Génie logiciel
Rappels de programmation orientée objet

MIF17 : semestre d'automne 2015

Cours par Julien MILLE

Bibliographie

- Cours de MIF17 de Yannick Prié 2011-2012

  pagesperso.lina.univ-nantes.fr/~prie-y/ens/11-12/MIF17/

- Programmer en langage C++

  C. Delannoy, 6ème édition, Eyrolles, 2004

- Effective C++

  S. Meyers, 3rd Edition, Addison Wesley, 2005

- The Java Langage Specification, Java SE 7 Edition

  J. Gosling et al, Final Edition, Oracle, 2011

  http://docs.oracle.com/javase/specs/

- developpez.com, siteduzero.com, ...

- Langages orientés objet
  - Ada, C++, C#, D, Eiffel, Java, Objective C, Perl, Python, PHP, Ruby, SmallTalk, …
  - Dans ce cours, exemples en C++ et Java

- C++ et Java
  - Similaires du point de vue de la syntaxe et des mécanismes liés à l'objet
  - Différents du point de vue de la philosophie et de l'environnement d'exécution

- Philosophie
  - C++
    - Proche du matériel : manipulation explicite des adresses, de la pile, du tas, …
    - Gestion manuelle de la mémoire
    - Bibliothèque standard (STL) limitée aux entrées/sorties (ifstream, ofstream, cout, cin, …), aux conteneurs génériques (list, vector, set, map, …) et algorithmes associés
    - Le développeur doit souvent utiliser des bibliothèques externes : multi-plateformes (Boost, Qt, wxWidgets, OpenCV, …) ou dédiées à un système d'exploitation particulier (Windows SDK, Framework .NET, MFC, ...)

- Philosophie
  - Java
    - Mécanismes de sureté (le programmeur est "encadré") : gestion avancée des exceptions
    - Gestion « masquée » de la mémoire
    - Désallocation automatique via le garbage collector
    - Outil « prêt à l'emploi » : de nombreuses bibliothèques (packages) standards pour la gestion des évènements, de l'interface graphique, des interactions avec les bases données, des sockets, …

- Des langages qui évoluent
  - C++
    - Première édition publiée en 1985
    - Normalisé par l'ISO depuis 1998
    - Mise à jour mineure en 2014 (norme C++14) et une prévue en 2017
  - Java
    - Première version développée par Sun en 1996
    - Java 2 (déclinaison en deux plateformes : Java SE et Java EE) en 1998
    - Dernière version (Java 7) en 2011

- Compilation / exécution
  - S'il ne fait aucun doute que C++ est un langage compilé, on peut lire sur Internet ou autres sources que Java est un langage interprété : oui… mais non

– C++ : « Write Once, Compile Anywhere »
  - Code source portable (s'il n'utilise que des bibliothèques portables)
  - Compilation ⟶ code objet (langage machine temporaire, dépendant de l'architecture)
  - Edition des liens ⟶ code exécutable (langage machine, dépendant de l'architecture) interprété nativement par le processeur

- Java : « Write Once, Run Anywhere »
  - Code source (portable)
  - Compilation (**`javac`**) ⟶ byte-code (portable)
  - Interprétation du byte-code par la machine virtuelle Java (JVM), composante de l'environnement d'exécution Java (JRE)
    - La machine virtuelle Java, elle, est dépendante de l'architecture
    - Compilation « à la volée » (JIT, Just In Time), durant l'exécution, du byte-code en langage machine
      - » Sur PC et Mac, cette compilation est logicielle
      - » Sur certains systèmes embarqués, cette compilation est directement codée en hardware

- Notion d'objet
  - Une entité ayant une existence propre, regroupant des caractéristiques, sur laquelle on peut effectuer des opérations (qui vont consulter ou modifier son état)
  - Peut correspondre à une entité "concrète" (animal, véhicule, …) ou "abstraite", "conceptuelle" (compte utilisateur, unité d'enseignement, ...)

    *(attention, on ne parle pas ici de **classe abstraite,** concept de POO rappelé plus loin)*

- **Membres** d'un objet
  - **Attributs** (champs) : caractéristiques pouvant être consultées ou modifiées
  - **Méthodes** (fonctions membre) : opérations sur l'objet
- Classe
  - Regroupement d'objets ayant le même comportement
  - Abstraction décrivant les propriétés communes des objets qui en sont des **instances**
    - **Attention à la distinction classe/objet**. Exemple : si l'on modélisait les enseignements, les entités "Algorithmique" et "Réseau" ne seraient pas des classes, mais des instances d'une même classe "UE"
  - Une classe peut décrire une infinité d'instances

- Implémentation d'une classe : Java

Student.java

```
...
public class Student {
    // Attributs
    private int number;
    private String surname;
    ...

    // Méthodes
    public Student() {
        number = 0;
        surname = "NEW STUDENT";
    }
    public void setNumber(int n) {
        number = n;
    }
    public int getNumber() {
        return number;
    }
    ...
};
```

- Implémentation d'une classe : Java (remarques)
  - Mode d'accès (**`public, private, protected`**), également appelé **visibilité**, spécifié avant chaque attributs ou méthodes

    *(ces modes d'accès sont détaillés plus loin, dans la section sur l'**encapsulation**)*
  - Définition des méthodes (écriture du corps des fonctions) dans la définition de la classe, dans un unique fichier .java
  - Obligatoirement un source .java par classe publique

- Implémentation d'une classe : C++

Student.hpp / .h

```
...
class Student
{
  // Attributs
  private:
    int number;
    std::string surname;
    ...
  // Méthodes
  public:
    Student() {
        number = 0;
        surname = "NEW STUDENT";
    }
    void SetNumber(int n);
    int GetNumber() const;
    ...
};
...
```

- Implémentation d'une classe : C++

Student.cpp

```
#include "Student.h"

void Student::SetNumber(int n)
{
    number = n;
}

int Student::GetNumber() const
{
    return number;
}

...
```

- Implémentation d'une classe : C++ (remarques)
  - Mode d'accès (**`public, private, protected`**) spécifié "par lot"
  - Définition des méthodes (écriture du corps des fonctions)
    - Dans la définition de classe (fichier d'en-tête .h/.hpp)

      -> méthodes inline (plutôt pour les petites fonctions)
    - En dehors de la définition de la classe (fichier .cpp)

      -> le plus courant (dans ce cas, la méthode est juste **déclarée** dans la classe)
  - Autant de classes que l'on veut par fichier .h/.hpp

    (tout en respectant les règles de conception modulaire !)
  - En mémoire, lors de l'exécution, une classe est identique à une structure

- Objet courant : **`this`**
  - Accès à l'objet courant au sein d'une méthode

Java :

```
...
public Student() {
    this.number = 0;
    this.surname = "NEW STUDENT";
}
...
```

C++ :

```
...
Student() {
    this->number = 0;
    this->surname = "NEW STUDENT";
}
...
```

  - Peut être omis la plupart du temps
  - Utile pour lever l'ambigüité, lorsqu'un paramètre porte le même nom qu'un attribut

- Instanciation
  - Une variable de type Student **allouée** = une instance de la classe Student

Java :

```
Student st1 = new Student(); // Création d'une
  nouvelle instance
Student st2 = st1; // Définition d'une référence
  sur une instance déjà existante
```

C++ :

```
Student st1; // Création d'une nouvelle instance
  (dans la pile)
Student *pSt1 = &st1; // Définition d'un pointeur
  sur une instance existante
Student &st1ref = st1; // Définition d'une
  référence sur une instance existante
Student *pSt2 = new Student(); // Création d'une
  nouvelle instance (dans le tas), via un pointeur
...
delete pSt2; // Désallocation
```

- Appel de méthode sur une instance
  - Semblable à un accès à un champ de structure

Java :

```
Student st1 = new Student();
...
st1.setNumber(20121080);
```

C++ :

```
Student st1;
Student *pSt2 = new Student();
...
st1.setNumber(20121080);
pSt2->setNumber(20121081);
```

- Constructeur et destructeur
  - Constructeur : appelé lors de la création d'une nouvelle instance
    - Constructeur par défaut : pas de paramètre
      - Lorsqu'il y a héritage, le constructeur par défaut de la classe mère est hérité automatiquement (voir héritage, diapo 57)
    - Constructeur de recopie : une référence de la classe passée en paramètre
    - Autres constructeurs : paramètres libres

Java :

```
...
public class Student {
    ...
    // Constructeur par défaut
    public Student() {
        number = 0;
        surname = "NEW STUDENT";
    }
```

```
    ...
    // Constructeur de recopie
    public Student(Student st) {
        number = st.number;
        surname = new String(st.surname);
    }

    // Autres constructeurs
    public Student(int n) {
        number = n;
        surname = "NEW STUDENT";
    }
    public Student(int n, String s) {
        number = n;
        surname = new String(s);
    }
    ...
}
```

– Recopie des attributs :
  - pour les types simples (int, float, ...), l'opérateur = suffit
  - pour les objets, appel au constructeur de recopie de la classe correspondante

C++ :

```
...
class Student
{
    ...

    // Constructeur par défaut
    Student()
    {
        number = 0;
        surname = "NEW STUDENT";
    }

    // Constructeur de recopie
    Student(const Student &st)
    {
        number = st.number;
        surname = st.surname;
    }
```

```
    // Autres constructeurs
    Student(int n)
    {
        number = n;
        surname = "NEW STUDENT";
    }
    Student(int n, const std::string &s)
    {
        number = n;
        surname = s;
    }
    ...
};
```

– Passage de paramètre :
  - Utilisation de la référence constante (pour les objets, les structures) dès que possible

– Recopie des attributs :
  - pour les types simples (int, float, ...), l'opérateur = suffit
  - pour les objets, l'opérateur = peut être utilisé seul **s'il a été surchargé** (c'est le cas pour la classe std::string)

- Petite parenthèse : surcharge d'opérateur (C++)
  - Possibilité de redéfinir le rôle des opérateurs (=, ==, !=, [], +, -, etc) sur une classe donnée
  - Cas étudié ici : surcharge de l'opérateur d'affectation = (rôle proche du constructeur de recopie)

```
class Student
{
    ...
    Student &operator =(const Student &st)
    {
        number = st.number;
        surname = st.surname;
        return *this;
    }
    ...
};
```

- Petite parenthèse : surcharge d'opérateur (C++)
  - Très utile lorsque la classe comporte des pointeurs (utilisés comme liens vers d'autres structures ou comme tableaux)
  - Exemple : tableau dynamique

```
class DynArrayReal
{
  private:
    int size, size_max;
    float *array;

  public:
    DynArrayReal()
    {
        size = 0;
        size_max = 0;
        array = NULL;
    }
```

```
    DynArrayReal(const DynArray &ar)
    {
        size = ar.size;
        size_max = ar.size_max;
        array = new float[size_max];
        for (int i=0; i<size; i++)
            array[i] = ar.array[i];
    }

    DynArrayReal &operator =(const DynArray &ar)
    {
        if (size_max!=0)
            delete []array;

        size = ar.size;
        size_max = ar.size_max;
        array = new float[size_max];
        for (int i=0; i<size; i++)
            array[i] = ar.array[i];
        return *this;
    }
    ...
};
```

- Destructeur : exécuté lors de la suppression de l'objet
  - En Java, appelé automatiquement par le ramasse-miettes (*garbage collector*) lorsque l'objet n'est plus référencé
  - En C++, appelé automatiquement ...
    - ... en fin de fonction, pour les instances déclarées sur la pile, ou dans le destructeur de la classe contenante (exemple : le destructeur de la classe **`Student`** appelle le destructeur de **`std::string`** sur **`surname`**)
    - ... lors de la désallocation avec **`delete`**, pour les instances précédemment allouées avec **`new`**
- Que met-on dedans ?
  - Fermeture des flux de données (fichiers, streams, ...) pouvant avoir été ouverts auparavant
  - **`delete`** sur les attributs alloués avec **`new`** (C++ uniquement)

– En Java, pas de destructeur proprement dit, mais possibilité d'ajouter une méthode **`finalize()`**

```
public class Student {
    ...
    public finalize() {
        number = 0;
        surname = "DEAD STUDENT";
    }
    ...
};
```

– En C++, "~" suivi du nom de la classe

```
class Student
{
    ...
    ~Student()
    {
        number = 0;
        surname = "DEAD STUDENT";
    }
    ...
};
```

– Peu utile dans l'exemple précédent, mais obligatoire dans l'exemple du tableau dynamique :

```
class DynArrayReal
{
    ...
  public:
    ...
    ~DynArrayReal()
    {
        if (size_max!=0)
            delete []array;
    }
    ...
};
```

- Encapsulation
  - Concept fondamental : protection de l'information contenue dans un objet
    - Garantit l'intégrité de l'objet (exemple : évite les affectations par erreur, = au lieu de == !)
    - Garantit la cohérence éventuelle entre plusieurs attributs inter-dépendants (exemple : un tableau dynamique et sa taille)
    - Masque en partie le fonctionnement interne d'une classe -> respecte la notion de Type Abstrait de Donnée (facilite la prise en main du code dans un premier temps)
    - Attention : protège vis à vis du programmeur seulement ! (n'empêche pas les erreurs mémoire à l'exécution)

- Notion de visibilité
  - L'accès aux membres de la classe est différent selon que l'on se trouve à l'**intérieur** ou à l'**extérieur** de la classe
  - A l'intérieur de la classe (définition de la classe + définition des méthodes dans un .cpp), tout est permis
  - A l'extérieur de la classe, le programmeur ne "voit" que ce que la classe "veut bien lui montrer"
- Java et C++ implémentent trois niveaux de "visibilité" pour les membres (attributs et méthodes)
  - **`public`** : les membres publics sont accessibles à tous
  - **`protected`** : les membres protégés sont accessibles seulement aux classes dérivées (voir héritage)
  - **`private`** : les membres privés sont accessibles seulement au sein de la classe elle-même

– Encapsulation en pratique : illustration de l'intérêt de la classe par rapport à une structure

C++ :

```
class DynArrayReal
{
  private:
    int size, size_max;
    float *array;

  public:
    DynArrayReal();
    DynArrayReal(const DynArrayReal &);
    ~DynArrayReal();
    void AddNumber(const float);
    ...
};
```

– On suppose que **`AddNumber()`** redimensionne **`array`** si besoin

Equivalent C :

```
typedef struct {
    int size, size_max;
    float *array;
} DynArrayReal;

void initDynArrayReal(DynArrayReal *);
void initCopyDynArrayReal(DynArrayReal *, const
  DynArrayReal *);
void freeDynArrayReal(DynArrayReal *);
void addNumberDynArrayReal(DynArrayReal *, const
  float);
...
```

– Utilisation correcte du tableau dynamique en C …

```
void main()
{
    DynArrayReal ar;
    initDynArrayReal(&ar);
    addNumberDynArrayReal(&ar, 3.0);
    ...
    freeDynArrayReal(&ar);
}
```

– ... mais dans l'absolu, rien n'empêche un programmeur peu averti de compiler et d'exécuter :

```
void main()
{
    DynArrayReal ar;

    ar.size = 1;
    ar.array[0] = 3.0; // Erreur à l'exécution
}
```

– Dans la version objet C++, les erreurs sont détectées lors de la compilation :

```
void main()
{
    DynArrayReal ar;    // Appel au constructeur
                        // par défaut
    ar.size = 1;        // Erreur : accès à un
                        // membre privé
    ar.array[0] = 3.0;  // Idem
}
```

– Utilisation correcte du tableau dynamique en C++ : à l'extérieur de la classe, accès uniquement aux membres publics :

```
void main()
{
    DynArrayReal ar;    // Appel automatique au
                        // constructeur par défaut
    ar.AddNumber(3.0);
    ...

    ...                 // Appel automatique au
                        // destructeur
}
```

– Même principe en Java : erreurs d'accès aux membres privés détectées lors de la compilation

– Comportement similaire dans une autre classe contenant une ou plusieurs instance(s) de la classe

Java

```
public class StudentPair {
    private Student st1, st2;

    ...
    public String getSurnames() {
        String s;
        s = st1.surname + " " + st2.surname;
    }
    // Erreur : surname est privé dans Student

    ...
};
```

C++

```
...
class StudentPair
{
  private:
     Student st1, st2;
     ...

  public:
    ...
    std::string getSurnames() const
    {
        std::string s;
        s += st1.surname;
        s += " ";
        s += st2.surname;
    }
    // Erreur : surname est privé dans Student
};
```

– Attention : l'erreur n'a rien à voir avec le fait que **`st1`** et **`st2`** soient privés dans **`StudentPair`**

- Accesseurs et mutateurs
  - Accesseurs
    - Méthodes **`get...()`** : lecture d'un attribut
    - Objet non modifé
    - Retour = une copie (valeur) de l'attribut
  - Mutateurs
    - Méthodes **`set...()`** : écriture d'un attribut (et éventuellement, d'attributs dépendants)
    - Objet modifié
    - Type de retour = void ou un type indiquant un statut (modification effectuée ou non, ...)
  - Si possible, un couple accesseur/mutateur par attribut

- Exemple avec un attribut chaîne de caractère :

Java

```
...
public class Student {
    ...
    private String surname;
    ...
    public String getSurname() {
        return surname;
    }
    public void setSurname(String s) {
        surname = new String(s);
    }
    ...
};
```

- Exemple avec un attribut chaîne de caractère :

C++

```
...
class Student
{
  private:
    ...
    std::string surname;
    ...

  public:
    void SetSurname(const std::string &s)
    {
        surname = s;
    }
    std::string GetSurname() const
    {
        return surname;
    }
    ...
};
```

- Membre statique
  - Un membre statique est partagé par toutes les instances de la classe
  - Utile pour implémenter une propriété (variable ou constante) commune à toutes les entités d'une même classe
  - Similaire à une variable ou fonction globale, mais interne à une classe
  - Accès possible avec ou sans instance de la classe
  - Exemple : tout étudiant doit obtenir 180 crédits ECTS pour valider une licence

– En Java, initialisation possible lors de la définition de la classe

Student.java

```
public class Student {
    ...
    public static int required_credits = 180;
    ...
};
```

Ailleurs.java

```
...
Student st1 = new Student();
Student st2 = new Student();

System.out.println(Student.required_credits); // 180
st1.required_credits = 100; // à éviter
System.out.println(st2.required_credits); // 100
Student.required_credits = 180;
...
```

– En C++, initialisation obligatoirement en dehors de la classe

Student.hpp

```
class Student
{
    ...

  public:
    static int required_credits;

    ...
};
```

Student.cpp

```
#include "Student.hpp"

int Student::required_credits = 180;
...
```

Ailleurs.cpp

```
...
Student st1, st2;

cout<<Student::required_credits<<endl; // 180

st1.required_credits = 100; // à éviter

cout<<st2.required_credits<<endl; // 100

Student::required_credits = 180;
...
```

– En dehors de la classe, éviter d'accéder à un membre statique via une instance (confusion avec membre non statique)
  – préférer l'accès via la classe elle-même

- Héritage
  - Concept fondamental : réutilisation du code en faisant dériver une nouvelle classe d'une classe existante, afin de récupérer ses attributs et méthodes, sans avoir à la réécrire complètement

  - Terminologie : si une classe **`Child`** hérite d'une classe **`Parent`** :
    - **`Child`** est une **spécialisation**, une **sous-classe**, une **classe fille**, une **classe dérivée** de **`Parent`**
    - **`Parent`** est une **généralisation**, une **super-classe**, une **classe mère** de **`Child`**

- Quand utiliser un héritage ?
  - Création d'une spécialisation d'une entité
  - L'entité dérivée (classe fille) possède **toutes** les caractéristiques et opérations de l'entité initiale (classe mère) avec, éventuellement, des caractéristiques et opérations supplémentaires
  - Classe dérivée plus "précise" que la classe mère. Exemples :
    - **`Châtaigner`**, **`Chêne`** peuvent hériter de **`Arbre`**
    - **`Voiture`**, **`Bicyclette`** peuvent hériter de **`Véhicule`**
  - Possibilité de modifier les opérations -> redéfinition de méthode
- Transitivité de la relation
  - Si B hérite de A et C hérite de B, alors indirectement, C hérite de A
  - Possibilité d'enchaîner les héritages

– Héritage simple public

- Conserve la visibilité : si B hérite de A, et que le membre i est public (resp. privé, protégé) dans A, alors il est public (resp. privé, protégé) dans B
- Un membre privé sera inaccessible même dans les classes dérivées

A.java

```
public class A {
    private int i;
    protected int j;
    public int k;

    public A() {
        i = 0;
        j = 0;
        k = 0;
    }
};
```

- En Java, par défaut, toute classe hérite de **`Object`**

B.java

```
public class B extends A {
    public B() {
        i = 0; // Erreur : i est privé dans A
        j = 0; // Ok : j est protégé dans A
        k = 0; // Ok : k est public
    }
};
```

Ailleurs.java

```
public class Ailleurs {
    private A a;
    private B b; // a et b alloués dans le constructeur
    ...
    public void setAllToOne() {
        a.i = 1; // Erreur : i est privé dans A
        a.j = 1; // Erreur : j est protégé dans A
        a.k = 1; // Ok : k est public dans A
        b.i = 1; // Erreur : i est inaccessible dans B
        b.j = 1; // Erreur : j est protégé dans B
        b.k = 1; // Ok : k est public dans B
    }
};
```

A.hpp

```
class A
{
  private:
    int i;

  protected:
    int j;

  public:
    int k;
    A()
    {
        i = 0;
        j = 0;
        k = 0;
    }
};
```

B.hpp

```
#include "A.hpp"
class B: public A
{
  public:
    B()
    {
        i = 0; // Erreur : i est privé dans A
        j = 0; // Ok : j est protégé dans A
        k = 0; // Ok : k est public
    }
};
```

Ailleurs.hpp

```
#include "B.hpp"
class Ailleurs
{
  private:
    A a;
    B b;

  public:
    ...
    void setAllToOne()
    {
        a.i = 1; // Erreur : i est privé dans A
        a.j = 1; // Erreur : j est protégé dans A
        a.k = 1; // Ok : k est public dans A
        b.i = 1; // Erreur : i est inaccessible dans B
        b.j = 1; // Erreur : j est protégé dans B
        b.k = 1; // Ok : k est public dans B
   }
};
```

– En C++, l'héritage lui-même peut également être protégé ou privé (pas d'équivalent Java, donc non détaillé ici)

– Redéfinition

- Par défaut, la classe fille a le même "comportement" que la classe mère.
- Intérêt : pouvoir changer partiellement ce comportement en **redéfinissant** certaines méthodes (en conservant le type de retour et les paramètres)
- Une méthode redéfinie peut être complètement différente de la méthode de base, ou bien réutiliser celle-ci en effectuant des opérations supplémentaires
- Exemple : *l'employé travaille. En plus de l'employé standard, on distingue deux sortes d'employé supplémentaires : le fainéant qui, lorsqu'il travaille, fait en réalité autre chose et le bourreau de travail qui, en plus de son travail obligatoire, réalise des tâches supplémentaires.*

  Modélisons ces employés...

Worker.java

```
public class Worker {
    private ...; // Attributs de l'employé
    private ...;

    public Worker() {...}

    public void work() {
        ...
    }
    ...
};
```

LazyWorker.java

```
public class LazyWorker extends Worker {
    private ...; // Attributs supplémentaires
    private ...; // propres au fainéant

    public LazyWorker() {...}

    public void work() {
        ... // Rien... ou autre chose que le travail
    }
    ...
};
```

– En Java, accès aux membres de la classe mère (s'ils sont protégés ou publics) avec le mot-clé **`super`**

Workaholic.java

```
public class Workaholic extends Worker {
    private ...; // Attributs supplémentaires
    private ...; // propres au bourreau de travail

    // Méthodes
    public Workaholic() {...}

    public void work() {
        super.work();  // Appel à work() de Worker
        ...            // Travail supplémentaire !
    }
    ...
};
```

Worker.hpp

```
class Worker
{
  private:
    ... // Attributs de l'employé
  public:
    Worker() {...}
    void work() {...}
    ...
};
```

LazyWorker.hpp

```
#include "Worker.hpp"
class LazyWorker : public Worker
{
  private:
    ... // Attributs supplémentaires propres au fainéant
  public:
    LazyWorker() {...}

    void work()
    {
        ... // Rien... ou autre chose que le travail
    }
    ...
};
```

– En C++, accès aux membres de la classe mère (s'ils sont protégés ou publics) avec "::" précédé du nom de la classe mère

Workaholic.hpp

```
#include "Worker.hpp"
class Workaholic : public Worker
{
  private:
    ... // Attributs supplémentaires propres
        // bourreau de travail
  public:
    Workaholic() {...}

    void work()
    {
        Worker::work(); // Appel à work() de Worker
        ...             // Travail supplémentaire !
    }
    ...
};
```

– Une référence (ou un pointeur C++) sur la classe mère peut être alloué(e) en une instance de cette classe mère, mais aussi en une instance de classe fille

– Permet de choisir le "type d'instance" **à l'exécution**

Java:

```
Worker w;
...
if (...)
    w = new LazyWorker();
else if (...)
    w = new Workaholic();
else
    w = new Worker();
...
w.work();
```

C++:

```
Worker *pW;
...
if (...)
    pW = new LazyWorker();
else if (...)
    pW = new Workaholic();
else
    pW = new Worker();
...
pW->work();
```

- Héritage du constructeur par défaut
- EN JAVA COMME EN C++
  - En Java, L'appel à super() au début du constructeur par défaut de la classe fille est implicite !
- A COMPLETER

– Méthodes virtuelles, polymorphisme

- Une méthode est dite **virtuelle** si, lorsqu'elle est redéfinie dans une classe dérivée, le type d'instance (classe fille ou mère) détermine la méthode réelle à appeler lors de l'exécution (permet le **polymorphisme**)
  - En Java, les méthodes sont virtuelles par défaut. Dans l'exemple précédent, `w.work()` entraîne un appel à la méthode de la classe réellement **instanciée avec** `new` (`Worker`, `LazyWorker`, `Workaholic`), même si `w` a été **déclaré** de type `Worker`.
  - En C++, les méthodes ne sont pas virtuelles par défaut. Dans l'exemple précédent, `pW->work()` appelle toujours la méthode de `Worker` (car `pW` est un pointeur sur `Worker`). Pour remédier à ce problème, la méthode doit être déclarée `virtual`.

Worker.hpp

```
class Worker
{
    ...
    virtual void work() {...}
    ...
};
```

– Méthode virtuelle pure, classe abstraite

- Une méthode **virtuelle pure** est déclarée sans être définie, et est donc destinée à être obligatoirement (re)définie dans une classe dérivée
- Une classe est dite **abstraite** si elle contient au moins une méthode virtuelle pure
- Une classe abstraite ne peut pas être instanciée, et est donc destinée à être dérivée (héritage obligatoire). Elle fournit un "modèle de base".
- Modification de l'exemple précédent : admettons qu'il n'existe plus de "travail standard".

  ⟶ la méthode **`work()`** de la classe mère devient virtuelle pure et **`Worker`** devient donc abstraite (mais déclaration de référence ou de pointeur sur **`Worker`** toujours possible)

– Méthode virtuelle pure, classe abstraite

- En Java, utilisation du modificateur abstract **sur la méthode et la classe**

```
public abstract class Worker {
    private ...; // Attributs de l'employé
    private ...;

    public Worker() {...}
    public abstract void work();
    ...
};
```

- En C++, ajout de "=0" à la déclaration de la méthode

```
class Worker
{
  private:
    ... // Attributs de l'employé
  public:
    Worker() {...}
    virtual void work()=0;
    ...
};
```

– Héritage multiple, interface

- Héritage multiple autorisé en C++, pour bénéficier des opérations de plusieurs classes mères
- Exemple : un véhicule amphibie est à la fois terrestre et aquatique

Terrestrial.hpp

```
class Terrestrial
{
      ...
   public:
      void move();
      ...
};
```

Aquatic.hpp

```
class Aquatic
{
      ...
   public:
      void swim();
      ...
};
```

Amphibian.hpp

```
class Amphibian : public Terrestrial, public Aquatic
{
    ...
};
```

- **`Amphibian`** dispose à la fois des méthodes **`move()`** et **`swim()`**, qu'elle peut redéfinir

- Héritage multiple impossible en Java, le problème est contourné par l'utilisation d'**interfaces**
- Différence : les interfaces sont **abstraites** (concept plus restrictif qu'une classe abstraite : les méthodes d'une interface sont **toutes** virtuelles)

Terrestrial.java

```
public interface Terrestrial {
      ...
   public void move();
      ...
};
```

Aquatic.java

```
public interface Aquatic {
      ...
   public void swim();
      ...
};
```

- Une classe peut **implémenter** plusieurs interfaces :

Amphibian.java

```
public class Amphibian implements
    Terrestrial, Aquatic {

    public void move() {...}
    public void swim() {...}
};
```

- **`Amphibian`** **doit** définir les méthodes **`move()`** et **`swim()`**

- **Sous-interface** : à la différence d'une classe, une interface peut **hériter** de plusieurs interfaces.