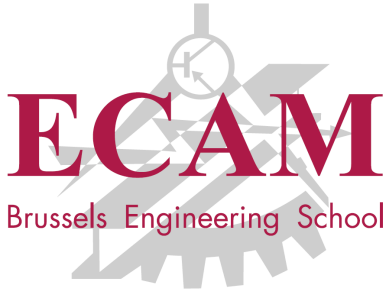


E303B Programmation orientée objet

Séance 1

Relations de composition et d'agrégation

Sébastien Combéfis, Quentin Lurkin

2017–2018

Objectifs

- Rappels sur la programmation orientée objet
  - Objet et classe
  - Variable d'instance, constructeur et méthode
- Relations entre objets et classe
  - Relation de composition et d'agrégation (*has-a*)
  - Relation d'utilisation (*uses*)

Rappels

Vie d'un objet

- Trois étapes principales dans la vie d'un objet
  - Création de l'objet (*initialisation*)
  - Appel de méthode et changement d'état (*utilisation*)
  - Destruction de l'objet (*finalisation*)

```
1  words = []
2
3  words.append('Hello')              # Appel de la méthode append
4  words.append('World!')             # de l'objet words (list)
5
6  print(' '.join(words))
```

```
Hello World!
```

Et en Ruby...

- Langage interprété proche du Python
  - Notation explicite pour la création d'un nouvel objet (`new`)
  - Méthode `join` sur la liste au lieu de la chaine de caractères

```
1  words = Array.new
2
3  words.push('Hello')                  # Appel de la méthode push
4  words.push('World!')                 # de l'objet words (Array)
5
6  puts(words.join(' '))
```

Et en Java...

- Langage compilé et typé statiquement proche du C#
  - Déclaration du type des éléments de la liste
  - Délimitation des corps (classe, méthode) avec accolades
  - Méthode `main` comme point d'entrée

```
1  import java.util.ArrayList;
2
3  public class Rappels
4  {
5      public static void main (String[] args)
6      {
7          ArrayList<String> words = new ArrayList<String>();
8
9          words.add ("Hello");          // Appel de la méthode add
10         words.add ("World!");         // de l'objet words (ArrayList)
11
12         System.out.println (String.join (" ", words));
13     }
14 }
```

Et en PHP...

- Langage interprété souvent utilisé côté serveur d'un site web
  - Nom des variable préfixé du symbole $
  - Utilisation d'une flèche (->) pour appeler une méthode
  - Fonction globale `join` et pas méthode d'un objet

```php
<?php
   $words = new ArrayObject();

   $words->append ("Hello");          // Appel de la méthode append
   $words->append ("World!");         // de l'objet $words (Array)

   echo join (' ', $words->getArrayCopy()) . "\n";
?>
```

Et en C++...

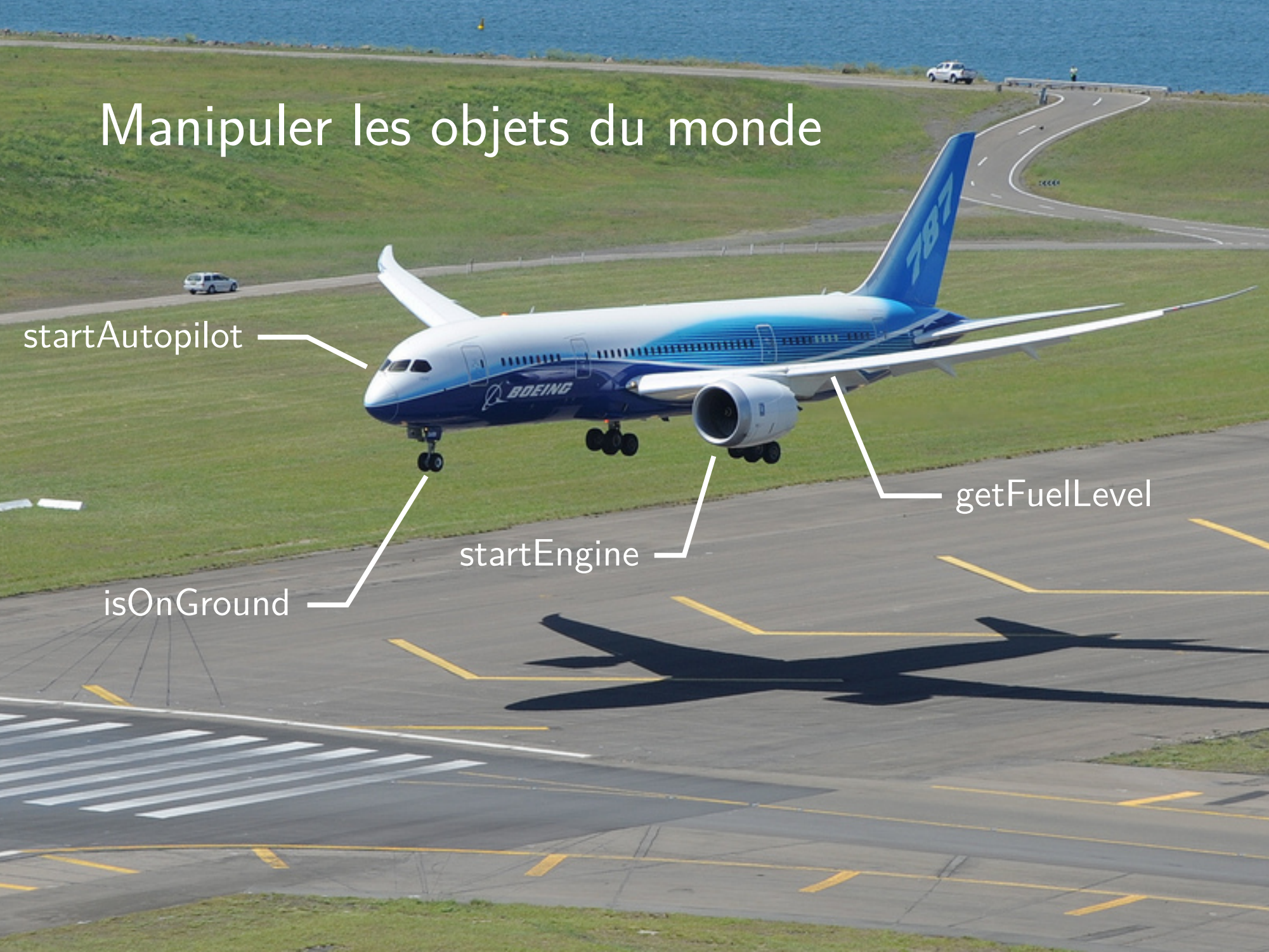
- Langage compilé orienté objet comme *« extension »* du C
  - Également typé statiquement (variable et contenu des listes)
  - Surcharge possible des opérateurs (<<)

```
1  #include <iostream>
2  #include <vector>
3
4  using namespace std;
5
6  int main (int argc, char *argv[])
7  {
8      vector<string> words;
9
10     words.push_back ("Hello");      // Appel de la méthode push_back
11     words.push_back ("World!");     // de l'objet words (vector)
12
13     string s;
14     join(words, ' ', s);
15     cout << s << endl;
16
17     return 0;
18 }
```

Et en C#...

- Langage compilé et typé statiquement proche du Java
  - Déclaration d'un espace de noms pour la classe

```
1  using System;
2  using System.Collections.Generic;
3
4  namespace Cours1
5  {
6      public class Rappels
7      {
8          public static void Main (string[] args)
9          {
10             List<string> words = new List<string>();
11
12             words.Add ("Hello");        // Appel de la méthode Add
13             words.Add ("World!");       // de l'objet words (List)
14
15             Console.WriteLine (String.Join (" ", words));
16         }
17     }
18 }
```


Manipuler les objets du monde
startAutopilot
isOnGround
startEngine
getFuelLevel
BOEING
787

Objet

- Un objet possède deux types de membres
  - des attributs (*caractéristique*)
  - des méthodes (*fonctionnalités*)
- L'état de l'objet peut varier tout au long de l'exécution

  Défini par l'ensemble des valeurs des attributs

Marque : Sony-Ericsson
Modèle : S500i
Couleur : Mysterious Green
Batterie : 80%
Luminosité : 60%

Marque : Sony-Ericsson
Modèle : S500i
Couleur : Spring Yellow
Batterie : 35%
Luminosité : 90%

Attribut

- Un attribut possède une valeur

  Cette valeur fait partie de l'état de l'objet

- Deux niveaux d'accessibilité pour les attributs

  En lecture seule ou en lecture et écriture

```
1  Console.WriteLine ("Taille de la liste : " + words.Count);
2
3  Console.WriteLine ("Ancienne capacité : " + words.Capacity);
4  words.Capacity = 10;
5  Console.WriteLine ("Nouvelle capacité : " + words.Capacity);
```

```
Taille de la liste : 2
Ancienne capacité : 4
Nouvelle capacité : 10
```

Fonctionnalité

- Une fonctionnalité est appliquée sur un objet cible

  Elle agit sur l'état de l'objet, en le lisant et/ou le modifiant

```
1  bool contains = words.Contains ("Hello");
2  bool allShorts = words.TrueForAll (s => s.Length <= 5);
3  Console.WriteLine (contains + ", " + allShorts);
4
5  Console.WriteLine ("Liste : " + String.Join(", ", words));
6  words.Clear();
7  Console.WriteLine ("Liste : " + String.Join (", ", words));
```

```
True, False
Liste : Hello, World!
Liste :
```

Destruction

- Destruction explicite d'un objet

  Libération de la mémoire et des ressources allouées

```
1  class Bullshit
2  {
3      public:
4          ~Bullshit();
5  };
6
7  Bullshit::~Bullshit()
8  {
9      cout << "Damn, I've been destroyed!";
10 }
11
12 int main (int argc, char *argv[])
13 {
14     Bullshit *b = new Bullshit();
15     delete b;
16
17     return 0;
18 }
```

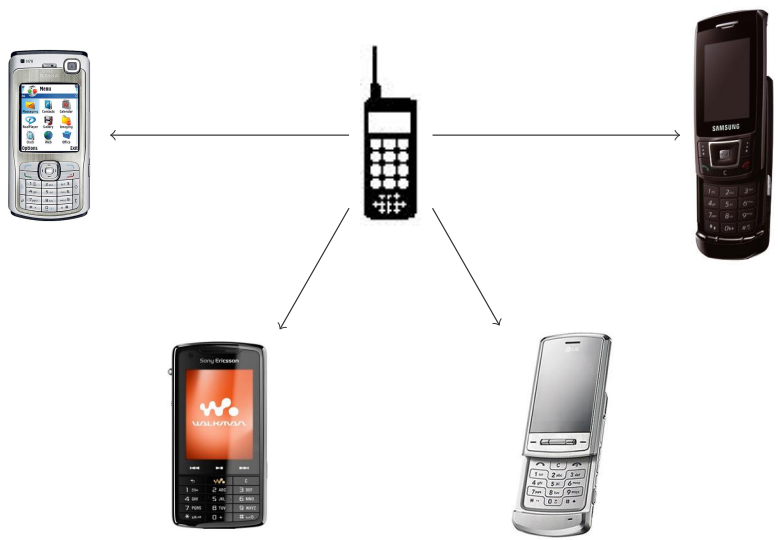
Classe

- Une classe permet de construire des objets

  Il s'agit d'un modèle qui décrit les attributs et fonctionnalités

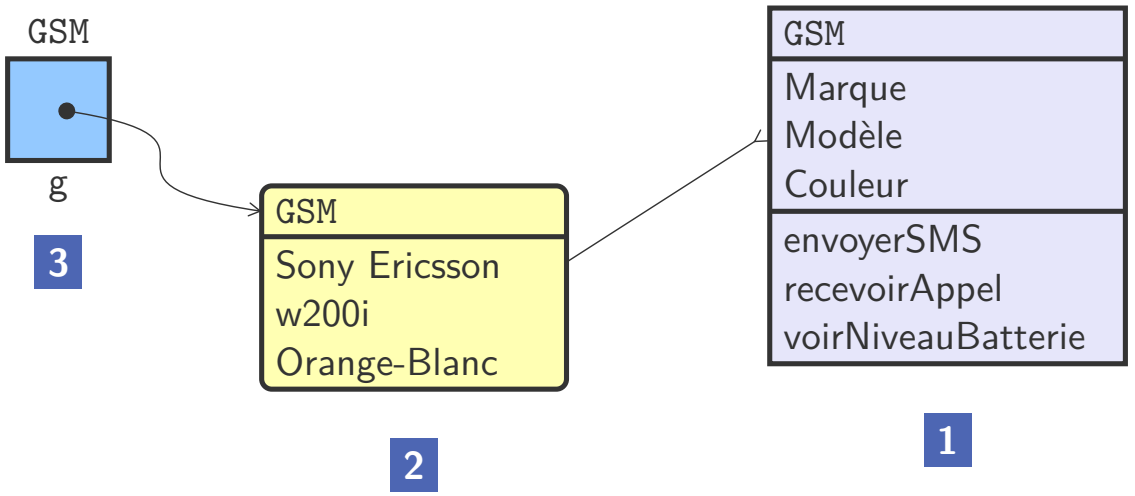
- Un objet est une instance d'une classe

  On crée un nouvel objet à partir d'une classe



Instance

- Trois éléments distincts sont à identifier
  1. Une classe est un modèle
  2. Un objet est une **instance** d'une classe (——≺)
  3. Une variable stocke une **référence** vers l'objet (●——→)

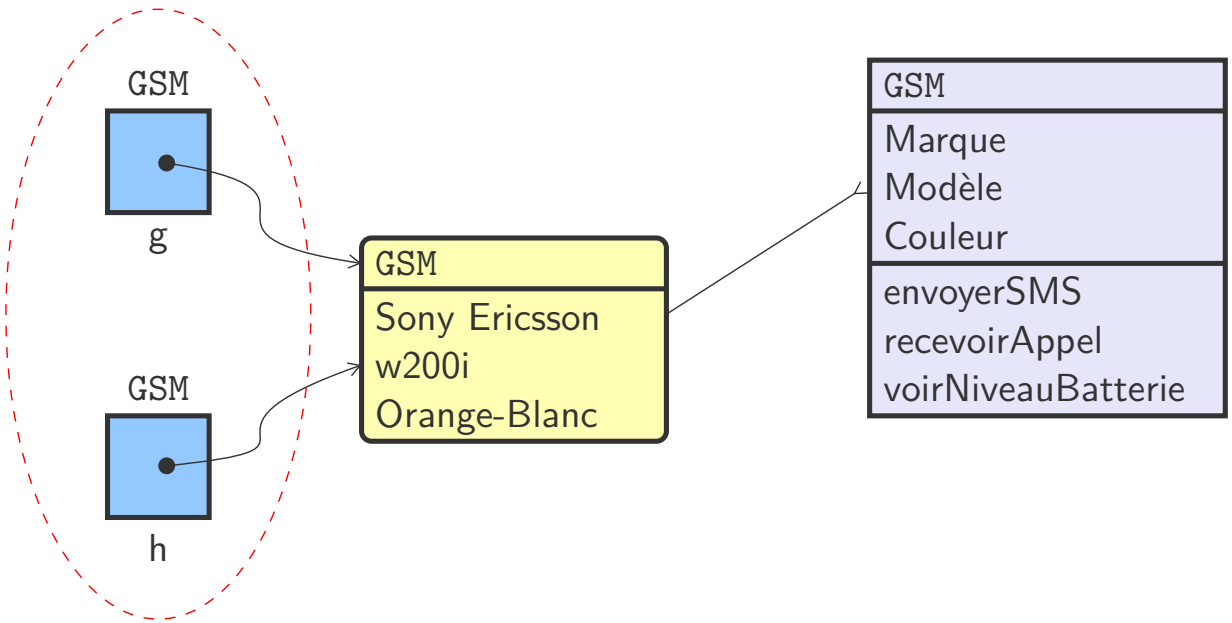


Alias

- Une variable est une référence vers un objet

  La variable permet d'accéder à l'objet et ainsi le manipuler

- Un alias réfère le même objet qu'une autre variable



Définir une classe

- Définition d'une classe `Contact` représentant un contact

  Caractérisé par un prénom, un nom et un numéro de téléphone

```
1  class Contact:
2      def __init__(self, firstname, lastname):
3          self.firstname = firstname
4          self.lastname = lastname
5          self.phonenumber = 0
6
7      def setPhoneNumber(self, number):
8          self.phonenumber = number
9
10     def __str__(self):
11         return '{} {} ({})'.format(self.firstname, self.lastname,
           self.phonenumber)
12
13 marchand = Contact('Cédric', 'Marchand')
14 print(marchand)                              # Cédric Marchand (0)
15
16 marchand.setPhoneNumber(8172)
17 print(marchand)                              # Cédric Marchand (8172)
```

Constructeur et variable d'instance

- Le constructeur initialise l'état de l'objet

  L'instantiation invoque implicitement le constructeur

- Initialisation des variables d'instance

  On affecte une valeur à chaque variable d'instance

```
1  class Contact:
2      def __init__(self, firstname, lastname):
3          self.firstname = firstname
4          self.lastname = lastname
5          self.phonenumber = 0
6
7      # [...]
8
9  marchand = Contact('Cédric', 'Marchand')
10 print(marchand)
11
12 # [...]
```

Et en Ruby...

- Définition d'une classe `Contact`
  - Constructeur toujours nommé `initialize`
  - Variables d'instance identifiée par un `@` au début du nom

```ruby
class Contact
    def initialize(firstname, lastname)
        @firstname = firstname
        @lastname = lastname
        @phonenumber = 0
    end

    # [...]
end

marchand = Contact.new('Cédric', 'Marchand')
puts(marchand)

# [...]
```

Et en Java... (1)

- Définition d'une classe `Contact`
  - Constructeur porte le même nom que la classe
  - Mot réservé `this` représente l'objet cible

```
1  class Contact
2  {
3      private String firstname, lastname;
4      private int phonenumber;
5
6      public Contact (String firstname, String lastname)
7      {
8          this.firstname = firstname;
9          this.lastname = lastname;
10         this.phonenumber = 0;
11     }
12
13     // [...]
14 }
```

Et en Java... (2)

- Définition d'une classe `Program`
  - Ne sera pas instanciée
  - Uniquement pour héberger méthode `main`

```
1  public class Program
2  {
3      public static void main (String[] args)
4      {
5          Contact marchand = new Contact ("Cédric", "Marchand");
6          System.out.println (marchand);
7
8          // [...]
9      }
10 }
```

Et en PHP...

- Définition d'une classe `Contact`
  - Constructeur toujours nommé `__construct`
  - Variable spéciale `$this` représente l'objet cible

```php
<?php
    class Contact
    {
        private $firstname, $lastname;
        private $phonenumber;

        function __construct ($firstname, $lastname)
        {
            $this->firstname = $firstname;
            $this->lastname = $lastname;
            $this->phonenumber = 0;
        }

        // [...]
    }

    $marchand = new Contact ('Cédric', 'Marchand');
    echo $marchand . "\n";

    // [...]
?>
```

Et en C++... (1)

- Définition d'une classe `Contact`
  - Constructeur même nom que la classe (`Contact::Contact`)
  - Mot réservé `this` représente l'objet cible

```
1  #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6  class Contact
7  {
8      private:
9          string firstname, lastname;
10         int phonenumber;
11     public:
12         Contact (string, string);
13
14     // [...]
15 };
16
17 Contact::Contact (string firstname, string lastname)
18 {
19     this->firstname = firstname;
20     this->lastname = lastname;
21     this->phonenumber = 0;
22 }
```

Et en C++...

- Programme utilisant la classe `Contact`
  - Création d'une nouvelle instance sans mot réservé spécial
  - Écriture de l'objet sur la sortie standard imprime son addresse

```
1  int main()
2  {
3      Contact marchand ("Cédric", "Marchand");
4      cout << marchand << endl;
5
6      // [...]
7  }
```

Et en C#... (1)

- Définition d'une classe `Contact`
  - Constructeur porte le même nom que la classe
  - Mot réservé `this` représente l'objet cible

```
1  class Contact
2  {
3      private string firstname, lastname;
4      private int phonenumber;
5
6      public Contact (string firstname, string lastname)
7      {
8          this.firstname = firstname;
9          this.lastname = lastname;
10         this.phonenumber = 0;
11     }
12
13     // [...]
14 }
```

Et en C#... (2)

- Définition d'une classe `Program`
  - Ne sera pas instanciée
  - Uniquement pour héberger méthode `Main`

```
1  public class Program
2  {
3      public static void Main (string[] args)
4      {
5          Contact marchand = new Contact ("Cédric", "Marchand");
6          Console.WriteLine (marchand);
7
8          // [...]
9      }
10 }
```

Variable d'instance

- Une variable d'instance est liée à l'objet

  Sa valeur fait partie de l'état de l'objet

- Peuvent avoir une certaine visibilité

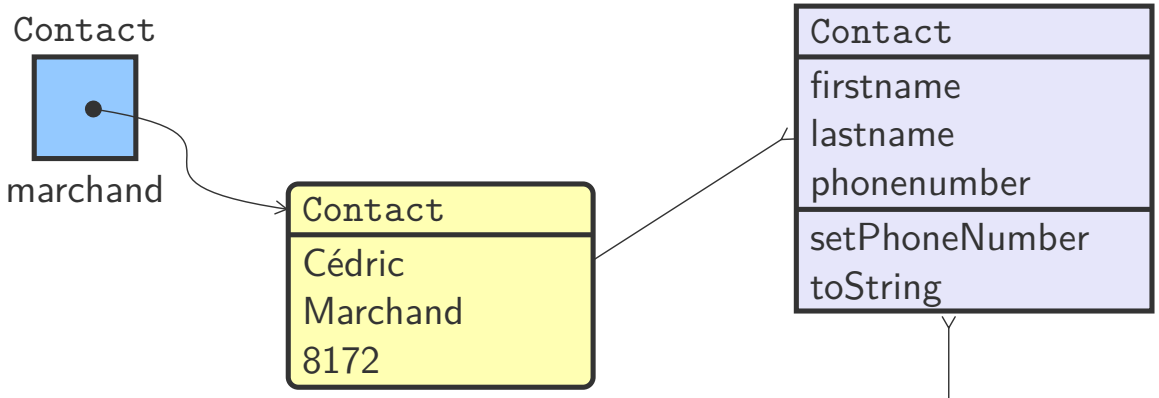
  Uniquement accessible dans la classe ou visible de l'extérieur

```
1  class Contact
2  {
3      private string firstname, lastname;
4      private int phonenumber;
5
6      // [...]
7  }
```

Instanciation

- On peut créer plusieurs instances d'une classe

  Chaque instance a son identité propre




```
marchand = Contact('Cédric', 'Marchand')
marchand.setPhoneNumber(8172)
print(marchand)

flemal = Contact('Clémence', 'Flemal')
```

Méthode

- Une méthode (d'instance) représente une fonctionnalité

  Elle agit sur une instance spécifique, l'objet cible

- Peuvent avoir une certaine visibilité

  Uniquement accessible dans la classe ou visible de l'extérieur

```
1  # [...]
2
3      def setPhoneNumber(self, number):
4          self.phonenumber = number
5
6  # [...]
7
8  marchand.setPhoneNumber(8172)
9  print(marchand)
```

Appel de méthode

- Une méthode (d'instance) est appelée sur un objet cible

  La méthode aura accès aux variables d'instances de cet objet

- Mot réservé spécifique pour accéder à l'objet cible

  Référence vers l'objet cible (`this`, `self`...)

```
1      public void SetPhoneNumber(int number)
2      {
3          this.phonenumber = number;
4      }
5
6  // [...]
7
8          marchand.SetPhoneNumber (8172);       // this fera référence
9          Console.WriteLine (marchand);         // à marchand
```

Et en Ruby...

- Définition d'une méthode `setPhoneNumber`
  - Une méthode se définit avec le mot réservé `def`
  - Méthode spéciale `to_s()` pour représentation textuelle

```
1  class Contact
2      # [...]
3
4      def setPhoneNumber(number)
5          @phonenumber = number
6      end
7
8      def to_s()
9          "#{@firstname} #{@lastname} (#{@phonenumber})"
10     end
11 end
12
13 # [...]
14
15 marchand.setPhoneNumber(8172)
16 puts(marchand)
```

Et en Java... (1)

- Définition d'une méthode `setPhoneNumber`
  - Une méthode se définit avec une structure particulière et typée
  - Méthode spéciale `toString()` pour représentation textuelle

```
1  class Contact
2  {
3      // [...]
4
5      public void setPhoneNumber (int number)
6      {
7          this.phonenumber = number;
8      }
9
10     @Override
11     public String toString()
12     {
13         return String.format("%s %s (%d)", firstname, lastname,
           phonenumber);
14     }
15 }
```

Et en Java... (2)

- Appel de la méthode `setPhoneNumber`
  - Exécution de la méthode avec l'opérateur d'appel (`.`)
  - Pas de valeur de retour pour une méthode de type `void`

```
1  public class Program
2  {
3      public static void main (String[] args)
4      {
5          // [...]
6
7          marchand.setPhoneNumber (8172);
8          System.out.println (marchand);
9      }
10 }
```

Et en PHP...

- Définition d'une méthode `setPhoneNumber` et appel
  - Une méthode se définit avec le mot réservé `function`
  - Méthode spéciale `__toString()` pour représentation textuelle

```php
<?php
class Contact
{
    // [...]

    function setPhoneNumber ($number)
    {
        $this->phonenumber = $number;
    }

    public function __toString()
    {
        return sprintf ("%s %s (%d)", $this->firstname, $this->lastname, $this
        ->phonenumber);
    }
}

// [...]

$marchand->setPhoneNumber (8172)
echo $marchand . "\n";
?>
```

Et en C++... (1)

- Définition d'une méthode « amie »
  - Méthode définie dans `Contact`, mais vient d'une autre classe
  - Redéfinition de l'opérateur <<

```
1  // [...]
2
3  class Contact
4  {
5      // [...]
6      public:
7          // [...]
8          void setPhoneNumber (int);
9
10     friend ostream& operator<< (ostream &strm, const Contact &contact)
11     {
12         return strm << contact.firstname << " " << contact.lastname << " (" <<
            contact.phonenumber << ")";
13     }
14 };
```

Et en C++... (2)

- Définition d'une méthode `Contact::setPhoneNumber` et appel
  - Une méthode se définit avec une structure particulière et typée
  - Pas de valeur de retour pour une méthode de type `void`

```
1  // [...]
2
3  void Contact::setPhoneNumber (int number)
4  {
5      this->phonenumber = number;
6  }
7
8  int main()
9  {
10     // [...]
11
12     marchand.setPhoneNumber (8172);
13     cout << marchand << endl;
14 }
```

En C#... (1)

- Définition d'une méthode `SetPhoneNumber`
  - Une méthode se définit avec une structure particulière et typée
  - Méthode spéciale `ToString()` pour représentation textuelle

```
1  class Contact
2  {
3      // [...]
4
5      public void SetPhoneNumber (int number)
6      {
7          phonenumber = number;
8      }
9
10     public override string ToString()
11     {
12         return String.Format("{0} {1} ({2})", this.firstname, this.
           lastname, this.phonenumber);
13     }
14 }
```

En C#... (2)

- Appel de la méthode `SetPhoneNumber`
  - Exécution de la méthode avec l'opérateur d'appel (`.`)
  - Pas de valeur de retour pour une méthode de type `void`

```
1  public class Program
2  {
3      public static void Main (string[] args)
4      {
5          // [...]
6
7          marchand.SetPhoneNumber (8172);
8          Console.WriteLine (marchand);
9      }
10 }
```

Accesseur

- Un getter permet de lire une variable d'instance

  Ou de manière générale, d'obtenir une information sur l'état

- Un setter permet de modifier une variable d'instance

  Ou de manière générale, de modifier l'état

```
1  # [...]
2
3      def getPhoneNumber(self):
4          return self.phonenumber
5
6      def setPhoneNumber(self, number):
7          self.phonenumber = number
8
9  # [...]
10
11 marchand.setPhoneNumber(8172)
12 print(marchand.getPhoneNumber())
```

Attribut Python

- Un attribut peut être défini avec une décoration

  Est vu de l'extérieur comme une variable d'instance publique

```
1  class Contact:
2      def __init__(self, firstname, lastname):
3          # [...]
4          self.__phonenumber = 0
5
6      @property
7      def phonenumber(self):
8          return self.__phonenumber
9
10     @phonenumber.setter
11     def phonenumber(self, value):
12         self.__phonenumber = value
13
14     # [...]
15
16 marchand.phonenumber = 666
17 print(marchand)
```

Propriété C#

- Une propriété permet de créer des accesseurs

  Est vu de l'extérieur comme une variable d'instance publique

- Peut être uniquement en lecture seule ou modifiable

  Getter avec `get` et setter avec `set`

```
1  // [...]
2
3      public int PhoneNumber
4      {
5          get { return phonenumber; }
6          set { phonenumber = value; }
7      }
8
9  // [...]
10
11         marchand.PhoneNumber = 666;
12         Console.WriteLine (marchand.PhoneNumber);
```

Relations entre classes

Relation entre classes

- Une classe définit un nouveau type de donnée

  On peut l'utiliser pour définir des objets de ce type

- Plusieurs classes peuvent être liées entre elle

  Plusieurs types de relation sont possibles entre classes

- Création de dépendances entre classes

  Et donc entre les instances de ces classes

Représentation d'un dé (1)

Constructeur

Nombre de faces désirées (6 par défaut)

Attributs

– Nombre de faces

– Face visible

Fonctionnalités

– Lancer le dé

(Il s'agit d'un rhombicuboctaèdre !)

Représentation d'un dé (2)

```
1  public class Die
2  {
3      public readonly int nbFaces;  // Constante
4      private int visibleFace;
5      private static Random generator = new Random();
6
7      public int VisibleFace {
8          get { return visibleFace; }
9      }
10
11     public Die() : this(6){}       // Appel de l'autre constructeur
12
13     public Die (int faces)
14     {
15         nbFaces = faces;
16         Roll();
17     }
18
19     public void Roll()
20     {
21         visibleFace = generator.Next (nbFaces) + 1;
22     }
23 }
```

Et en Python...

- Déclaration explicite valeur par défaut du nombre de faces

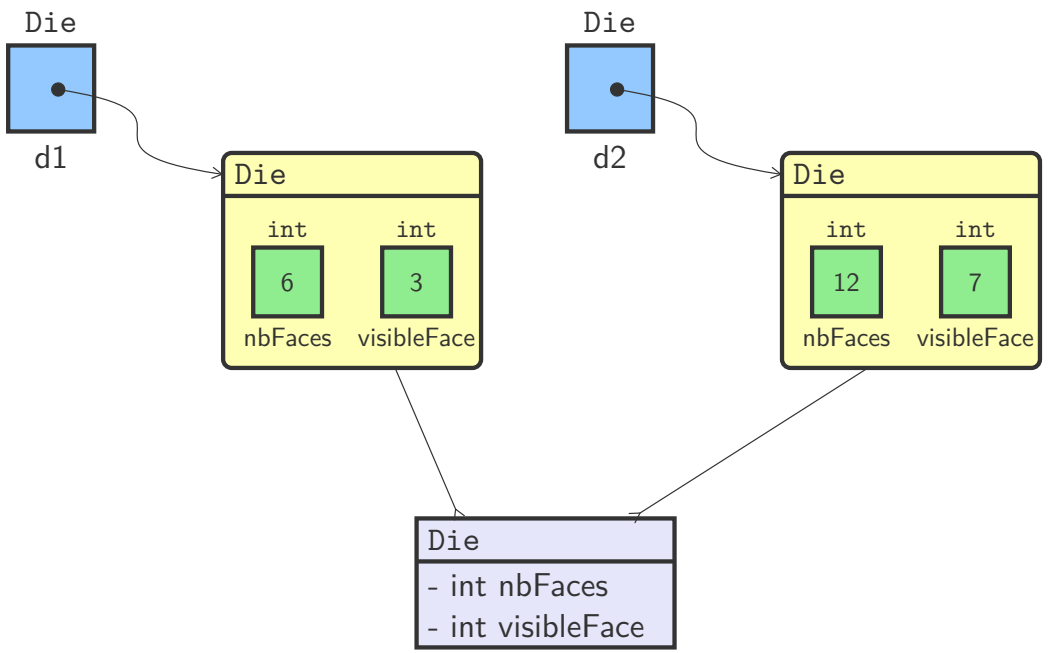
```
1  from random import randint
2
3  class Die:
4      def __init__(self, faces=6):
5          self.__nbfaces = faces
6          self.roll()
7
8      @property
9      def nbfaces(self):
10         return self.__nbfaces
11
12     @property
13     def visibleface(self):
14         return self.__visibleface
15
16     def roll(self):
17         self.__visibleface = randint(1, self.nbfaces)
```

Et en Java...

```java
public class Die
{
    public final int nbFaces;      // Constante
    private int visibleFace;

    public Die()
    {
        this(6);
    }

    public Die (int faces)
    {
        nbFaces = faces;
        roll();
    }

    public int getVisibleFace()
    {
        return visibleFace;
    }

    public void roll()
    {
        visibleFace = (int) (Math.random() * nbFaces) + 1;
    }
}
```

Création de dés

```
1  public static void Main (string[] args)
2  {
3      Die d1 = new Die();          // Dé à 6 faces
4      Die d2 = new Die (12);       // Dé à 12 faces
5
6      Console.WriteLine (d1.VisibleFace);
7      Console.WriteLine (d2.VisibleFace);
8  }
```



Représentation d'une paire de dés (1)

- Objet représentant deux dés ayant le même nombre de faces

  Comme le simple dé, mais deux faces visibles

```
1  public class PairOfDice
2  {
3      private readonly int nbFaces;
4      private int visibleFace1, visibleFace2;
5      private static Random generator = new Random();
6
7      public PairOfDice (int faces)
8      {
9          nbFaces = faces;
10         visibleFace1 = generator.Next (nbFaces) + 1;
11         visibleFace2 = generator.Next (nbFaces) + 1;
12     }
13
14     public void PrintFaces()
15     {
16         Console.WriteLine (String.Format ("{0}, {1}", visibleFace1,
            visibleFace2));
17     }
18 }
```

Composition de classes

- Définir une nouvelle classe à partir d'autres

  En déclarant des variables d'instance des types utilisés

- Éviter la répétition de code inutile

  Facilite les corrections et les évolutions

- Construire des objets à partir de blocs simples

  Comme on le fait dans la vraie vie...

Représentation d'une paire de dés (2)

- Un objet `PairOfDice` est composé à partir de deux objets `Die`

  On déclare deux variables d'instance de type `Die`

```
1  public class PairOfDice
2  {
3      private int nbFaces;
4      private Die die1, die2;          // Composition à partir
5                                       // de deux objets Die
6      public PairOfDice (int faces)
7      {
8          nbFaces = faces;
9          die1 = new Die (faces);
10         die2 = new Die (faces);
11     }
12
13     public void PrintFaces()
14     {
15         Console.WriteLine (String.Format ("{0}, {1}", die1.
           VisibleFace, die2.VisibleFace));
16     }
17 }
```

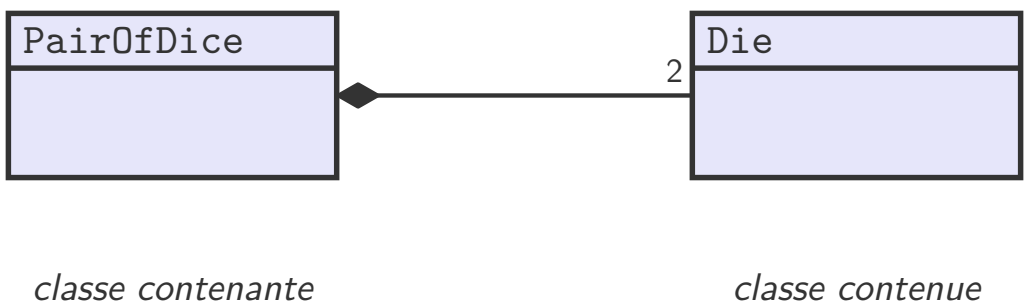
Relation de composition

- Une classe *A* est composée à partir d'une classe *B*

  Une instance de A a des variables d'instance de type B

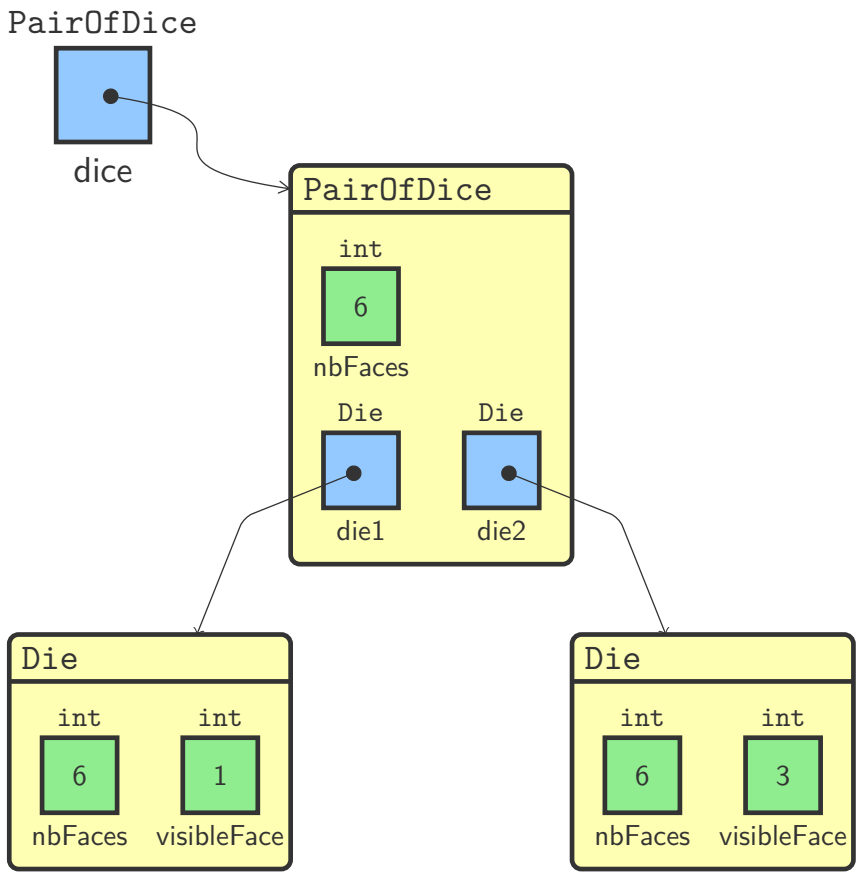
- Également appelée relation *has-a* (ou *is-made-up-of*)

  Une instance de A has-a une(des) instance(s) de B



Une paire de dés

```
1  PairOfDice dice = new PairOfDice (6);
2  dice.PrintFaces();                          // 1, 3
```



Lien fort entre les instances composées

- Objets contenus fortement liés à l'objet contenant

  Ils disparaissent de la mémoire avec l'objet contenant

- Instances contenues créées en même temps que la contenante

  Lors de l'initialisation de l'instance contenante

- Avantages et inconvénients
  - On construit sur l'existant, plus grande modularité
  - Redondance et duplication de données (nombre de faces)
  - Souplesse et évolutivité (différents nombres de faces possible)

Agrégation (1)

- Généralisation de la composition, sans l'appartenance

  Deux objets indépendamment créés vont pouvoir être agrégés

```
1  public class City
2  {
3      private Citizen mayor;     // Bourgmestre de la ville
4      private String name;       // Nom de la ville
5
6      public City (String s)
7      {
8          name = s;
9      }
10
11     public void ChangeMayor (Citizen c)
12     {
13         mayor = c;
14     }
15 }
```

Agrégation (2)

- Généralisation de la composition, sans l'appartenance

  Deux objets indépendamment créés vont pouvoir être agrégés

- Suppression de l'objet contenant sans toucher aux contenus

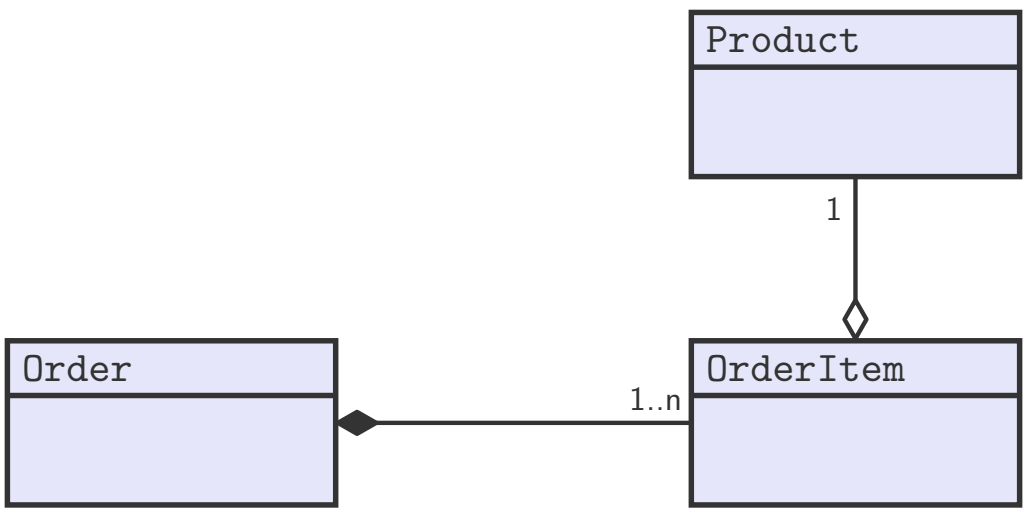
```
1  Citizen philippe = new Citizen ("Philippe", "Melotte");
2
3  City woluwe = new City ("Woluwé-Saint-Lambert");
4  woluwe.ChangeMayor (philippe);
```

Composition et agrégation

- Relation de contenance entre objets

  Objet(s) contenu(s) est(sont) dans un objet contenant (unique)

- Existence indépendante de l'objet contenu ou non
  - Oui dans le cas d'une agrégation (*owns-a*)
  - Non dans le cas d'une composition (*is-made-up-of*)



Relation uses

- Relation d'utilisation entre deux classes

  Beaucoup plus générale que composition et agrégation

- Plusieurs situations possibles d'utilisation
  - Recevoir un objet en paramètre
  - Renvoyer un objet
  - Utiliser un objet dans le corps d'une méthode

Association et dépendance

- L'association est une relation basée sur la référence

  Un objet conserve une référence vers un autre objet

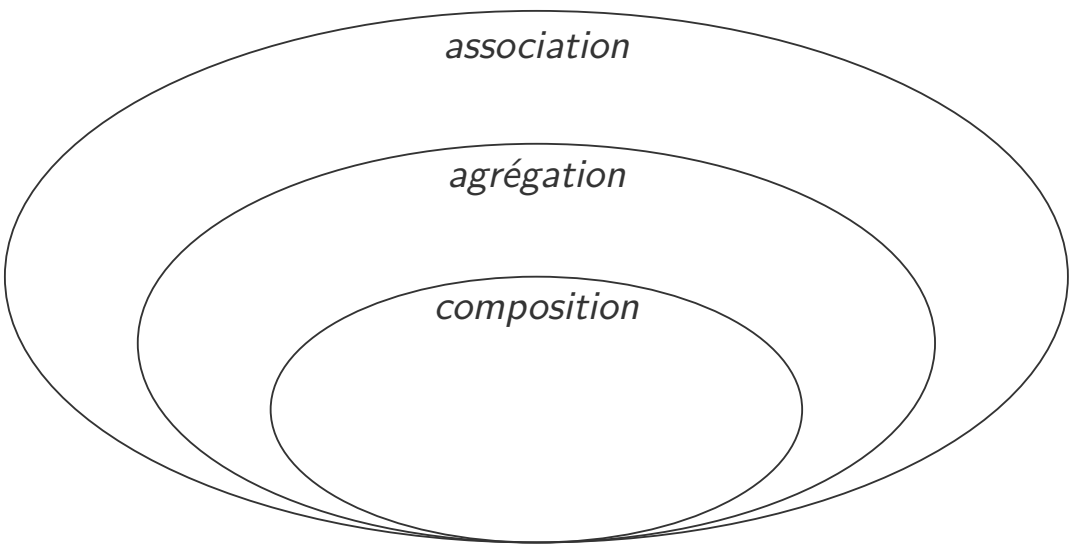
- Réception d'une référence suite à opération est dépendance

  Réception en paramètre, création locale par `new`

Comparaison des relations

- Classement des relations en fonction de leur force

  Association $<$ *Agrégation* $<$ *Composition*



	cycles de vie indépendants	*appartenance*
association	✔	✘
agrégation	✔	✔
composition	✘	✔

Couplage et cohésion

- Classes couplées si l'une dépend de l'implémentation de l'autre

  Une classe accède aux variables d'instance de l'autre...

- Cohésion d'une classe mesure son niveau d'indépendance

  Classe cohérente facilement maintenable et réutilisable

- Il faut **minimiser** $\downarrow$ le couplage et **maximiser** $\uparrow$ la cohésion

  Règle de bonne pratique en programmation orientée objet

Crédits

- https://www.flickr.com/photos/sharynmorrow/14549114
- https://www.flickr.com/photos/jetstarairways/6769120131
- https://www.flickr.com/photos/cameliatwu/6122062721
- https://en.wikipedia.org/wiki/File:D18_rhombicuboctahedron.JPG