ÉCOLE NATIONALE D'INGÉNIEURS DE BREST

Langages Orientés Objet

Abstraction, Encapsulation

Elisabetta Bevacqua

Programmation orientée objet

- Paradigme de programmation informatique qui consiste en la définition et l'assemblage de briques logicielles appelées objets.
- Un objet représente :
  - une chose réelle, comme une voiture, une personne ou encore une page d'un livre ;
  - une chose intangible, comme un rendez-vous chez le médecin, une réservation au cinéma, ou un compte à la banque.
- En termes de programmation, un objet est n'importe quoi pour lequel nous avons besoin d'enregistrer et élaborer des données.
- Objet = entité = instance

Objets

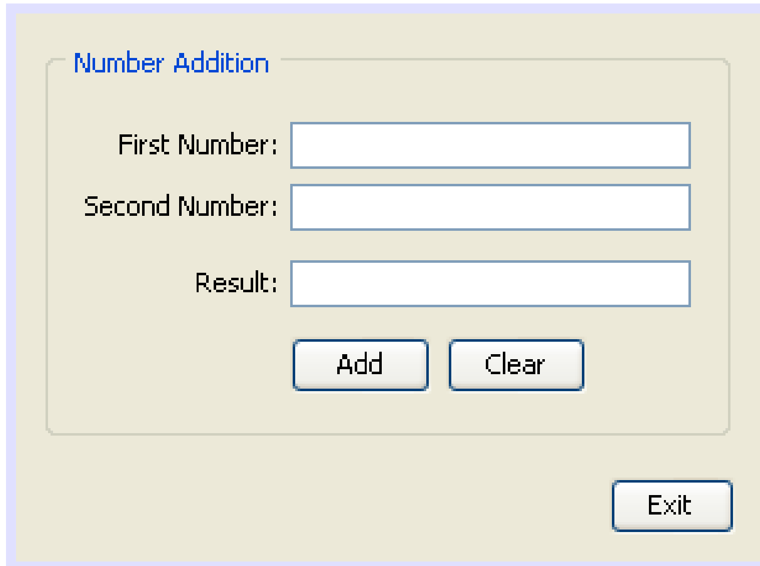
- Un objet est composé de :
  - données membres (attributs or propriétés), qui consistent en les informations relatives à l’objet dont l’application a besoin ;
  - fonctions (méthodes, opérations), qui définissent l’ensemble des opérations qui peuvent être faites sur l’objet.

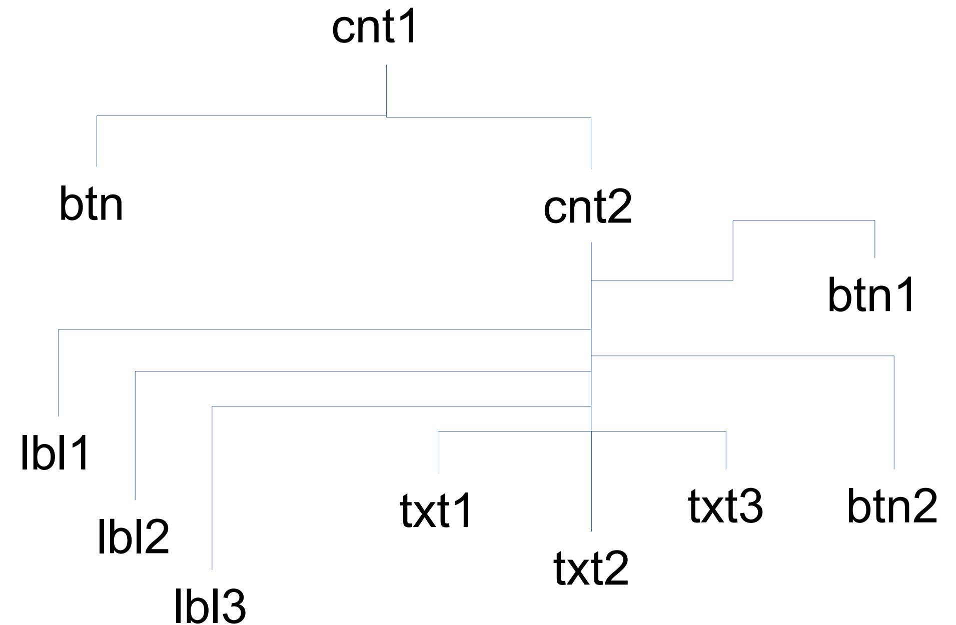
POO – quand l’utiliser

- La POO n’est pas la panacée pour résoudre n’importe quel type de problème.
- Comme tout paradigme, la programmation orientée objet se prête bien dans des cas particuliers, i.e. :
  - les éléments sont de types différents,
    - ont une structure interne et un ensemble de valeurs qui leur sont propres,
    - les actions qu’on peut leur appliquer leur sont spécifiques.
- Exemple : interfaces graphiques.

GUI

Graphical User Interface





- C'est seulement au moment du clic de l'utilisateur qu'il faudra chercher dans la structure l'élément cliqué,
  - chaque élément doit pouvoir gérer le clic,
  - mais chacun à sa façon (comportement dynamique).

Concepts fondamentaux de la POO

- Abstraction
- Encapsulation
- Collaborations
  - Association, Agrégation Composition
- Héritage
- Polymorphisme

Abstraction

- Pour créer des objets, en termes informatiques, nous avons besoin d'un moyen pour définir les attributs et les opérations des objets.
- Ce moyen s'appelle : **classe**
- La classe est une abstraction (simplification de la réalité) qui ne définit que ce qui est nécessaire pour les objets dans l'application.

Classe

- Classe => modèle pour créer les objets (*template*, *blueprint*).

Class : Princess

Attributs
Name Date of birth Nationality Eyes color

Operations
Get dressed Save herself Sing Dance

Instanciation

- Tout objet est une **instance** d'une classe.

Class : Princess

Attributs
Elsa 12 04 2013 Norway blue

Attributs
Belle 10 21 1992 France brown

état de l'objet

Attributs
Rapunzel 12 01 2010 Germany green

Attributs
Jasmine 11 24 1993 Irak black

Les classes dans les langages

- Tout langage de programmation utilisant le paradigme orienté objet fournit les moyens pour écrire des classes.
- Une classe est un **type**.
- Le concept reste le même mais la forme peut différer.
- Elles peuvent avoir une signification sémantique différente.

Les classes dans les langages

- En Java et en Python, un mot clef **class** et la classe a une sémantique de référence.

```java
//class Point
public class Point {
  //attributes (default access: friendly)
  double x;
  double y;

  //constructor
  public Point(double x, double y) {
    this.x = x;  this.y = y;
  }

  //method
  public void translate(double dx, double dy) {
    this.x += dx;  this.y += dy;
  }

  //main
  public static void main(String[] args) {
    Point p1 = new Point(10,15);       //instantiation
    p1.translate(5,5);                 //call method
    Point p2 = p1;                     //copy address
  }
}
```

```python
#class Point
class Point:
  #constructor
  def __init__(self, x=0, y=0):
    #attributes (default access: public)
    self.x = x
    self.y = y

  #method
  def translate(dx, dy):
    self.x += dx
    self.y += dy

p1 = Point(10, 15)  #instantiation
p1.translate(5,5)   #call method
p2 = p1             #copy address
```

Les classes dans les langages

- En C++, deux mots clef **struct** et **class** :
  - accès implicite aux attributs différent (public/private) ;
  - structures et classes ont une sémantique de valeur et de référence (selon les besoins du programmeur).

```
//class Point
class Point {
  //attributes (default access: private,
  //                 public in struct)
  double x;
  double y;

  public:
  //constructor
  Point(double x, double y)
  : this->x{x}
  , this->y{y}  {}

  //method
  void translate(double dx, double dy) {
        x += dx;
        y += dy;
  }
};

//main
int main() {
  Point p1{10,15};   //instantiation
  p1.translate(5,5); //call method
  Point p2 = p1;     //copy attributes
  Point *p3 = new Point(20,25);
  Point *p4 = p3;    //copy address
  return 0;
}
```

Les classes dans les langages

- En C#, deux mots clefs **struct** et **class** :
  - les structures ont une sémantique de valeur ;
  - les classes ont une sémantique de référence.

```
//class Point
public class Point {
  //attributes (default access: internal)
  double x;
  double y;

  //constructor
  public Point(double x, double y) {
        this.x = x;
        this.y = y;
  }

  //method
  public void translate(double dx, double dy) {
        this.x += dx;
        this.y += dy;
  }

  //main
  static void main(String[] args) {
        Point p1 = new Point(10,15);        //instantiation
        p1.translate(5,5);                  //call method
        Point p2 = p1;                      //copy address
  }
}
```

Les classes dans les langages

- En Rust, un mot clef **struct** :
  - les classes ont une sémantique de valeur ;
  - le clonage (duplication) doit être explicite.

```
//class Point
#[derive(Clone)]
pub struct Point {
    //attributes (default access: module)
    x: f64,
    y: f64,
}

impl Point {
    //constructor
    pub fn new(x: f64, y: f64) -> Self {
        Self { x, y }
    }

    //method
    pub fn translate(&mut self, dx: f64, dy: f64) {
        self.x += dx;
        self.y += dy;
    }
}

fn main() {
    let mut p1 = Point::new(10.0, 15.0);   //instantiation
    p1.translate(5.0, 5.0);                //call method
    let _p2 = p1.clone();                  //explicit copy attributes
    let p3 = p1;                           //move value
}
```

ÉCOLE NATIONALE D'INGÉNIEURS DE BREST

Bref...

- Dans tous les exemples ce qui est commun :
  - Le **concept** (abstraction de la réalité, type).
  - Données membres (attributs).
  - Accès aux données membres.
  - Constructeur (fonction d'initialisation).
  - Fonctions membres (méthodes).
  - Instanciation.
  - Invocations des méthodes.

Encapsulation

- L'encapsulation (ou masquage d'information) consiste à séparer les aspects externes d'un objet, accessibles par les autres objets, des détails de son implémentation interne, rendus invisibles aux autres objets.
- Elle permet d'assurer l'intégrité des données lorsque les attributs dépendent les uns des autres (invariant).
- Pour créer des objets à partir d'une classe, affecter des valeurs à ses attributs et invoquer les méthodes, il n'est pas nécessaire de comprendre comme la classe a été codée. Nous avons besoin juste de connaître son nom et ses attributs et méthodes publiques (interface).

Encapsulation

- L’encapsulation n’expose que les détails nécessaires à l’utilisation du type.
- Elle cache à l’utilisateur tous les détails d’implémentation.

L'encapsulation dans les langages

- Tout langage de programmation utilisant le paradigme orienté objet fournit les moyens pour appliquer l'encapsulation :
- En C++, C#, Java, les mot clefs :
  - **public**, **private**, **protected**
- En Python les soulignés comme préfixé du nom de l'attribut :
  - _ (protégé), __ (privé)
- En Rust, le mot clef **pub**.

L'encapsulation dans les langages

- À nouveau, le concept est le même mais la signification sémantique peut être différente.

C++

```
class Point {
  //attributes (default access: private,
  //              public in struct)
  double x;
  double y;
```

C#

```
public class Point {
  //attributes (default access: internal)
  double x;
  double y;
```

Python

```
class Point:
  #constructor
  def __init__(self, x=0, y=0):
    #attributes (default access: public)
    self.x = x
    self.y = y
```

Rust

```
pub struct Point {
    //attributes (default access: module)
    x: f64,
    y: f64,
}
```

```
public class Point {
  //attributes (default access: friendly)
  double x;
  double y;
```

Java

Accès aux attributs en Rust

- Vu en S4PRG :
  - le code d'un module a implicitement accès à toutes les ressources du même module et de tout ceux qui le contiennent ;
  - un code n'a pas librement accès aux ressources d'un module qu'il contient :
    - sauf si tout ce qui mène à cette ressource est qualifié https://web.enib.fr/~harrouet/rust/rust_06_organisation.html#pub
- L'organisation des modules en Rust doit être explicitement déclarée dans le code.

Exemple en Rust

Point sur un cercle unitaire

```
mod invariant {
    #[derive(Debug, Clone)]
    pub struct PointOnCircle {
        //invariant : sqrt(x*x+y*y) = 1
        x: f64,
        y: f64,
    }

    impl PointOnCircle {
        //constructor
        pub fn new(x: f64, y: f64) -> Self {
            let sqr_mag = x * x + y * y;
            if sqr_mag > 0.0 {
                let factor = 1.0 / sqr_mag.sqrt();
                Self {
                    x: x * factor,
                    y: y * factor,
                }
            } else {
                Self { x: 1.0, y: 0.0 }
            }
        }

        //methods
        pub fn _get_x(&self) -> f64 {
            self.x
        }

        pub fn set_x(
                &mut self,
                x: f64,
            ) {
                self.x = x.clamp(-1.0, 1.0);
                self.y *= ((1.0 - self.x * self.x) /
                           (self.y * self.y)).sqrt();
            }

            //... get_y et set_y
    }
} //mod invariant

use invariant::PointOnCircle;
use point::Point;

fn main() {
    let mut p1 = PointOnCircle::new(0.1, 0.6);
    //p1.x = 0.5; //error x of PointOnCircle is private
    p1.set_x(0.5);
    println!("y: {}", p1.get_y());
    p1.set_y(0.2);
    println!("{:?}", p1);
}
```

Exercice en Rust
classe Sensor

- Déclarer un type de données Sensor contenant quatre champs :
  - le nom du capteur ;
  - le min et max des valeurs qu'il peut contenir ;
  - un vecteur de valeurs (double).
  - **Attention !** Appliquer l'encapsulation pour respecter l’invariant.
- Fournir un constructeur qui reçoit le nom et un min et un max.
- Déclarer et définir une fonction membre qui permet d'ajouter une valeur.
- Déclarer et définir trois fonctions membres, une qui renvoie le nom du capteur, une qui renvoie le nombre de valeurs stockées et une dernière qui renvoie un tuple contenant les deux valeurs min et max.

Exercice (la suite)

- Ajouter deux fonctions qui permettent de calculer respectivement la moyenne et l'écart type de l'ensemble de valeurs.
- Dans le main() :
  - Déclarer un Sensor mutable (valeurs entre 0 et 100).
  - Le remplir à l’aide d’une boucle de 0 à 12 en ajoutant au Sensor la valeur de l’indice * 10.
  - Modifier le min → 28.
  - Afficher le Sensor (à l’aide de l’affichage debug).
  - Afficher la moyenne et l’écart type du Sensor.

ÉCOLE NATIONALE D'INGÉNIEURS DE BREST

Langages Orientés Objet

Abstraction, Encapsulation

Elisabetta Bevacqua