
  - doublons : recherche du premier ou de toutes les occurrences ?
  - structures ordonnées : plus efficace, si les éléments sont comparables (voir tri)

Classe Collections

- Cette classe ne contient que des méthodes **static**, utilitaires pour travailler avec des collections :
  - tris (sur listes)
  - recherches (sur listes triées)
  - copie non modifiable d'une collection
  - minimum et maximum
  - ……

Trier une liste

- Si **l** est une liste, on peut trier **l** par :
  Collections.sort(l);
  - Cette méthode ne renvoie rien ; elle trie **l**
  - Pour que cela compile, les éléments de la liste doivent implémenter l'interface **java.lang.Comparable<T>** pour un type **T** ancêtre du type **E** de la collection

Interface Comparable<T>

- Cette interface correspond à l'implantation d'un ordre naturel dans les instances d'une classe
- Elle ne contient qu'une seule méthode :
  - **int compareTo(T t)**
  - Cette méthode renvoie
    - un entier positif si l'objet qui reçoit le message est plus grand que **t**
    - 0 si les 2 objets ont la même valeur
    - un entier négatif si l'objet qui reçoit le message est plus petit que **t**

- Toutes les classes du JDK qui enveloppent les types primitifs (**Integer** par exemple) implantent l'interface **Comparable**
- Il en est de même pour les classes du JDK **String**, **Date**, **Calendar**, **BigInteger**, **BigDecimal**, **File**, **Enum** et quelques autres
- Par exemple, **String** implémente **Comparable<String>**
- Il est fortement conseillé d'avoir une méthode **compareTo** compatible avec **equals** : **e1.compareTo(e2) == 0** *ssi* **e1.equals(e2)**
- **compareTo** lance une **NullPointerException** si l'objet passé en paramètre est **null**

Question

- Que faire :
  - si les éléments de la liste n'implantent pas l'interface **Comparable**,
  - ou si on veut les trier suivant un autre ordre que celui donné par **Comparable** ?

Réponse

- 1 On construit un objet qui sait comparer 2 éléments de la collection (interface **java.util.Comparator<T>**)
- 2 on passe cet objet en paramètre à la méthode **sort**

Interface Comprator<T>

- Elle comporte une seule méthode :
  - **int compare(T t1, T t2)** qui doit renvoyer
    - un entier positif si **t1** est « plus grand » que **t2**
    - 0 si **t1** a la même valeur (au sens de **equals**) que **t2**
    - un entier négatif si **t1** est « plus petit » que **t2**
- **Attention**, il est conseillé d'avoir une méthode **compare** compatible avec **equals** : **compare(t1, t2) == 0** *ssi* **t1.equals(t2)**

Interface Set

- Correspond à une collection qui ne contient pas 2 objets égaux au sens de **equals** (comme les ensembles des mathématiques)
- On fera attention si on ajoute des objets modifiables : la non duplication d'objets n'est pas assurée dans le cas où on modifie les objets déjà ajoutés

Méthodes de Set

- Mêmes méthodes que l'interface **Collection**
- Mais les « contrats » des méthodes sont adaptés aux ensembles
- • Par exemple,
  - la méthode **add** n'ajoute pas un élément si un élément égal est déjà dans l'ensemble (la méthode renvoie alors **false**)
  - quand on enlève un objet, tout objet égale (au sens de **equals**) à l'objet passé en paramètre sera enlevé

SortedSet<E>

- Un **Set** qui ordonne ses éléments
- L'ordre total sur les éléments peut être donné par l'ordre naturel sur **E** ou par un comparateur
- Des méthodes sont ajoutées à **Set**, liées à l'ordre total utilisé pour ranger les éléments

Méthodes de SortedSet

- **E first**() : 1er élément
- • **E last**() : dernier élément
- **Comparator<? super E> comparator**() : retourne le comparateur (retourne **null** si l'ordre naturel sur **E** est utilisé)
- Des méthodes renvoient des vues
- (**SortedSet<E>**) d'une partie de l'ensemble :
  - **subSet(E debut, E fin)** : éléments compris entre le début (inclus) et la fin (exclue)
  - **headSet(E fin)** : éléments inférieurs strictement au paramètre
  - **tailSet(E d)** : éléments qui sont supérieurs ou égaux au paramètre **d**

Implémentation Set

- Classes qui implémentent cette interface :
  - **HashSet<E>** implémente **Set** avec une table de hachage ; temps constant pour les opérations de base (**set**, **add**, **remove**, **size**)
  - **TreeSet<E>** implémente**NavigableSet** avec un arbre ordonné ; les éléments sont rangés dans leur ordre naturel (interface **Comparable<E>**) ou suivant l'ordre d'un **Comparator<? super E>** passé en paramètre du constructeur

Iterator

- Un itérateur (instance d'une classe qui implante l'interface **Iterator<E>**) permet d'énumérer les éléments contenus dans une collection
- Il encapsule la structure de la collection : on pourrait changer de type de collection (remplacer un **ArrayList** par un **TreeSet** par exemple) sans avoir à réécrire le code qui utilise l'itérateur
- Toutes les collections ont une méthode **iterator**() qui renvoie un itérateur

Méthodes itérateur

- **boolean hasNext()**
- **E next()**
- **void remove()**

```
List<Employe> le = new ArrayList<>();
Employe e = new Employe( "Jabri");
le.add(e);
Iterator<Employe> it = le.iterator();
 while (it.hasNext()) {
// le 1er next() fournit le 1er élément
        System.out.println(it.next().getNom());
}
```

Interface Iterable<T>

- Nouvelle interface (depuis JDK 5.0) du paquetage **java.lang** qui indique qu'un objet peut être parcouru par un itérateur
- Toute classe qui implémente **Iterable** peut être parcourue par une boucle « for each »
- L'interface **Collection** en hérite

Boucle for each

```
Iterator<Employe> it = coll.iterator();
it.hasNext(); ) {
        Employe e = it.next();
        String nom = e.getNom();
 }
```

```
for (Employe e : coll) {
  String nom = e.getNom();
}
```

Restriction for each

- On ne dispose pas de la position dans le tableau ou la collection pendant le parcours
- On ne peut pas modifier la collection pendant qu'on parcourt la boucle (alors que c'est possible par l'intermédiaire de l'itérateur)
- L'utilisation d'une boucle ordinaire avec un itérateur ou un compteur de boucle explicite est indispensable si ces 2 restrictions gênent

Interface Map<K,V>

- Il arrive souvent en informatique d'avoir à rechercher des informations en connaissant une clé qui permet de les identifier
  - Par exemple, on connait un nom et on cherche un numéro de téléphone ou on connait un numéro de matricule et on cherche les informations sur l'employé qui a ce matricule
- L'interface **Map<K,V>**correspond à un groupe de couples clé-valeur
  - Une clé repère une et une seule valeur
  - Dans la map il ne peut exister 2 clés égales au sens de **equals**()

Méthodes Map<K,V>

```
void clear()
boolean containsKey(Object clé)
boolean containsValue(Object valeur)
V get(Object clé)
boolean isEmpty()
Set<K> keySet()
Collection<V> values()
Set<Map.Entry<K,V>> entrySet()
V put(K clé, V valeur)
void putAll(Map<? extends K, ? extends V> map)
V remove(Object key) int size()
```

Interface *interne* **Entry<K,V>** de **Map**

- L'interface**Map<K,V>**contient l'interface interne **public Map.Entry<K,V>** qui correspond à un couple clé-valeur
- • Cette interface contient 3 méthodes
  - **K getKey**()
  - **V getValue**()
  - **V setValue(V valeur)**
- La méthode **entrySet**() de **Map** renvoie un objet de type « ensemble (**Set**) de **Entry** »

Modifications des clés

- La bonne utilisation d'une *map* n'est pas garantie si on modifie les valeurs des clés avec des valeurs qui ne sont pas égales (au sens de **equals**) aux anciennes valeurs
- Pour changer une clé, il faut d'abord enlever l'ancienne entrée (avec l'ancienne clé) et ajouter ensuite la nouvelle entrée avec la nouvelle clé et l'ancienne valeur

Récupérer les valeurs d'une **Map**

- On récupère les valeurs sous forme de C**ollection<V>** avec la méthode **values**() La collection obtenue reflétera les modifications futures de la *map*, et vice-versa
- On utilise la méthode **iterator**() de l'interface **Collection<V>** pour récupérer un à un les éléments

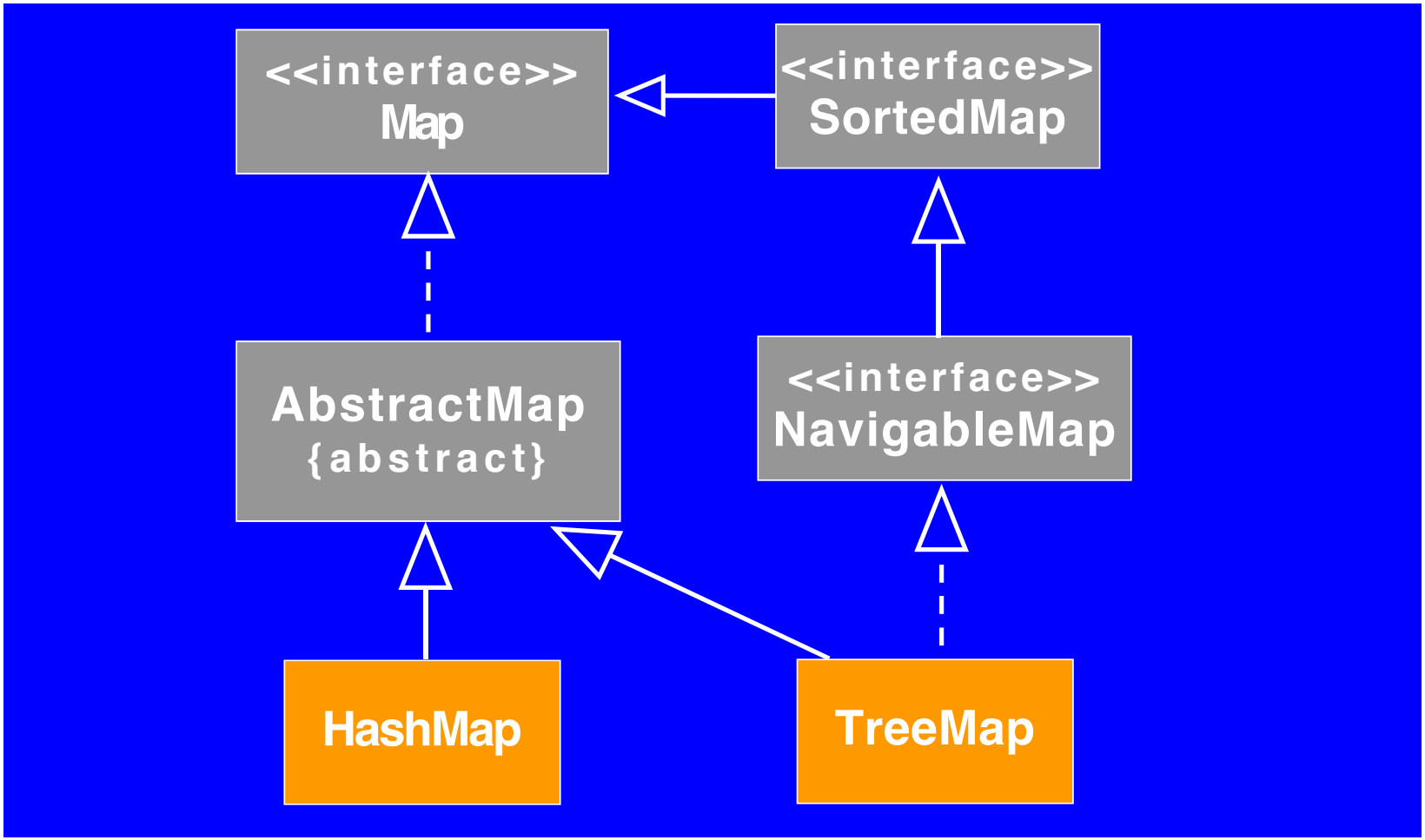
Récupérer les clés d'une **Map**

- On récupère les clés sous forme de **Set<K>** avec la méthode **keySet**() ; l'ensemble obtenu reflètera les modifications futures de la *map*, et vice-versa
- 2. On utilise alors la méthode **iterator**() de l'interface **Set<K>** pour récupérer une à une les clés

Récupérer les entrées d'une **Map**

- On récupère les entrées (paires clé-valeur) sous forme de **Set<Entry<K,V>>** avec la méthode **entrySet**() ; l'ensemble obtenu reflètera les modifications futures de la *map*, et vice-versa
- On utilise alors la méthode **iterator**() de l'interface **Set<Entry<K,V>>** pour récupérer une à une les entrées

Interface et classes d'implémentation



HashMap

- Structure de données qui permet de retrouver très rapidement un objet si on connait sa clé
- En interne l'accès aux objets utilise un tableau et une fonction de hachage appliquée à la clé
- La classe **HashMap<K,V>** implémente l'interface **Map<K,V>** en utilisant une table de hachage
- La méthode **hashCode**()(héritée de **Object** ou redéfinie) est utilisée comme fonction de hachage

Erreur

- On a vu que les classes qui redéfinissent **equals** doivent redéfinir **hashcode**
- Sinon, les objets de ces classes ne pourront être ajoutés à une **HashMap** sans problèmes
- Par exemple ces objets ne pourront être retrouvées ou pourront apparaître en plusieurs exemplaires dans la **HashMap**

Exemple HashMap

```
Map<String,Employe> hm = new HashMap<>();
Employe e = new Employe( "Omar");
e.setMatricule("E125");
hm.put(e.getMatricule(), e);
Employe e2 = hm.get("E369");
Collection<Employe> elements = hm.values();
 for (Employe employe : employes) {
        System.out.println(employe.getNom());
}
```

Exceptions

Prévoir les erreurs d'utilisation

- Certains cas d'erreurs peuvent être prévus à l'avance par le programmeur.
  exemples:
  - erreurs d'entrée-sortie (I/O fichiers)
  - erreurs de saisie de données par l'utilisateur

- Le programmeur peut :
  - «Laisser planter» le programme à l'endroit où l'erreur est détectée
  - Manifester explicitement le problème à la couche supérieure
  - Tenter une correction

Notion d'exception

En Java, les erreurs se produisent lors d'une exécution sous la forme d'exceptions.

Une exception :

- est un objet, instance d'une classe d'exception
- provoque la sortie d'une méthode
- correspond à un type d'erreur
- contient des informations sur cette erreur

Les exceptions

- Exception
  situation particulière imposant une rupture dans le cours d'un programme : erreur, impossibilité…

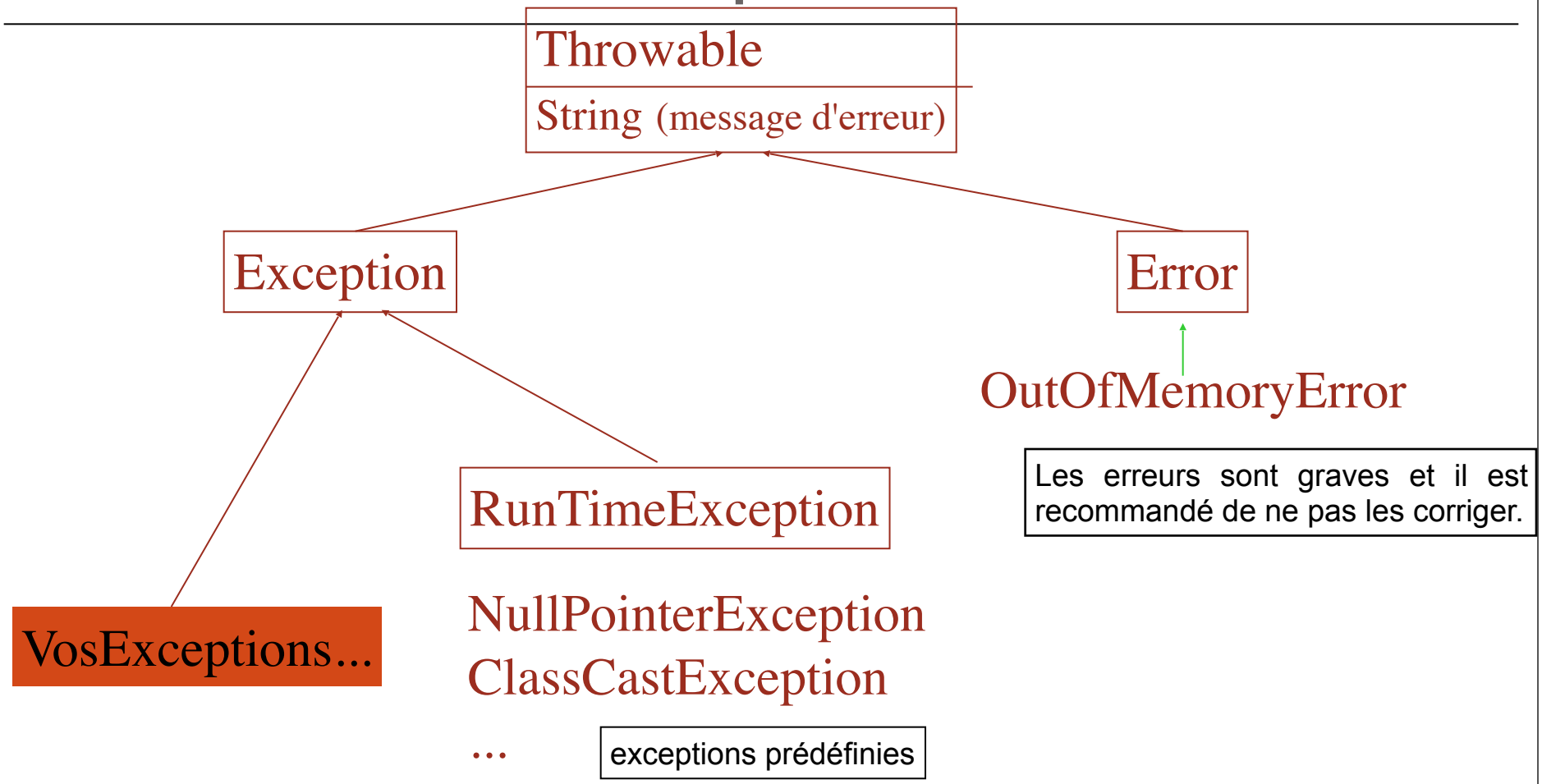
- Un objet JAVA `Exception` est une « bulle » logicielle produite dans cette situation qui va remonter la pile d'exécution pour trouver une portion de code apte à la traiter

- Si cette portion n'existe pas, le programme s'arrête en affichant la pile d'exécution. Sinon, la portion de code sert à pallier au problème (poursuite éventuelle, ou sortie)

Terminologie

- Une exception est un signal qui indique que quelque chose d'exceptionnel est survenu en cours d'exécution.
- Deux solutions alors :
  - laisser le programme se terminer avec une erreur,
  - essayer, malgré l'exception, de continuer l'exécution normale.
- Lever une exception consiste à signaler quelque chose d'exceptionnel.
- Capturer l'exception consiste à essayer de la traiter.

Arbre des exceptions



Nature des exceptions

- En Java, les exceptions sont des objets ayant 3 caractéristiques:
  - Un type d'exception (défini par la classe de l'objet exception)
  - Une chaîne de caractères (option), (hérité de la classe Throwable).
  - Un « instantané » de la pile d'exécution au moment de la création.
- Les exceptions construites par l'utilisateur étendent la classe *Exception*
- *RunTimeException, Error* sont des exceptions et des erreurs prédéfinies et/ou gérées par Java

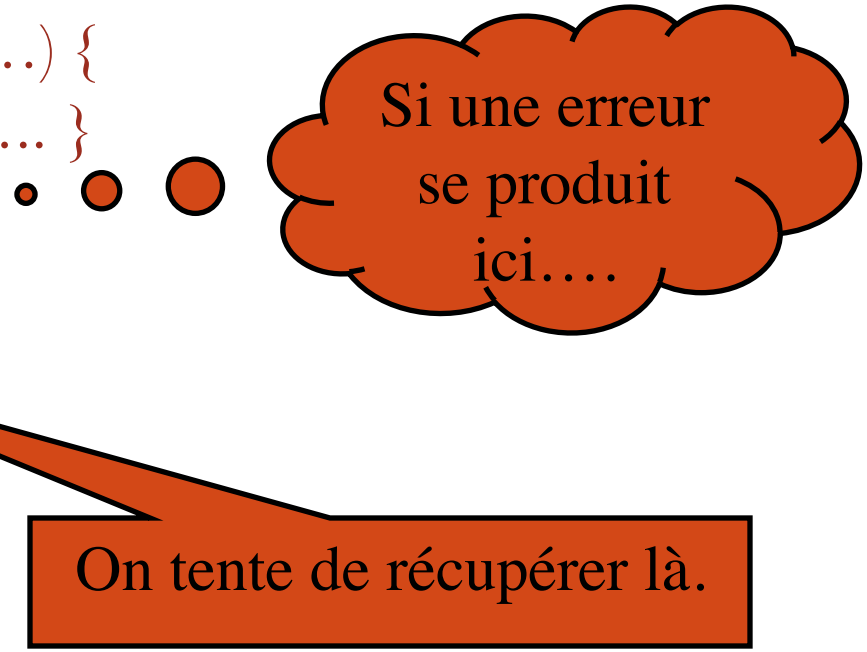
Quelques exceptions prédéfinies en Java

- Division par zéro pour les entiers : ArithmeticException
- Référence nulle : NullPointerException
- Tentative de forçage de type illégale : ClassCastException
- Tentative de création d'un tableau de taille négative : NegativeArraySizeException
- Dépassement de limite d'un tableau : ArrayIndexOutOfBoundsException

Capture d'une exception

- Les sections `try` et `catch` servent à capturer une exception dans une méthode (attraper la bulle…)
- exemple :

```
public void XXX(……..) {
        try{ ……..……. }
        catch {
         ……..
         ……..
        }
}
```



try / catch / finally

```
try
  {
    ...
  }
catch (<une-exception>)
  {
    ...
  }
catch (<une_autre_exception>)
  {
    ...
  }

...

finally
  {
    ...
  }
```

➜ Autant de blocs catch que l'on veut.

➜ Bloc finally facultatif.

Multi-catch

- Depuis le JDK 7 il est possible d'attraper plusieurs types d'exceptions dans un bloc catch :

```
try {
 ...
}
catch(Type1Exception | Type2Exception
Type3Exception e)
.....
}
```

- Aucune des exceptions du multi-catch ne doit être un sur-type d'une autre exception

Traitement des exceptions (1)

- Le bloc try est exécuté jusqu'à ce qu'il se termine avec succès ou bien qu'une exception soit levée.
- Dans ce dernier cas, les clauses catch sont examinées l'une après l'autre dans le but d'en trouver une qui traite cette classe d'exceptions (ou une superclasse).
- Les clauses catch doivent donc traiter les exceptions de la plus spécifique à la plus générale.
- Si une clause catch convenant à cette exception a été trouvée et le bloc exécuté, l'exécution du programme reprend son cours.

Traitement des exceptions (2)

- Si elles ne sont pas immédiatement capturées par un bloc catch, les exceptions se propagent en remontant la pile d'appels des méthodes, jusqu'à être traitées.
- Si une exception n'est jamais capturée, elle se propage jusqu'à la méthode main(), ce qui pousse l'interpréteur Java à afficher un message d'erreur et à s'arrêter.
- L'interpréteur Java affiche un message identifiant :
  - l'exception,
  - la méthode qui l'a causée,
  - la ligne correspondante dans le fichier.

Bloc finally

- Un bloc finally permet au programmeur de définir un ensemble d'instructions qui est toujours exécuté, que l'exception soit levée ou non, capturée ou non.
- La seule instruction qui peut faire qu'un bloc finally ne soit pas exécuté est System.exit().

Interception vs propagation

Si une méthode peut émettre une exception (ou appelle une autre méthode qui peut en émettre une) il faut :

- soit **propager** l'exception (la méthode doit l'avoir déclarée);
- soit **intercepter** et traiter l'exception.

Exemple de propagation

```
public int ajouter(int a, String str) throws NumberFormatException
    int b = Integer.parseInt(str);
    a = a + b;
    return a;
}
```

Exemple d'interception

```
public int ajouter(int a, String str) {
    try {
            int b = Integer.parseInt(str);
            a = a + b;
    } catch (NumberFormatException e) {
            System.out.println(e.getMessage());
    }
    return a;
}
```

Les objets `Exception`

- La classe Exception hérite de La classe Throwable.
- La classe Throwable définit un message de type String qui est hérité par toutes les classes d'exception.
- Ce champ est utilisé pour stocker le message décrivant l'exception.
- Il est positionné en passant un argument au constructeur.
- Ce message peut être récupéré par la méthode `getMessage()`.

Exemple

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

Levée d'exceptions

- Le programmeur peut lever ses propres exceptions à l'aide du mot réservé throw.
- throw prend en paramètre un objet instance de Throwable ou d'une de ses sous-classes.
- Les objets exception sont souvent instanciés dans l'instruction même qui assure leur lancement.

```
throw new MonException("Mon exception s'est produite !!!");
```

Emission d'une exception

- L'exception elle-même est levée par l'instruction throw.

- Une méthode susceptible de lever une exception est identifiée par le mot-clé throws suivi du type de l'exception

  exemple :

```
public void ouvrirFichier(String name)  throws MonException
        {if (name==null) throw new MonException();
                else
                 {...}
        }
```

throws (1)

- Pour "laisser remonter" à la méthode appelante une exception qu'il ne veut pas traiter, le programmeur rajoute le mot réservé throws à la déclaration de la méthode dans laquelle l'exception est susceptible de se manifester.

```
public void uneMethode() throws IOException
  {
    // ne traite pas l'exception IOException
    // mais est succeptible de la générer
  }
```

throws (2)

- Les programmeurs qui utilisent une méthode connaissent ainsi les exceptions qu'elle peut lever.
- La classe de l'exception indiquée peut tout à fait être une super-classe de l'exception effectivement générée.
- Une même méthode peut tout à fait "laisser remonter" plusieurs types d'exceptions (séparés par des `,`).
- Une méthode doit traiter ou "laisser remonter" toutes les exceptions qui peuvent être générées dans les méthodes qu'elle appelle.

Conclusion

- Grâce aux exceptions, Java possède un mécanisme sophistiqué de gestion des erreurs permettant d 'écrire du code « robuste »
- Le programme peut déclencher des exceptions au moment opportun.
- Le programme peut capturer et traiter les exceptions grâce au bloc d'instruction `catch … try … finally`
- Le programmeur peut définir ses propres classes d 'exceptions