Java et OOP
Paquetages standards

Paquetages standards

- `java.lang` : les bases du langage.
- `java.io` : entrées-sorties.
- `java.util` : utilitaires, structures de données.
- `java.text` : internationalisation.
- `java.awt` : graphisme (Abstract Window Toolkit).
- `java.applet` : applets HTML.
- `java.rmi` : objets distribués (Remote Method Invocation).
- `java.net` : réseau.
- `java.math` : calculs en précision arbitraire.
- `java.sql` : bases de données (JDBC).
- `java.security` : cryptage, authentification etc.

Paquetage `java.lang`

- Des classes permettant d'encapsuler les types élémentaires dans des objets : `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, `Short`.
- Des classes pour la manipulation des chaînes de caractères : `String`, `StringBuffer`
- La librairie mathématique standard (classe `Math`.
- Paquetage importé automatiquement.

`String`

- Classe finale !
- Implémentant les interfaces `CharSequence, Comparable`.
  `CharSequence` permet l'accès uniforme et en **lecture seule** à une séquence de caractères.
- Exemples de constructeurs :
  - `String(char[])`, `String(char[] value, int offset, int count)`.
  - `String(String), String(StringBuffer)` – copie.
- Méthodes héritées :
  - `Object compareTo(Object)` – héritée de `Comparable` !
    Cette méthode lève `ClassCastException` si le paramètre n'est pas un String.
  - `char charAt(int), int length(), CharSequence subsequence(int start, int end)` – héritées de `CharSequence` !
    `charAt` et `subsequence` lèvent `IndexOutOfBoundsException` si indice(s) hors limites.
  - `Object equals(Object), String toString()` – héritée de `Object`.

String

- Recherche dans la chaîne : `int indexOf(char)`, `indexOf(String)`, `lastIndexOf(char)`, `lastIndexOf(String)`.
- Comparaison en ordre lexicographique : `int compareTo(String)`.
- Égalité partielle : `int regionMatches(int start, String other, int oStart, int len)`.
- Test début/fin de chaîne : `boolean startsWith(String)`, `boolean endsWith(String)`.
- Sous-chaîne : `String substring(int deb, int fin)`, `void getChars(int begin, int end, char[] dest, int dinit)`.
  Lèvent `IndexOutOfBoundsException`.
- Concaténation : `String concat(String s)` – renvoie une *nouvelle chaîne* qui représente la concaténation de `this` avec le String `s`.
- Changement de case : `toLowerCase()`, `toUpperCase()`
- Substitution de caractères : `String replace(char old, char new)`.
- Conversion en chaînes de caractères : `staticString valueOf(int)`, `valueOf(float)`, etc.

StringBuffer

- Héritière de `Object` et implémentant `CharSequence`.
- Permet les **modifications**, ce que la classe `String` ne le permet pas.
- Constructeurs : `StringBuffer()`, `StringBuffer(String)`, `StringBuffer(int capacité)`.
- Chaque StringBuffer a une capacité qui peut accroître au cours de l'utilisation.
  - `int capacity()` : renvoie la capacité courante.
  - `ensureCapacity(int mincap)` : l'augmente, si nécessaire.
- Reprend quelques méthodes de String : `getChars`, `indexOf`, `length`, `substring`.
- Méthodes spécifiques : `append` et `insert`, surchargées pour accepter des données de n'importe quel type.
  - `StringBuffer append(float)`, `append(int)`,... , `append(char[], int offset, int len)` pour rajouter à la fin du `this` la conversion en chaîne de caractères de l'argument.
  - `StringBuffer insert(int où, float)`, `insert(int où, char[])`,...
  - `void setCharAt(int pos, char rempl)`.
- Pour effacer : `StringBuffer delete(int d, int f)`, `deleteCharAt(int pos)`.
- Est utilisé dans la concaténation des Strings : `x = "bla" + 4` est compilée dans l'équivalent du `new StringBuffer().append("a").append(4).toString()`

Paquetage `java.util`

- Interfaces `Collection`, `Iterator`, `Map` et leur héritières.
- Des implémentations de ces interfaces : `Vector`, `ArrayList`, `AbstractCollection`, `AbstractSet`, `LinkedList`, `Stack`.
- Des implémentations polymorphes des algos de tri, recherche, min/max, mais aussi de destruction de l'ordre (*shuffling*)
- D'autres classes parfois utiles : `Calendar`, `Currency`, `Date`, `Random`, `StringTokenizer...`
- Exceptions définies : `EmptyStackException`, `NoSuchElementException`.

`AbstractCollection/List/Map/SequentialList/Set`

- Des classes **abstraites** qui soulagent certains programmeurs de la tâche d'implémenter toutes les méthodes déclarées par les interfaces respectives, `Collection/List/Map` etc.
- Pour les utiliser, un programmeur a besoin de réécrire moins de méthodes que pour implémenter les interfaces.
- Certaines méthodes des interfaces mentionnées sont **optionelles** : il n'est pas nécessaire de les implémenter lors d'un héritage.
- Le programmeur doit seulement hériter de la classe qu'il veut, et de redéfinir certaines méthodes (bcp moins que dans les interfaces).
- Mais bien sûr, il peut redéfinir les méthodes non-abstraites qu'il n'aime pas !
- Contiennent des *constructeurs protégés* !
- En principe, pour les classes héritières il faut définir un constructeur sans arguments et un constructeur de copie.
- Par contre, *il n'y a pas* d'implémentation standard de l'interface `Iterator` !

`AbstractCollection`

- Classe **abstraite** implémentant la majorité des fonctionnalités de l'interface `Collection`
- Constructeur sans arguments et **protégé**.
- Méthodes **abstraites** :
  - `abstract Iterator iterator()`
  - `abstract int size()`
- Méthodes implémentées :
  - Rajout d'éléments : `boolean add(Object o)`, `addAll(Collection c)`.
  - Test d'appartenence : `boolean contains(Object o)`, `containsAll(Collection c)`.
  - Test de collection vide : `boolean isEmpty()`.
  - Suppression : `void clear()`, `boolean remove(Object o)`, `removeAll(Collection c)`, `retainAll(Collection c)`.
  - Conversion : `Object[] toArray()`, `String toString()`.
- Exceptions levées `UnsupportedOperationException`, `NullPointerException`, `ClassCastException`, `IllegalArgumentException`.

AbstractList

En tant que héritière de l'interface `List`, elle implémente les mêmes méthodes que la classe `AbstractCollection`, plus ce qui est spécifique à l'interface `List`.

- *Rappel* : l'interface `List` permet d'organiser des objets dans des listes suivant un ordre d'apparition.
- Rajout à un indice donné : `void add(int ind, Object o)`, qui lève `IndexOutOfBoundsException`.
- Renvoi de l'élément d'indice donné `abstract Object get(int ind)`. méthode à redéfinir lors de l'héritage !
- Élimination de tous les éléments entre deux indices donnés : `protected void removeRange(int de, int à)`.
- Remplacement d'un objet à un indice donné : `Object set(int ind, Object o)`.
- Création d'un iterateur
  - de type Iterator : `Iterator iterator()`
  - de type ListIterator : `ListIterator listIterator()`

  Ces sont des méthodes non-abstraites ! c'est normal, on sait qu'il y a un ordre.
- Un **attribut** est aussi disponible en mode protégé : `modCount`.

Essayez de découvrir aussi ce qui se cache derrière les autres héritières abstraites des interfaces à l'adresse `http ://java.sun.com/j2se/1.4.2/docs/api` !

Classe `ArrayList`

- Implémentation "concrete" de l'interface `List`.
- Constructeurs : `ArrayList()`, `ArrayList(Collection a)`, `ArrayList(int cap)`
- Toutes les opérations sont conçues à être exécutées en temps optimal :
  - En temps *constant* : `int size()`, `boolean isEmpty()`, `Object get()`, `Object set(int index, Object o)`, `Iterator iterator()`, `ListIterator listIterator()`.
  - En temps linéaire : `boolean add(Object o)`, `add(int où, Object o)`.
  - En temps *linéaire amorti* : les autres, e.g. `boolean addAll(Collection c)`, `void clear()`, `boolean contains(Object o)`...
- Comme les StringBuffer, chaque ArrayList a une capacité :
  - Initialisée dans le constructeur `ArrayList(int)`.
  - Incrémentée par chaque opération d'ajout `add(...)`, `addAll(...)`.
  - Modifiée avec `void ensureCapacity(int cap)`.
- Il est plus utile d'incrémenter d'un seul coup la capacité d'un ArrayList (de même pour un StringBuffer !) si on a une longue séquence de rajouts :

  La réallocation de la mémoire d'un seul coup prend toujours moins de temps qu'une longue séquence de réallocations.

La classe `Vector`

- Un autre exemple d'extension de la classe `AbstractList`.
- Une différence imporante avec `ArrayList` réside dans son comportement par rapport aux threads – dont on ne va pas parler...
- Attributs : `protected int capacityIncrement ; protected int elementCount ; protected Object[] elementData ;`
- C'est une classe très amicale, elle nous permet d'accéder (en tant que hériteur !) à son vecteur d'éléments !
- Constructeurs : `Vector(), Vector(Collection c), Vector(int capinit), Vector(int capinit, int capinc)`.
- Méthodes supplémentaires (par rapport à AbstractList) :
  - `int addElement(Object o), setElement(Object o, int où)`.
  - `void copyInto(Object[] dest)`.
  - `Enumeration elements()`.
  - `Object firstElement(), lastElement()`.
  - `int removeElement(Object o), removeElementAt(int où), removeAllElements()`.
  - `void trimToSize()`.

La classe `Stack`

- Étend la classe `Vecteur` par les fonctionnalités d'une pile.
- Constructeur sans arguments.
- Test de pile vide : `boolean empty()`.
- Renvoi de l'élément du sommet de la pile sans le dépiler : `Object peek()`.
- Dépilage d'un élément : `Object pop()`.
- Empilage d'un élément : `Object push()`.
- Recherche d'un élément (position par rapport au *sommet* de la pile, qui est considéré d'indice 1) : `int search(Object o)`.

La classe `Arrays`

- Classe contenant *que* des méthodes statiques.
- Implémentation des algos génériques de recherche *binaire* :
  - Séquence d'éléments de type primitif : `static int binarySearch(float[], float)` et toutes les autres versions.
  - Le même algorithme générique : `static int binarySearch(Object[], Object)`.
  - Renvoie l'indice de l'objet cherché.
  - La séquence d'objets doit être ordonnée – si non-ordonnée le résultat peut ne rien avoir avec le fait que l'objet se trouve ou non.
- Implémentation des algos génériques de tri :
  - `static void sort(int[])` et pour tous les types primitifs.
  - `static void sort(Object[])` pour le tri générique, les `Object`s doivent *tous* être `Comparable` (ordre total !).
- Test d'égalité entre deux séquences : `static boolean equals(int[], int[])` et tous les autres types primitifs, et aussi pour des séquences d' `Object`s.
- Initialisation d'une séquence ou d'une sous-séquence avec la même valeur :
  - `static void fill(int[], int)` etc.
  - `static void fill(int[] seq, int de, int à, int avec)`.

Flux d'entrée-sortie

- Concept de **flux** : séquence d'informations produite par une source (programme, utilisateur) et qui est principalement lue dans la même ordre de production.
- Lecture-écriture *séquentielle*.
- Exemples : fichiers, entrées/sorties/communication à travers le réseau, etc.
- Mais aussi on peut lire dans une séquence (e.g. String) de façon séquentielle !
- Deux hiérarchies indépendantes de flux :
  - **Flux d'octets** – lecture/écriture sur des supports par octets.
  - **Flux de caractères** – lecture/écriture sur des supports en 16 bits.
- Mais on va grouper les classes plutot par leurs fonctionnalités que par les types de flux sur lesquelles elles travaillents.
- Certaines classes font des lectures/écritures *directes* dans les flux, d'autres processent les données après les avoir lus, resp. avant de les écrire.

Hiérarchies de classes d'I/O

- Quelques descendants de `Reader` : `BufferedReader`, `CharArrayReader`, `InputStreamReader`, `FilterReader`, `StringReader`.
- Descendants de `Writer` : `BufferedWriter`, `CharArrayWriter`, `OutputStreamWriter`, `FilterWriter`, `StringWriter`.
- Descendants de `InputStream` : `FileInputStream`, `FilterInputStream`, `ByteArrayInputStream`, `StringBufferInputStream`, `ObjectInputStream`.
  - `System.in` est un objet de type `InputStream` !
- Descendants de `OutputStream` : `FileOutputStream`, `FilterOutputStream`, `ByteArrayOutputStream`, `ObjectInputStream`.
  - Desccendant de `FilterOutputStream` : `DataOutputStream`, `PrintStream`.
  - `System.out` est un objet de type `PrintStream` !
- Certaines classes lisent ou écrivent directement dans leur destination, d'autres (comme `FilterInputStream`, `FilterOutputStream`) font certaines opérations de processage des données.

Méthodes principales

Pour les classes `Reader` et `InputStream` :

- `int read()` pour les deux, pour lire un seul caractère, resp. octet .
- `read(char[] buf)` pour lire dans un `InputStream` une séquence de caractères.
- `read(byte[] buf)` pour lire dans un `Reader` une séquence d'octets.
- Toutes ces méthodes lèvent l'exception `IOException` qui *doit être attrapée ou re-levée* ! (ce n'est pas une RunTimeException !)
- `long skip(long nb)` pour "sauter" un nbre d'éléments.
- `abstract void close()` pour fermer le flux.

Pour les classes `Writer` et `OutputStream` :

- `void write(int c)` pour les deux, pour écrire un seul caractère, resp. octet .
- `write(char[] buf)` pour écrire dans un `OutputStream` une séquence de caractères.
- `write(byte[] buf)` pour écrire dans un `Writer` une séquence d'octets.
- Toutes ces méthodes lèvent l'exception `IOException` qui *doit être attrapée ou re-levée* ! (ce n'est pas une RunTimeException !)
- `flush` pour forcer tout tampon d'écriture (invisible pour le programmeur !) d'être écrit sur le support.
- `abstract void close()` pour fermer le flux.

Flux "concrets"

- Lecture/écriture séquentielle dans un tableau en mémoire :
  - `CharArrayReader, CharArrayWriter`, `ByteArrayInputStream, ByteArrayOutputStream`.
  - Pour les lecteurs, le tableau est passé en paramètre au constructeur, e.g. `CharArrayReader(char[] tableau)`.
  - Pour les écrivains, le tableau est créé et renvoyé par `char[] toCharArray()`, `byte[] toByteArray()`.
- Lecture/écriture séquentielle dans fichiers
  - `FileReader, FileWriter`, `FileInputStream, FileOutputStream`.
  - Le nom du fichier est spécifié dans le constructeur, e.g. `FileReader(String nom_fichier)`.
- Lecture/écriture dans une chaîne de caractères :
  - `StringReader/StringWriter`.
  - Pour le lecteur, la chaîne est donnée dans le constructeur `StringReader(String)`.
  - Pour l'écrivain, la chaîne est récuperée par `String toString()`

Sont utilisés plutot par des constructions parallèles (threads).

Conversions pour flux binaire

Pour lire ou écrire autre chose que octets ou caractères on interface avec des classes plus évoluées qui font des conversions avec les flux "concrets" de bas niveau.

- Flux données binaires pour les types primitifs :
  `DataInputStream/DataOutputStream`
  - Couche au dessus d'un flux d'octets : constructeur `DataInputStream(InputStream)` resp. `DataInputStream(InputStream)`.
  - Méthodes de lecture adaptée à chaque type : `readInt()`, `readFloat()` etc.
  - Par contre, codage en octal !
    Donc ne peut être utilisé pour lire de l'entrée standard que si vous entrez des données en octal.
  - Sont plutot utilisés en couple, e.g. pour écrire dans un fichier, et ensuite lire dans ce fichier.
- Flux de données de type objets : `ObjectInputStream/ObjectOutputStream`.
  - Couche au dessus d'un flux d'octets.
  - Même méthodes que `DataInputStream`, en plus `Object readObject()` et `void writeObject(Object o)`.
  - Les objets doivent être d'une classe implémentant l'interface `Serializable`.

Autres conversions de flux

- Flux avec tampons : `BufferedInputStream/BufferedOutputStream`, `BufferedReader/BufferedWriter`.
- Permettent l'encapsulation de flux de bas niveau pour lecture à travers un tampon.
- Le tampon est invisible pour le programmeur et pour l'utilisateur !
- Constructeurs :
  - `BufferedReader(Reader)` et similaire, avec initialisation par défaut de la taille du tampon (qui en général est suffisante).
  - `BufferedReader(Reader, int dim)` pour donner explicitement une taille au tampon.
- Conversions flux octets/flux caractères : `InputStreamReader/OutputStreamWriter`.
  - Constructeurs `InputStreamReader(InputStream)` et similaire, on peut aussi spécifier le *charset*.
- Flux pour écriture sur la sortie standard : classe `PrintWriter` avec les méthodes `print(...)`, `println(...)` connues pour `System.out`.

Qu'est-ce qu'on fait pour lire un entier ?

- Le `System.in` est l'entrée standard pour un programme.
- C'est un `InputStream`, donc renvoie des octets bruts qu'on ne peut pas manipuler de manière très intélligente...
- Alors on le transforme en `InputStreamReader`, qui transforme pour nous les octets lus en caractères.
- Mais même `InputStreamReader` n'est pas très évolué, il renvoie directement les caractères qui arrivent, sans les gardant, et donc il est fort probable qu'on ne peut pas communiquer à travers lui.
- On a besoin donc d'un `BufferedReader` qui suppose l'existence d'un tampon où les caractères sont stoqués dès leur arrivée, et qu'on peut récupérer sans se soucier de la vraie implémentation du tampon mémoire.
- Finalement, si ce qu'on a écrit au clavier est une donnée d'autre type que `char`, on doit la convertir dans son type avec les méthodes des classes encapsulant les types primitifs, e.g. `Integer` et companie.
- Toutefois, chaque appel à la lecture dans notre flux doit attraper l'exception `IOException` !