Cours 4TYE707U
Programmation orientée objet - Master Miage semestre 7

Lionel Clément

2019-2020

Table des matières

Informations administratives

- Équipe pédagogique :
  - CLÉMENT Lionel
  - COUNILH Marie-Christine
- Code UE : 4TYE707U
- Cours : les lundi matin de 10h15 à 12h15 au début et 11h00 à 13h00 à la fin
- 2 groupes de TDs : vendredi 8h00 – 10h00, vendredi 10h00 – 12h00
- Quantité de travail :
  - Assiduité et contrôle continu obligatoire
  - Heures d'enseignement dispensées à l'étudiant : 58 heures
  - Temps de travail personnel : 102 heures
- Contrôle des connaissances
  - Note éliminatoire : inférieure à 7/20
  - Session 1 :
    - Examen terminal (2/3) : 3 heures
    - Contrôle continu (1/3) : 1 DS, 1 TP noté
  - Session 2 :
    - Examen écrit (3 heures) ou oral selon l'effectif (2/3)
    - Contrôle continu (1/3) : report de la session 1

Prérequis

Connaissance de la programmation impérative. Première utilisation d'un langage de programmation à objet (Java, C++, C#, Python, Php, Javascript, etc).

Objectifs

Cette UE a pour objectif d'enseigner les concepts avancés de la programmation objet (affectation, conception interne, héritage, délégation, typage).

L'objectif est aussi de couvrir les aspects de programmation par *thread*, par événement, et les connexions aux bases de données relationnelles.

Comment réussit-on une UE « Programmation Objet » ?

- Savoir académique
  - Bien comprendre et apprendre les concepts
  - Savoir reproduire soi-même l'argumentaire développé sur chaque partie du cours
  - Attention : assister à des exposés sur *Youtube* ou autre est utile et souhaitable, mais clairement insuffisant pour savoir le cours
- Savoir-faire
  - Répéter inlassablement les exercices soi-même jusqu'à savoir programmer vite et bien et en toute indépendance les programmes informatiques demandés en exercice
  - Ne pas se contenter du temps passé en cours et en TD, mais passer du temps à la programmation chez soi ou en salle machine

Références bibliographiques

Documentation

— La documentation en ligne sur Java fournie par Oracle
`https://docs.oracle.com/en/java/javase/12/`
— Les tutoriels :
`https://docs.oracle.com/javase/tutorial/`
— Le cours "Approche Objet" de Xavier Blanc en Master Informatique 1ère année (vidéos et TD)
`http://www.labri.fr/perso/xblanc/teaching.html`
— Livres :
  — Le cours de Java d'Irène Charon (ENST Paris) Irène Charon, "Le langage Java : concepts et pratique - le JDK 5.0", 3e édition, Éditions Hermès, 2006
  — Sur les classes génériques et le paquetage Collection Maurice Naftalin et Philip Wadler, "Java Generics and Collections", O'Reilly 2006
  — Le livre de référence sur les modèles de conception E. Gamma, R. Helm, R. Johnson et J. Vlissides, "Design Patterns. Catalogue de modèles de conception réutilisables", Vuibert, 1999

1 Introduction

Un programme informatique développé avec un langage de programmation impératif récent est modulaire et structuré. Cela suffit-il pour envisager de l'utiliser sur de très gros projets ?

Quels sont les problèmes qu'on rencontre avec un logiciel développé sans l'approche objet ?

Prenons le cas du développement d'un site de vente en ligne :

— Peut-on organiser le développement de l'ensemble du programme par une équipe en partageant des tâches ?
Exemple : Celui qui développe la gestion du catalogue peut-il le faire indépendamment de celui qui gère la mise en œuvre de la présentation graphique et qui clairement n'a pas les mêmes préoccupations ?

— Pourra-t-on reprendre une partie du code dans le cas d'une refonte importante ?
Exemple : La demande est de porter le code d'un site de commerce en ligne existant vers une application *Android* pour téléphone portable. Le code est-il assez modulaire et redistribuable pour être utilisé dans ce nouveau contexte ? Doit-on tout reprendre ? Sait-on quantifier le travail à réaliser ?

— Peut-on distribuer le code facilement ?
Exemple : Le code est documenté et confié en logiciel libre à la communauté. Est-il simple de faire des modifications, des ajouts pour améliorer le code en s'assurant qu'il est toujours exempt de bugs ? Peut-on exploiter les contributions des développeurs qui interviennent sur le logiciel libre ?

— Est-on certain de ne pas avoir développé plusieurs fois le même code ?
Exemple : La gestion du calcul du devis, de la facture et de la commande semble commune quand il s'agit d'appliquer des politiques de promotion à ces trois aspects différents de la facturation. A-t-on développé plusieurs fois la même chose ? Peut-on réexploiter ce qui a été fait pour les trois cas ?

— Peut-on tester tout le code ?
Exemple : Est-on certain d'avoir testé l'ensemble des possibilités offertes au client final qui doit valider un panier, confirmer une commande ou demander un devis ? Sait-on s'il existe des failles ? Peut-on les quantifier ?

Quiconque a développé un projet un peu ambitieux avec un langage de programmation non objet, sait qu'il est difficile, voire impossible de répondre favorablement à ces différentes questions.

Le but de la programmation objet est d'organiser un programme sous forme d'objets, et de structurer ces objets pour :

— Développer des éléments isolés du projet global
— Réutiliser le même code pour plusieurs objets différents
— Abstraire la partie technique de l'implémentation
— Organiser le code en rangeant les objets par classes
— Tester les différentes parties du code indépendamment du reste

2 Un peu d'histoire de la programmation objet

— Algol 58, 60, 68 (1958-1975) – Pascal (1971)
Dans les années 1960, la programmation connait une période où l'on structure les programmes par boucles, tests et procédures. En plus de l'inscription des programmes dans des

systèmes d'exploitation qui assurent depuis peu une gestion des processus et des fichiers, c'est une avancée considérable pour l'informatique encore juvénile.
Avant cela, un code revenait à un organigramme difficile à segmenter et à modulariser. Techniquement les boucles et les tests étaient implémentés par des sauts dans le code, on était encore très près de la programmation câblée des ordinateurs et l'informatique était une discipline de l'électronique dans nos Universités.
Enfin, avec l'arrivée des langages de programmation structurée, les programmes pouvaient s'organiser structurellement selon ces trois éléments de programmation fondamentaux : boucles, tests et procédures.

— Simula-1 (1962) – Simula (1967)
A la fin des années 60, on voit émerger le premier langage orienté objet :
  — Simula-1 (1962) est destiné aux problèmes de simulation. C'est le premier langage qui regroupe les données et les procédures sous une même entité en brisant un tabou. Pour information, Simula-1 est un sur-ensemble d'Algol qui intègre un mécanisme à événements discrets pour formaliser les éléments de simulation.
  — Simula (1967) est le premier véritable langage de programmation orienté objet.
  Un programme Simula est un ensemble d'objets autonomes et qui disposent de leurs propres données et procédures. Ceci permet de simuler des comportements parallèles.
  Simula intègre la notion d'héritage. Les notions d'encapsulation et l'abstraction des données réhabilite la fusion des données et des programmes. Contredisant un principe fort de la programmation structurée des années 70-80. Ada et CLU sont issues de cette école.
  Simula a servi comme modèle pour un ensemble de langages de programmation à typage statique dite de *l'école scandinave* (C++, Clascal, Object Pascal, Eiffel, Beta).

— Smalltalk-72, Smalltalk-76, Smalltalk-80
Smalltalk généralise la notion d'objet qui devient le seul élément de programmation (y compris au niveau de la structure des programmes). L'envoi de message est le seul moyen qui permet aux objets de communiquer entre eux. Smalltalk-76 introduit une relation d'héritage entre classes. Smalltalk-80 est de typage dynamique, il introduit la notion de métaclasse (classe dont les instances sont des classes).

— Flavors, Ceyx, Clos
Ces langages forment un mariage entre le langage de programmation Lisp et le paradigme objet.

— Java (1995)
Développé cher Sun, acheté par Oracle en 2009

— Javascript (1995)

— Objective Caml (1996)
Le langage de programmation fonctionnel Caml se dote de systèmes de types de d'inférence de types pour la programmation par objets.

3 Objet

En informatique, un objet est une entité qui va contenir une information qui lui est propre. Cet objet permet de définir un élément du programme et son comportement singulier sans se soucier du reste du logiciel supposé très complexe.

La programmation par objets consiste à distinguer ces entités du programme qui vont entrer dans un système complexe (au sens de compositionnel). Les mécanismes par lesquels les objets se composent et entrent en relation les uns avec les autres est donc crucial. Par exemple, le panier d'un client doit être en relation avec les produits dont il fait référence pour en connaître le prix, le poids, etc.

3.1 Trio (entité, attribut, valeur)

Il est d'usage de formaliser des données sous la forme d'un ensemble de tuples

$$\{(e_1, a_1, v_1), (e_2, a_2, v_2), \ldots, (e_k, a_k, v_k)\}$$

comme (stylo, couleur, rouge), (stylo, marque, "Waterman"), (client, nom, "Clément") etc. Les entités se caractérisent par un ensemble de propriétés. On dit qu'elles se définissent *en propre*.

L'attribut prend une valeur en fonction de son domaine de définition. Par exemple couleur peut prendre les valeurs *jaune, rouge, noir, etc.*, et seulement celles-ci. C'est le *type* de l'attribut.

Les objets seront stockés en mémoire sous la forme d'ensembles attribut/valeur. L'ensemble des objets d'un programme est stocké sous la forme d'une relation

$$\{(e_1, a_1, v_1), (e_2, a_2, v_2), \ldots, (e_k, a_k, v_k)\}$$

Notons qu'on type les attributs selon des types primitifs, mais aussi en référence à d'autres objets. C'est-à-dire qu'un attribut d'un objet peut avoir comme valeur un autre objet. C'est l'un des mécanismes utilisés pour composer l'ensemble des objets d'un programme.

Prenons un exemple : Le panier d'un client est un objet. Le client est lui-même un objet (du point de vue informatique, c'est une entité qui se définit par des caractéristiques propres). Un client peut donc contenir une information sur son panier sous la forme du tuple

$$(client_i, panier, panier_i)$$

où $client_i$ et $panier_i$ sont deux objets, le second composant le premier.

3.2 Envoi de message

Au lieu d'avoir une sorte de gros programme qui traite l'ensemble des objets un-à-un, comme nous le ferions en programmation impérative conventionnelle, le paradigme de la programmation objet utilise l'*envoi de message* aux objets pour opérer des calculs et des échanges.

La seule action réalisable entre deux objets revient à appeler une procédure déclarée dans un objet depuis un autre objet. Ceci s'appelle l'*envoi de message*.

Par exemple un `panier` contient en plus des attributs qui le définissent en propre, l'ensemble des procédures pour sa propre gestion (calcul du prix total, calcul du poids du panier pour une expédition par colis, ajout d'un produit, suppression d'un produit, modification de la quantité d'un produit, etc, etc).

Il suffit d'envoyer le message ≪ calcule le total ≫ à un objet de type `panier` pour que celui-ci s'exécute et calcule le total à afficher.

Un programme revient donc à une sorte de grosse fourmilière où chaque objet envoie des messages à d'autres pour faire fonctionner le système complet.

L'intérêt à faire fonctionner un système de la sorte est multiple :

— **Sécuriser le code**
Chaque objet reçoit un ensemble de traitements qui lui sont propres et pour lesquels il est possible de s'assurer de la **bonne formation** de l'objet et de ses attributs. Cela passe par un mécanisme de restriction sur lesquels nous reviendrons et qui s'appelle **l'encapsulation**.
— **Moduler le code**
Chaque objet étant traité individuellement, il est envisageable de segmenter le projet en micro-projets indépendants.
— **Tester le code**
Avec ces procédures limitées aux seuls objets, il est possible d'opérer des tests fiables et complets.

3.3 Un exemple en langage javascript

Listing 1 – Premier exemple de création d'objets

```
var catalog = {
    produits: [{nom: 'chaussettes',
                couleur: 'rose',
                quantite: 1,
                poids: 0.125},
               {nom: 'pantalon',
                couleur: 'bleu',
                quantite: 1,
                poids: 0.925},
               {nom: 'chemise',
                couleur: 'blanc',
                quantite: 1,
                poids: 0.625}]
};

var panier = {
    produits: [],
    ajouter: function(produit) {
        this.produits.push(produit);
    },
    calcul: function() {
        var poidsTotal = 0;
        for (var i=0 ; i< this.produits.length ; i++) {
            poidsTotal += this.produits[i].poids;
        }
        return poidsTotal;
    }
};


panier.ajouter(catalog.produits[0]);
panier.ajouter(catalog.produits[2]);

alert (panier.calcul());
```

3.4 Objets composites, objets agrégés, objets associés

Objets composites

Les objets peuvent entrer dans une relation de type composition, où les uns se trouvent contenus dans les autres et ne sont accessibles qu'à partir de ces autres.

Exemple : un catalogue contient l'ensemble des produits qui le composent. Il est donc envisageable de définir l'existence des occurrences des produits mis en vente directement dans le catalogue. Le catalogue est un objet qui compose un ensemble de produits, qui sont aussi des objets. Dans cette conception, un produit n'a pas de sens hors catalogue.

Techniquement, la création d'une nouvelle instance de `Produit` sera de la **responsabilité** de `Catalogue`.

Listing 2 – Composition Catalogue.java

```
public class Catalogue {

    private Produits produits;

    public Catalogue(){
        System.out.println("Catalogue");
        produits =  new Produits();
    }

    public ajouterProduit(String name, int quantity){
        this.produits.add( new Produit(name, quantity) );
    }
}
```

Objets agrégés

C'est une généralisation de la composition qui n'entraîne pas l'appartenance.

Exemple : un panier contient une liste de produits, mais une liste de produits n'est pas définitoire d'un panier. Cette liste est principalement contenue dans le catalogue. Il est possible de créer plusieurs paniers qui partagent les mêmes produits, il est aussi possible de détruire des paniers sans détruire les produits qu'ils contiennent.

Techniquement, la création d'une nouvelle instance de `Produit` ne sera pas de la **responsabilité** de `Panier` qui ne pourra que faire référence aux produits.

Listing 3 – Agrégation Panier.java

```
public class Panier {

    private Produits produits;

    public Panier(){
        System.out.println("Panier");
        produits =  new Produits();
```

```
    }

    public ajouterProduit(Catalogue catalogue, String name, int quantity){
        this.produits.add(catalogue.findProductByName(name), quantity);
        catalogue.sub(name, quantity);
    }
}
```

Objets associés

Un objet peut faire référence à un autre objet pour lui envoyer un message sans que celui-ci soit ni composé ni agrégé au premier.

Exemple : Le panier peut faire référence au catalogue pour savoir si un produit est encore disponible. Mais il n'est pas utile que l'objet `Catalogue` soit composé d'une manière ou d'une autre à l'objet `Panier`

Listing : Voir que l'objet `catalogue` est passé en argument au message `ajouterProduit()`

Terminologie

— Agrégation forte = Agrégation par valeur = Composition
— Agrégation
— Agrégation faible = Association

3.5 Un exemple commenté en C++

Listing 4 – Premier exemple en C++

```
#include <iostream>

class Produit
{
  char nom[32];

  Produit(){
    std::cerr << "Produit " << this << std::endl;
  }
  ~Produit(){
    std::cerr << "~Produit " << this << std::endl;
  }
};

class ListProduits
{
  Produit produits[5]; // creation automatique

  ListProduits(){
    std::cerr << "ListProduits " << this << std::endl;
  }
  ~ListProduits(){
    std::cerr << "~ListProduits " << this << std::endl;
  }
};

class Catalogue
{
  ListProduits *listProduits;

  Catalogue(){
    std::cerr << "Catalogue " << this << std::endl;
    this->listProduits = new ListProduits(); // creation dynamique
  }
  ~Catalogue(){
    delete this->listProduits;
    std::cerr << "~Catalogue " << this << std::endl;
  }
};

class Catalogue catalogue; // creation statique

int
main(int argn,
     char **argv)
{
}
```

4 Définition d'un objet

Il y a deux façons principales de définir un objet :

Soit il est défini sur le modèle d'un autre objet qui lui sert de prototype, soit il est défini par une classe d'objets.

4.1 Programmation par prototype d'objets

Les langages à *cadres* (*frame*) ou à *prototypes* sont inspirés des travaux de Marvin Minsky. Un *cadre* est un prototype d'objet standard qui a des attributs et un comportement qui sert de modèle pour d'autres objets qui en sont issus. On dit que l'objet est hérité d'un autre dont il est instance. Lui-même devient générateur d'autres objets en tant que prototype. Tout objet entre donc dans une hiérarchie dont le plus standard est `Object` : le cadre qui est censé avoir les attributs et les comportements communs à tous les objets du programme.

Le langage le plus connu par programmation par prototype d'objets est **Javascript**. Nous ne l'étudierons pas ici, au profit des langages par classes d'objets. Nottons que les versions récentes de `Javascript` ont intégré la définition d'objet par classe qui est la plus généralisée.

4.2 Programmation par classes d'objets

La classe correspond à un **type abstrait de données** qui décrit un ensemble d'objets partageant les mêmes propriétés et les mêmes traitements. Elle sert de définition pour produire des objets qui en sont des **instances**.

5 Classe

Une classe est un ensemble d'objets qui ont les mêmes types de propriétés et qui appliquent les mêmes traitements à leurs données.

La définition d'une classe est

— Le nom de cette classe

— Les propriétés décrites par des **attributs**, leur type et éventuellement leur valeur initiale.

— Les traitements décrits par des **méthodes**, leur type de retour et leurs arguments.

5.1 Instance d'un objet

Comment sont créés les objets ? La création d'un objet consiste à réserver une place mémoire de l'ordinateur pour y inscrire l'ensemble de ses propriétés. Tout comme la création d'une variable ordinaire, la création d'un objet peut être statique, automatique ou dynamique, c'est-à-dire qu'il est respectivement enregistré dans la mémoire statique, dans la pile ou dans le tas.

En Java, la création d'objets est dynamique à l'exception d'un objet **main** à la source de tous les autres objets et des parties statiques sur lesquelles nous reviendrons. D'une manière générale, en Java, c'est donc un objet qui est chargé de créer d'autres objets pendant l'exécution du programme.

La référence de l'objet ainsi créé porte le nom **this** dans son propre code.

Constructeur

Une ou plusieurs méthodes sont spéciales à la création de l'instance d'un objet. On les appelle **constructeurs**. En Java, elles n'ont pas de type de retour et portent le même nom que la classe. Comme toutes les autres méthodes, elles peuvent accepter des arguments et peuvent être surchargées (voir plus loin ce terme).

La méthode est exécutée juste après la création de l'instance. Cette instance dont on fait référence par le mot clef `this` est donc accessible pour attribuer ses propriétés.

Destructeur

Un destructeur est une méthode qui est exécutée juste avant la libération de la mémoire correspondant à l'instance d'un objet.

En programmation Java, il est d'usage et même recommandé de ne pas utiliser les destructeurs en laissant le ramasse-miettes gérer les parties inaccessibles de la mémoire.

Dans d'autres langages comme `C++`, le destructeur sert essentiellement à libérer la mémoire dynamique allouée aux attributs, et ceci de façon explicite et contrôlé.

5.2 Héritage

Il peut se trouver que l'on cherche à construire un objet `a` qui aurait toutes les propriétés et méthodes de `b`, comme si `b` en était un prototype, mais en le modifiant un peu, par exemple en précisant quelques propriétés supplémentaires ou en réformant des comportements.

Par exemple, un objet de type `FoodProduct` peut correspondre à peu près à ce qu'on attend d'un objet `Product`, mais en plus il aura une date limite de consommation, un mode de conservation, une application tarifaire particulière, etc. Nous allons construire une relation entre les deux classes `FoodProduct` et `Product` pour exprimer l'idée qui se cache derrière l'expression « peut correspondre à ».

L'héritage entre classes est une relation d'ordre entre les ensembles d'objets qui leur correspondent.

`A` hérite de `B` si les objets instances de `A` sont aussi instances de `B`. Autrement dit, on trouve l'ensemble des attributs et des méthodes de la classe `A` dans la classe`B`.

On peut gloser ainsi cette relation :

— La classe `B` est une sous-classe de la classe `A`
— La classe `B` est une classe fille de la classe `A`
— Un objet de la classe `B` est un objet de la classe `A`
— L'ensemble des objets de la classe `B` est un sous-ensemble des objets de la classe `A`

Attention, dans toutes ces relations, l'inverse est faux (par exemple un objet de la classe `A` n'est pas nécessairement un objet de la classe `B`).

Du bon usage de l'héritage

— Extension de classe
  On ajoute une méthode à une classe qui en étend une autre.
— Adaptation du code
  On `redéfinit` une méthode.

— Factorisation du code
On étend une classe abstraite qui contient du code destiné à être réutilisé à différents endroits du projet.

Listing 5 – Héritage A.java

```
public class A {
    public void message(){
        System.out.println("message d'un objet instance de A");
    }
}
```

Listing 6 – Héritage B.java

```
public class B extends A{
    public void message(){
        System.out.println("message d'un objet instance de B");
    }
}
```

Listing 7 – Héritage Main.java

```
public class Main {
    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        a.message();
        a = b;
        a.message();
        // b = a; ERROR
        b = (B) a;
        b.message();
    }
}
```

Classe, Classe abstraite et Interface

Rappeler les définitions et usages.

Langage Java, C#
— Une classe implémente une ou plusieurs interfaces
— Une classe étend une seule classe
Langage C++
— Une classe implémente un ou plusieurs headers
— Une classe étend une ou plusieurs classes
Langage PHP
— Traits de mixin
Langage Python
— Une classe étend une seule classe

6 Polymorphisme

Polymorphisme

Implémentations différentes de la même interface pour un opérateur, une procédure ou une fonction donnée.

6.1 Polymorphisme ad-hoc / overloading / surcharge

Définition différente des fonctions, procédures et opérateurs homonymes selon les types des opérandes (exclu le type de self passé en paramètre en Python)

Exemples : surcharge d'opérateurs en C++, surcharge de constructeurs en Java, etc.

Exemple en C++

Listing 8 – generics-1.cc

```
#include <iostream>
using namespace std;

int calculerSomme(int operande1, int operande2){
  int resultat = operande1 + operande2;
  return resultat;
}

string calculerSomme(string operande1, string operande2){
  string resultat = operande1 + operande2;
  return resultat;
}

int max (int o1, int o2){
  return o1 > o2 ? o1 : o2;
}

string max (string o1, string o2){
  return o1 > o2 ? o1 : o2;
}

int main(){
  int n1 = 4, p1 = 12;
  cout << n1 << " + " << p1 << " = " << calculerSomme(n1, p1) << endl;

  string n2 = "4", p2 = "12";
  cout << n2 << " + " << p2 << " = " << calculerSomme(n2, p2) << endl;

  cout << "Le max de " << n1 << " et " << p1 << " est " << max(n1, p1) << endl;

  cout << "Le max de " << n2 << " et " << p2 << " est " << max(n2, p2) << endl;
  return 0;
}
```

6.2 Polymorphisme paramétrique / type variable / template / type générique

Même définition des fonctions, procédures et opérateurs homonymes avec un type variable
Exemples : Types variables en Ocaml, templates en C++ et Java

6.3 Polymorphisme d'héritage / overriding / redéfinition

Définition des méthodes par spécialisation. Quand les méthodes ont déjà été définies ou non (interface en Java, méthode virtuelle pure en C++), je parle encore d'overriding, bien que cette

terminologie ne soit pas toujours employée.

Exemple : Implémentation d'une méthode d'une interface en Java, Redéfinition d'une méthode héritée en Java, C++ ou Python.

Exemple en Java

Listing 9 – polymorphism-1/src/A.java

```
public class A {
        public A(){};
        public final void identifyMyself1(){
                System.out.println("Je suis un A et on ne revient pas dessus");
        }
        public void identifyMyself2(){
                System.out.println("Je suis un A sauf contre-ordre");
        }
}
```

Listing 10 – polymorphism-1/src/B.java

```
public class B extends A {

        public B(){};

        public void identifyMyself2(){
                System.out.println("Je suis un B");
        }



}
```

Listing 11 – polymorphism-1/src/Main.java

```
public class Main {

        public static void main(String[] args) {

                B b = new B();

                b.identifyMyself1();
                b.identifyMyself2();

                A a = new A();
                a = (A)b;

                a.identifyMyself1();
                a.identifyMyself2();

                }
}
```

Exemple en C++

Listing 12 – polymorphism-1.cc

```
#include <iostream>
using namespace std;

class A{
public:
  void identifyMyself(void){
    cout << "Je suis un A " << endl;
  }
};

class B : A {
 public:
  void identifyMyself(void ){
    cout << "Je suis un B " << endl;
  }
};

class C : A{
 public:
  void identifyMyself(void){
    cout << "Je suis un C " << endl;
  }
};

int main(){
  class A a;
  class B b;
  class C c;
  a.identifyMyself();
  b.identifyMyself();
  c.identifyMyself();
  return 0;
}
```

Exemple en C++

Listing 13 – polymorphism-2.cc

```
#include <iostream>
using namespace std;

class A{
public:
  void identifyMyself1(void){
    cout << "Je suis un A " << endl;
  }
  virtual void identifyMyself2(void){
    cout << "Je suis un A sauf contreordre" << endl;
  }
};

class B : public A {
 public:
  void identifyMyself1(void ){
    cout << "Je suis un B " << endl;
  }
  void identifyMyself2(void ){
    cout << "Je suis un B " << endl;
  }
};

int main(){
  cout << "class A a:" << endl;
  class A a;
  a.identifyMyself1();
  a.identifyMyself2();

  cout << endl << "class B b:" << endl;
  class B b;
  b.identifyMyself1();
  b.identifyMyself2();

  cout << endl << "class A &ra = b" << endl;
  class A &ra = b;
  ra.identifyMyself1();
  ra.identifyMyself2();


  return 0;
}
```

Classes génériques

Exemple en C++

Listing 14 – generics-1-bis.cc

```
#include <iostream>
using namespace std;

int *calculerSommeInt(int *operande1, int *operande2){
  int *resultat = new int(*operande1 + *operande2);
  return resultat;
}

string *calculerSommeString(string *operande1, string *operande2){
  string *resultat = new string(*operande1 + *operande2);
  return resultat;
}

int main(){
  void *(* calculerSomme)(void *, void *);
  int n1 = 4, p1 = 12;

  calculerSomme = (void* (*)(void*, void*))&calculerSommeInt;

  cout << n1 << " + " << p1 << " = " << *(((int *)(calculerSomme(&n1, &p1)))) <<

  string n2 = "4", p2 = "12";

  calculerSomme = (void* (*)(void*, void*))&calculerSommeString;

  cout << n2 << " + " << p2 << " = " << *(((string *)(calculerSomme(&n2, &p2))))

  return 0;
}
```

Exemple en C++

Listing 15 – generics-2.cc

```
#include <iostream>
using namespace std;

template<class T>
T calculerSomme(T operande1, T operande2)
{
  T resultat = operande1 + operande2;
  return resultat;
}

template<typename T>
T myMax (T o1, T o2){
  return o1 > o2 ? o1 : o2;
}

int main(){
  int n1 = 4, p1 = 12;
  cout << n1 << " + " << p1 << " = " << calculerSomme(n1, p1) << endl;

  string n2 = "4", p2 = "12";
  cout << n2 << " + " << p2 << " = " << calculerSomme(n2, p2) << endl;

  cout << "Le max de " << n1 << " et " << p1 << " est " << myMax(n1, p1) << endl

  cout << "Le max de " << n2 << " et " << p2 << " est " << max(n2, p2) << endl;
  return 0;
}
```

Exemple en C++

Listing 16 – generics-3.cc

```
#include <iostream>
using namespace std;

class A{
public:
  string toString(){return "Je suis un A ";}
};

class B{
public:
  string toString(){return "Je suis un B ";}
};

template<class T1, class T2>
string conc(T1 operande1, T2 operande2)
{
  string resultat = operande1.toString() + operande2.toString();
  return resultat;
}

int main(){
  class A a;
  class B b;

  cout << conc(a, b) << endl;
  return 0;
}
```

Exemple en C++

Listing 17 – generics-4.cc

```
#include <iostream>
using namespace std;

template <class T>
class Item {
public:
  T element;
  Item<T> *next;

  Item();
  ~Item();
};

template <class T>
Item<T>::Item(){
  cout << "Item" << endl;
}

template <class T>
Item<T>::~Item(){
  cout << "~Item" << endl;
}

template <class T>
class LinkedList{

private:
  Item<T> *head;

public:
  LinkedList();
  ~LinkedList();

  T pop();
  void push(T);
  bool empty();
  void flush();
};

template <class T>
LinkedList<T>::LinkedList(){
  cout << "LinkedList" << endl;
  head = NULL;
```

```
}

template <class T>
LinkedList<T>::~LinkedList(){
  cout << "~LinkedList" << endl;
}

template <class T>
void LinkedList<T>::push(T element)
{
  Item<T> *tmp = new Item<T>();
  tmp->element = element;
    tmp->next = head;
    head = tmp;
    return;
}

template <class T>
T LinkedList<T>::pop()
{
    T tmp;
    Item<T> *ptmp = head;
    if (head != NULL){
        tmp = head->element;
        head = head->next;
        delete ptmp;
    }
    return tmp;
}

template <class T>
bool LinkedList<T>::empty()
{
  return head == NULL;
}

template <class T>
void LinkedList<T>::flush()
{
  while(head != NULL)
    pop();
}


int main(){
```

```
  LinkedList<string> l;
  l.push("A");
  l.push("B");
  l.push("C");
  cout << l.pop() << endl;
  cout << l.pop() << endl;
  l.flush();
  return 0;
}
```

Exemple en C++

Listing 18 – generics-5.cc

```
#include <iostream>
using namespace std;

class A{
public:
  string toString();
};

class B extends A{
public:
  string toString(){return "Je suis un B ";}
};

class C extends A{
public:
  string toString(){return "Je suis un C ";}
};

template<class T1, class T2>
string conc(T1 operande1, T2 operande2)
{
  string resultat = operande1.toString() + operande2.toString();
  return resultat;
}

int main(){
  class A a;
  class B b;
  class C c;

  cout << conc(a, b) << endl;
  return 0;
}
```

6.4 Héritage multiple

Listing 19 – Héritage multiple

```
class A {
public:
  void fa() { /* ... */ }
protected:
  int _x;
};

class B {
public:
  void fb() { /* ... */ }
protected:
  int _x;
};

class C: public B, public A {
public:
  void fc();
};
void C::fc() {
  int i;
  fa();
  //i = _x; // ERROR
  i = A::_x + B::_x; // resolution
}

int main(){
}
```