CHAPITRE 1 : Des classes et des objets

méthode const, ne modifient pas les variables d'instance, peut seulement les lire (pour les get). (const à la fin de la déclaration)

méthode inline : définie dans le header. théorie : éxécution + rapide mais exécutable plus lourd. Réalité : pratique pour les petites méthodes appelées souvent

Pointeurs vs. références : pointeurs en C++ vs. références en java : pas d'arithmétique des pointeurs, l'adresse est une valeur cachée, mais se comporte comme un pointeur

encapsulation : séparer la *spécification* (déclaration des méthodes, interface (API)) et *l'implémentation* (variables et définitions de méthodes, interne à l'objet)

Java : **finalize()** appelée juste avant destruction. peu utilisée (ramasse-miettes)

les classes de bases polymorphes doivent avoir un destructeur virtuel.

pointeurs pendants les mettre à nullptr après delete

alternative à la surcharge : paramètres par défaut (comme en python, pas en Java)

static : comme en Java, définie dans un seul fichier .cpp OU BIEN `const int` / `constexpr` (C++11)

namespace : comme un package `namepspace nom_du_package { class Circle {}; }`, similaire à *package* en Java (**using namespace** : similaire à *import* en Java)

streams : package `<iostream>`, std::cout/cerr/cin/ostream/istream/ofstream/ifstream

CHAPITRE 2 : héritage et polymorphisme

C++ : **héritage multiple**

polymorphisme : choix de la méthode (celle du père ou du fils) :

- Java : liaison dynamique/tardive : choix de la méthode à l'exécution, appelle la méthode du pointé
- C++ : avec virtual (comme en Java), sans virtual : liaison statique (méthode du pointeur)

non virtuelle si : classe/méthode pas héritée, méthode utilise très très très souvent

méthode abstraite: `virtual void setWidth(int) = 0;`

classe abstraite : classe qui possède au moins une méthode abstraite

interface : classe totalement abstraite

POO : 4 fondamentaux : méthodes, encapsulation, héritage, polymorphisme d'héritage

CHAPITRE 3 : Gestion mémoire

- **mémoire automatique** (pile) En C++ certains objets peuvent être contenue directement dans des variables (pas possible en Java, que les types de base)
- **mémoire globale/statique** : existent au début et à la fin du programme
- **mémoire dynamique** (tas) : pointé, créé par new, détruit par delete

($\neq$ Java) C++ traite certains objets comme des types de bases (String), et un objet peut avoir une variable d'instance contenant directement un autre objet.

copie superficielle champs à champs (pointe vers le même) / **copie profonde** copie les pointés

Java, `=` est une copie superficielle

```
Car(const Car& from) = delete;
Car& operator=(const Car& from) = delete;
```

CHAPITRE 4 : Types, constantes et smart pointers

const porte sur ce qui suit. (devant le type ou "*" c'est le pointé, devant la variable c'est le pointeur)

objets immuables : soit la classe n'a pas de méthodes permettant de le modifier, soit on déclare l'objet avec *const* et seules les méthodes const peuvent être appelées

Constance logique : un objet avec un attribut *mutable* n'a pas de méthode permettant de le modifier, sauf son attribut *mutable* qui lui est modifiable (non-constance physique)

unique_ptr (1 seul ref, moins coûteux), **weak_ptr** (compte pas, rel circulaire)

CHAPITRE 5 : Bases des Templates et STL

STL : Standard template library, (std::) avec des conteneurs, itérateurs et algorithme s'appliquant à ces conteneurs

```
for (auto it : path) { it->print();}
```

détruire une liste et ses objets pointés : (1) `for (auto it : path) { delete it; }` / (2) smart pointers / (3) contenir objets (pas si polymorphisme)

différence liste/vecteur/deque : vecteur (accès direct aux éléments), liste (insertion/suppr facile), deque (les deux, mais coûteux en mémoire)

supprimer un élément d'une liste :

```
for (auto it = path.begin(); it != path.end(); ) {
  if ((*it)->x != value) it++;
  else {
    auto it2 = it;
    ++it2;
    delete *it;
    path.erase(it);
    it = it2;
```

```
  }
}
```

Generics Java : pas de type de bases, pas instanciés à la compilation (+ de CPU mais - de mémoire), pas de spécialisation, pas de traitement sur les types

CHAPITRE 6 : Passage par valeur et par référence

Par défaut, c'est un passage par valeur (Java et C++).

const : pour que le pointé `*p` ne puisse pas changer (Java, toujours immutable) ET pour éviter que l'attribut soit modifié/supprimer à travers une modification ultérieure du paramètre.

C++ permet le passage par référence grâce à `&` (PAS JAVA)

Solution en Java, modifier les pointés (méthode .replace), utilier "*" en C++ marche aussi

Références C++ (éviter les calculs de pointeurs dangereux) `Circle c; Circle & ref = c;`

CHAPITRE 7 : Compléments

pointeurs de fonctions : `type(*fun)` en paramètre (doit mettre un & pour un pointeur de méthode)

foncteurs objet considérés comme des fonctions `void operator()(Object&) {}`

lambdas : déclarés `std::function<res(const Data&)>` puis utilisés `[var](const Data& d) {return d.var > var}`

exceptions : `try {} catch() {}` pas besoin de les spécifier en C++ ($\neq$ Java)

Langages de programmation

paradigmes : *impérative* (Ada, C, Pascal, Fortran, Cobol), *programmation à objets et à acteurs* (C++, Java, Eiffel), *fonctionnelle* (Lisp, Scheme, Caml), *logique* (Prolog)

POO : définition et intéractions de briques logicielles appelées objets.

Programmation impératives Suite d'instruction. Deux types : affectation et saut (conditionnel ou non) ... et ensuite on a rajouté les boucles (construites à partir de ça)

Théorème de Boehm et Jacopini : Tout programme comportant éventuellement des sauts conditionnels peut être réécrit en un programme équivalent n'utilisant que les structures de contrôle while et if-then.

Paradigme fonctionnel

les arguments sont des séquences, leurs éléments peuvent être des atomes ou d'autres séquences.

réfléxivité : quand un langague peut se définir lui-même (langague fonctionnel)

Paradigme logique

À partir de clauses logiques un démonstrateur/moteur d'inférence répond à une requête/question.

méthode de résolution SLNDF-résolution. Selection, Linear, Definite, Negation as failure

- unification : unification d'un but avec la tête de clause
- négation par l'échec : ce qui n'est pas vrai est faux
- résolution suivant l'ordre des clauses (arbitraire) l'utilisateur doit faire attention à leur ordre

Programmation Java Swing

Tout hérite de JComponent (qui hérite de AWT mais osef)

Arbre d'instanciation : superposition (enfants devant), clipping(dépasse pas)

"père" : JFrame ou JApplet, `add(enfant)`

Ne pas oublier : `frame.pack()` (calcule position et taille) et `frame.setVisible(true)`

Listener : on peut créer une classe (1) exprès ou le conteneur peut directement implémenter `ActionListener` (2) (mieux si pas bcp de boutons). Must (3) : classes imbriquée le conteneur contient un listener

puis dans tous les cas : `button.addActionListener(listener)`