Agrégation, composition, délégation et extension

Arnaud Labourel arnaud.labourel@univ-amu.fr

Section 1

Composition, agrégation et délégation

Exemple d'une classe Point

On considère la classe Point suivante :

```
public class Point {
    public final double x, y;
    public Point(double x, double y){this.x = x;
                                     this.y = y;}
    public double distanceTo(Point p){
        double dx = this.x - p.x;
        double dy = this.y - p.y;
        return Math.hypot(dx, dy);
    }
    public double getX(){ return x; }
    public double getY(){ return y; }
}
```

Réutilisation de Point : composition

Afin d'implémenter ses services, une instance peut créer des instances d'autres classes et conserver leurs références dans ses attributs.

```
public class Circle {
    private Point center;
    private double radius;

    public Circle(double x, double y, double radius){
        this.center = new Point(x, y);
        this.radius = radius;
    }

    public double getCenterX(){ return center.getX(); }
    public double getCenterY(){ return center.getY(); }
    public double getRadius(){ return radius; }
}
```

Définition composition

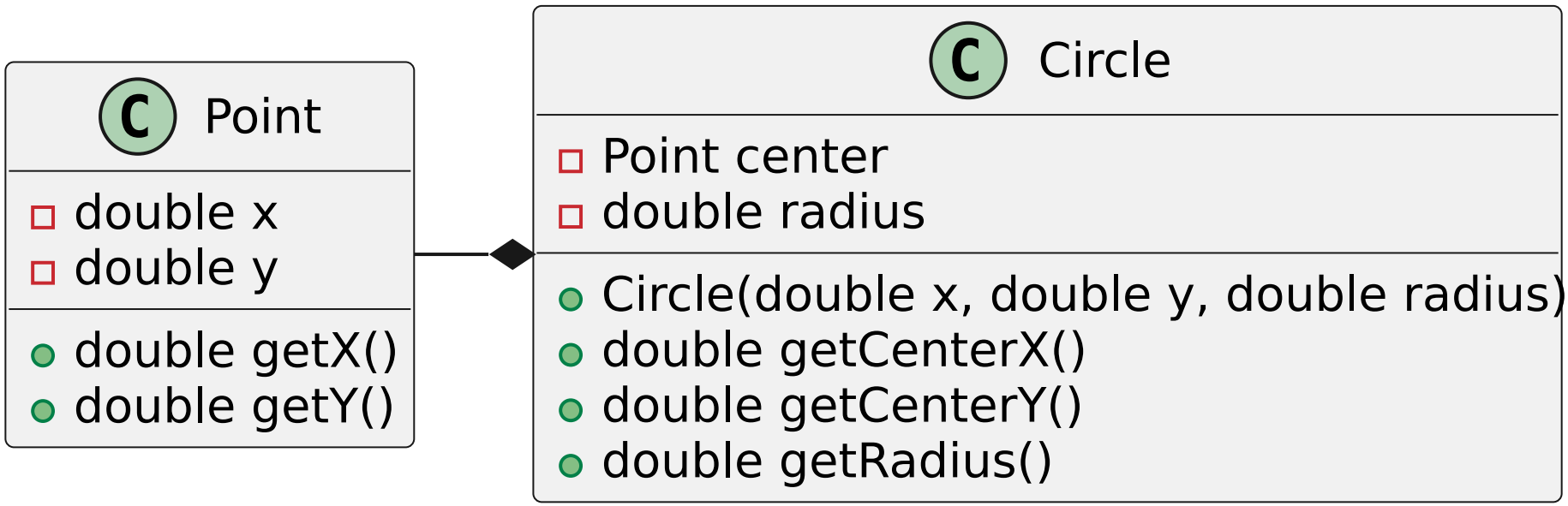
Composition d'une classe `Partie` dans une classe `Classe`

- une (ou plus) instance(s) de `Partie` est stockée dans une instance de `Classe`
- Chaque instance de `Partie` appartient à une seule instance de `Classe`
- Chaque instance de `Partie` a sa durée de vie (construction) gérée par l'instance de `Classe` qui la contient
- Les instances de `Partie` n'ont pas connaissance de l'existence des instances de `Classe`

Exemple: Pièces d'une maison

Une maison contient une ou plusieurs pièces. La durée de vie d'une pièce est contrôlée par la maison, car elle n'existerait pas sans la maison.

Diagramme composition



Variante de Circle : agrégation

Une instance peut simplement posséder des références vers des instances d'une autre classe :

```
public class Circle {
    private Point center, point;
    public Circle(Point center, Point point){
        this.center = center;
        this.point = point;
    }
    public Point getCenter(){ return center; }
    public double getRadius(){
        double dx = center.x - point.x;
        double dy = center.y - point.y;
        return Math.hypot(dx, dy);
    }
}
```

Définition agrégation

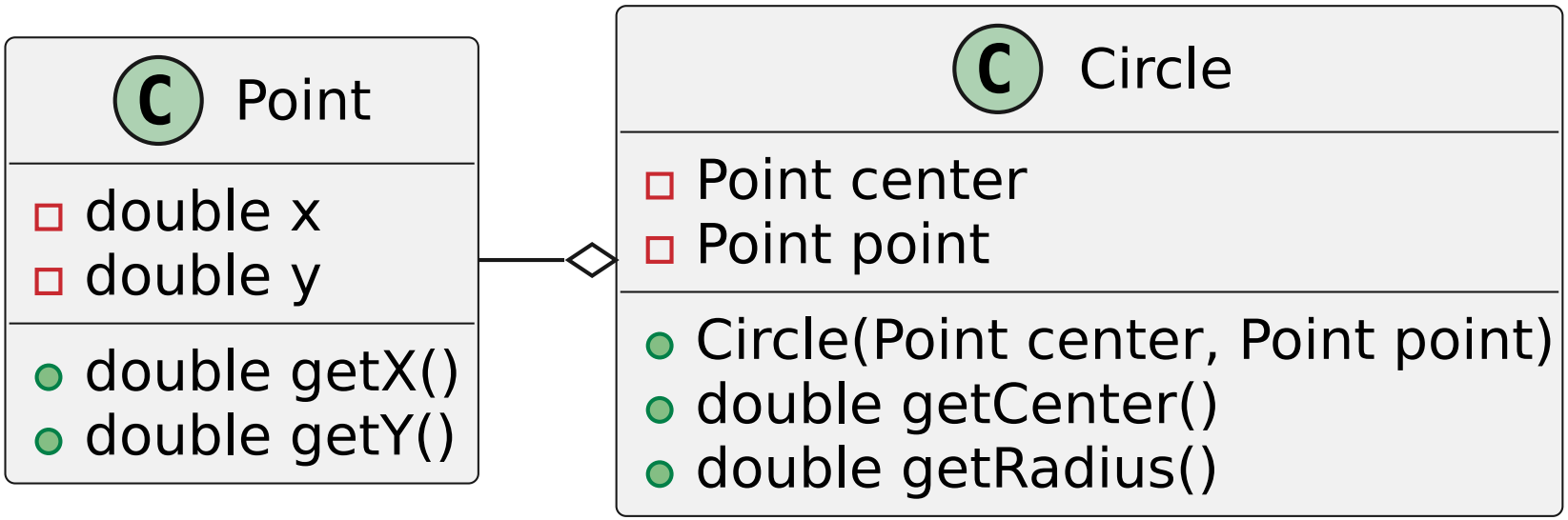
Agrégation d'une classe `Partie` dans une classe `Classe`

- une (ou plus) instance(s) de `Partie` est stockée dans une instance de `Classe`
- Chaque instance de `Partie` peut appartenir à plusieurs instances de `Classe`
- La durée de vie des instances de `Partie` n'est pas gérée par les instances de `Classe` qui la contient
- Les instances de `Partie` n'ont pas connaissance de l'existence des instances de `Classe`

Exemple: Bloc de construction d'une maison jouet

On a une maison jouet construite à partir de blocs. Vous pouvez la démonter, mais les blocs resteront.

Diagramme agrégation



Délégation

Délégation du calcul de la distance à l'instance `center` de la classe `Point` :

```
public class Circle {
    private Point center, point;

    public Circle(Point center, Point point){
        this.center = center;
        this.point = point;
    }
    public Point getCenter(){ return center; }
    public double getRadius(){
        return center.distanceTo(point);
    }
}
```

Définition délégation

Délégation d'une opération d'une classe `Classe` à une instance d'une classe `Delegate`

Une instance de `Classe` confie la résolution d'une de ses opérations (méthodes) à une instance de `Delegate`.
L'objet endossant cette responsabilité est nommé le *delegate* (ou délégué). Le rôle d'un *delegate* est de définir du comportement laissé à sa charge.

Exemple : confier le fait de faire un café à un stagiaire

Votre patron vous a demandé de lui faire un café, vous l'avez plutôt fait faire par un stagiaire.

Section 2

Classes abstraites et extension

Classe ListSum

Supposons que nous ayons la classe suivante :

```
public class ListSum {
  private int[] list = new int[10];
  private int size = 0;
  public void add(int value) {
    list[size] = value;
    size++;
  }
  public int eval() {
    int result = 0;
    for (int i = 0; i < size; i++)
      result += list[i];
    return result;
  }
}
```

Classe ListProduct

Supposons que nous ayons aussi la classe suivante (très similaire) :

```
public class ListProduct {
  private int[] list = new int[10];
  private int size = 0;
  public void add(int value) {
    list[size] = value;
    size++;
  }
  public int eval() {
    int result = 1;
    for (int i = 0; i < size; i++)
      result *= list[i];
    return result;
  }
}
```

Refactoring pour isoler la répétition

Les deux classes sont très similaires et il y a de la répétition de code.

Il est possible de *refactorer* (réécrire le code) les classes précédentes de façon à isoler les différences dans des méthodes :

```
public class ListSum {
  /* attributs, constructeur et méthode add. */
  public int eval() {
    int result = neutral();
    for (int i = 0; i < size; i++)
      result = compute(result, list[i]);
    return result;
  }
  private int neutral() { return 0; }
  private int compute(int a, int b) { return a+b; }
}
```

Refactoring pour isoler la répétition

Il est possible de *refactorer* les classes précédentes de façon à isoler les différences dans des méthodes :

```
public class ListProduct {
  /* attributs, constructeur et méthode add. */
  public int eval() {
    int result = neutral();
    for (int i = 0; i < size; i++)
      result = compute(result, list[i]);
    return result;
  }
  private int neutral() { return 1; }
  private int compute(int a, int b) { return a*b; }
}
```

Comment éviter la répétition ?

Après la *refactorisation* du code :

- seules les méthodes `neutral` et `compute` diffèrent
- il serait intéressant de pouvoir supprimer les duplications de code

Deux solutions :

- La délégation en utilisant une interface et l'agrégation
- L'extension et les classes abstraites

Solution délégation : interface Operator

Nous allons externaliser les méthodes `neutral` et `compute` dans deux nouvelles classes. Elles vont implémenter l'interface `Operator` :

```
public interface Operator {
  public int neutral();
  public int compute(int a, int b);
}

public class Sum implements Operator {
  public int neutral() { return 0; }
  public int compute(int a, int b) { return a+b; }
}

public class Product implements Operator {
  public int neutral() { return 1; }
  public int compute(int a, int b) { return a*b; }
}
```

Délégation

Les classes `ListSum` et `ListProduct` sont fusionnées dans une unique classe `List` qui délègue les calculs à un objet qui implémente l'interface `Operator` :

```
public class List {
  /* attributs et méthode add */
  private Operator operator;
  public List(Operator operator){
    this.operator = operator;
  }
  public int eval() {
    int result = operator.neutral();
    for (int i = 0; i < size; i++)
      result = operator.compute(result, list[i]);
    return result;
  }
```

Délégation

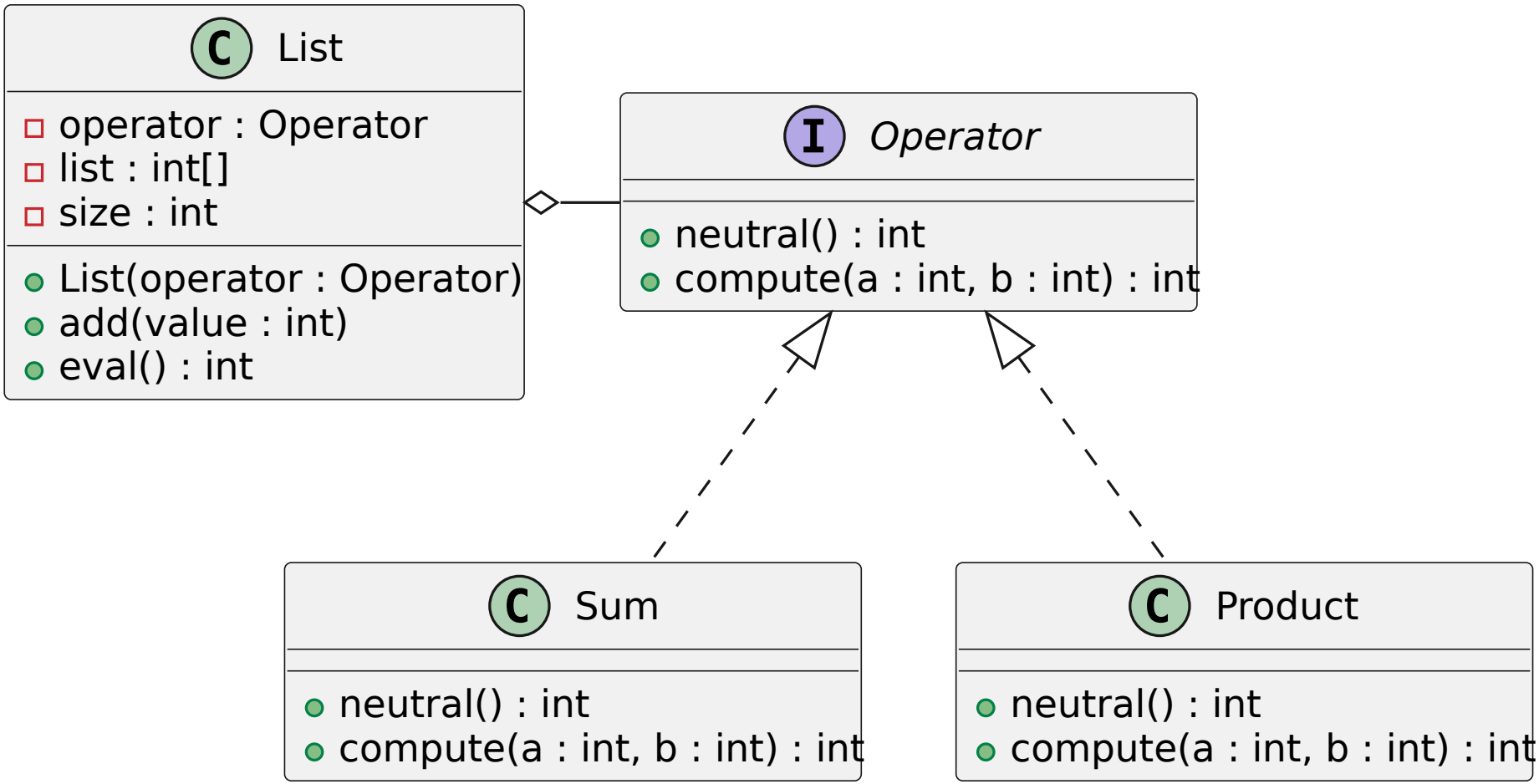
Utilisation des classes ListSum et ListProduct :

```
ListSum listSum = new ListSum();
listSum.add(2); listSum.add(3);
System.out.println(listSum.eval());
ListProduct listProduct = new ListProduct();
listProduct.add(2); listProduct.add(3);
System.out.println(listProduct.eval());
```

Utilisation après la *refactorisation* du code :

```
List listSum = new List(new Sum());
listSum.add(2); listSum.add(3);
System.out.println(listSum.eval());
List listProduct = new List(new Product());
listProduct.add(2); listProduct.add(3);
System.out.println(listProduct.eval());
```

Diagramme de la solution avec délégation



Solution utilisant une classe abstraite

```
public abstract class List {
  private int[] list = new int[10];
  private int size = 0;
  public void add(int value) {
    list[size] = value; size++;
  }
  public int eval() {
    int result = neutral();
    // utilisation d'une méthode abstraite
    for (int i = 0; i < size; i++)
      result = compute(result, list[i]); // idem
    return result;
  }
  public abstract int neutral(); // méthode abstraite
  public abstract int compute(int a, int b); // idem
}
```

Mot-clé `abstract`

Classe abstraite

- On peut mettre `abstract` devant le nom de la classe à sa définition pour signifier qu'une classe est abstraite.
- Une classe est abstraite si des méthodes ne sont pas implémentées.
  $\Rightarrow$ Classe abstraite = classe avec des méthodes abstraites
- Tout comme pour une interface, une classe abstraite n'est pas instanciable.

Méthode abstraite

- `abstract` devant le nom du type de retour de la méthode à sa définition pour signifier qu'une méthode est abstraite.
- Méthode abstraite = méthode sans code, juste la signature (type du retour et des paramètres) est définie

Classes abstraites et extension

Tout comme pour les interfaces, il n'est pas possible d'instancier une classe abstraite :

```
List list = new List(); // erreur
System.out.println(list.eval()) // car que faire ici ?
```

Nous allons étendre la classe `List` afin de récupérer les attributs et les méthodes de `List` et définir le code des méthodes abstraites :

```
public class ListSum extends List {
  public int neutral() { return 0; }
  public int compute(int a, int b) { return a+b; }
}
```

Classes abstraites et extension

La classe `ListSum` n'est plus abstraite, toutes ses méthodes sont définies :

- les méthodes `add` et `eval` sont définies dans `List` et `ListSum` hérite du code de ses méthodes.
- les méthodes `neutral` et `compute` qui étaient abstraites dans `List` sont définies dans `ListSum`.

On peut donc instancier la classe `ListSum` :

```
ListSum listSum = new ListSum();
listSum.add(3);
listSum.add(7);
System.out.println(listSum.eval());
```

Classes abstraites et extension

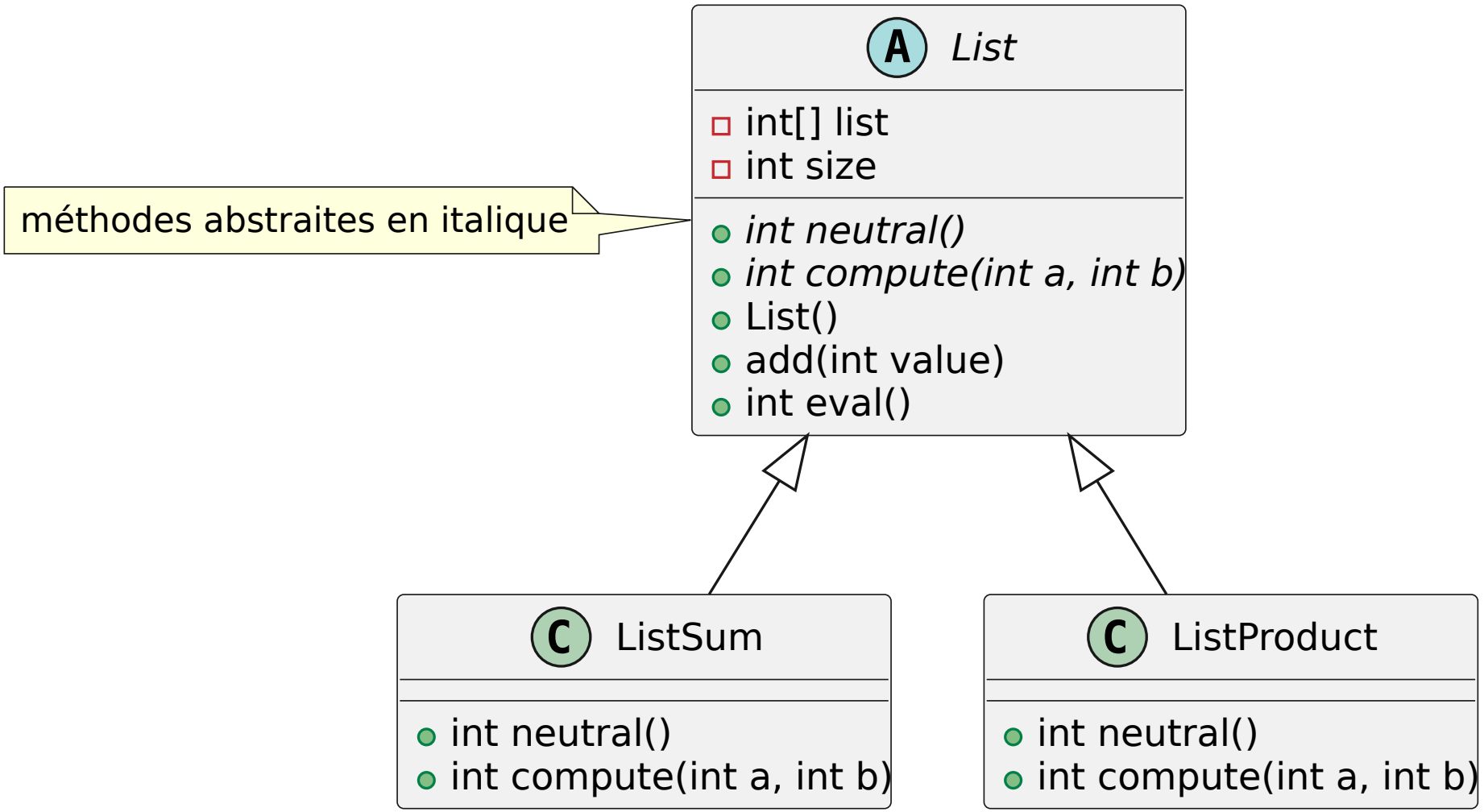
On peut procéder de manière similaire pour créer une classe `ListProduct`

```
public class ListProduct extends List {
  public int neutral() { return 1; }
  public int compute(int a, int b) { return a*b; }
}
```

La classe `ListProduct` n'est plus abstraite, toutes ses méthodes sont définies :

```
ListProduct listProduct = new ListProduct();
listProduct.add(3);
listProduct.add(7);
System.out.println(listProduct.eval());
```

Diagramme de la solution avec extension



Généralisation de l'extension aux classes non-abstraites

Plus généralement, l'extension permet de créer une classe en :

- conservant les services (attributs et méthodes) d'une autre classe ;
- ajoutant de nouveaux services (attributs et méthodes) ;
- redéfinissant certains services (méthodes).

En Java :

- On utilise le mot-clé `extends` pour étendre une classe ;
- Une classe ne peut étendre directement qu'une seule classe.

Important

Il est toujours préférable de privilégier l'implémentation à l'extension.

Extension pour ajouter des nouveaux services

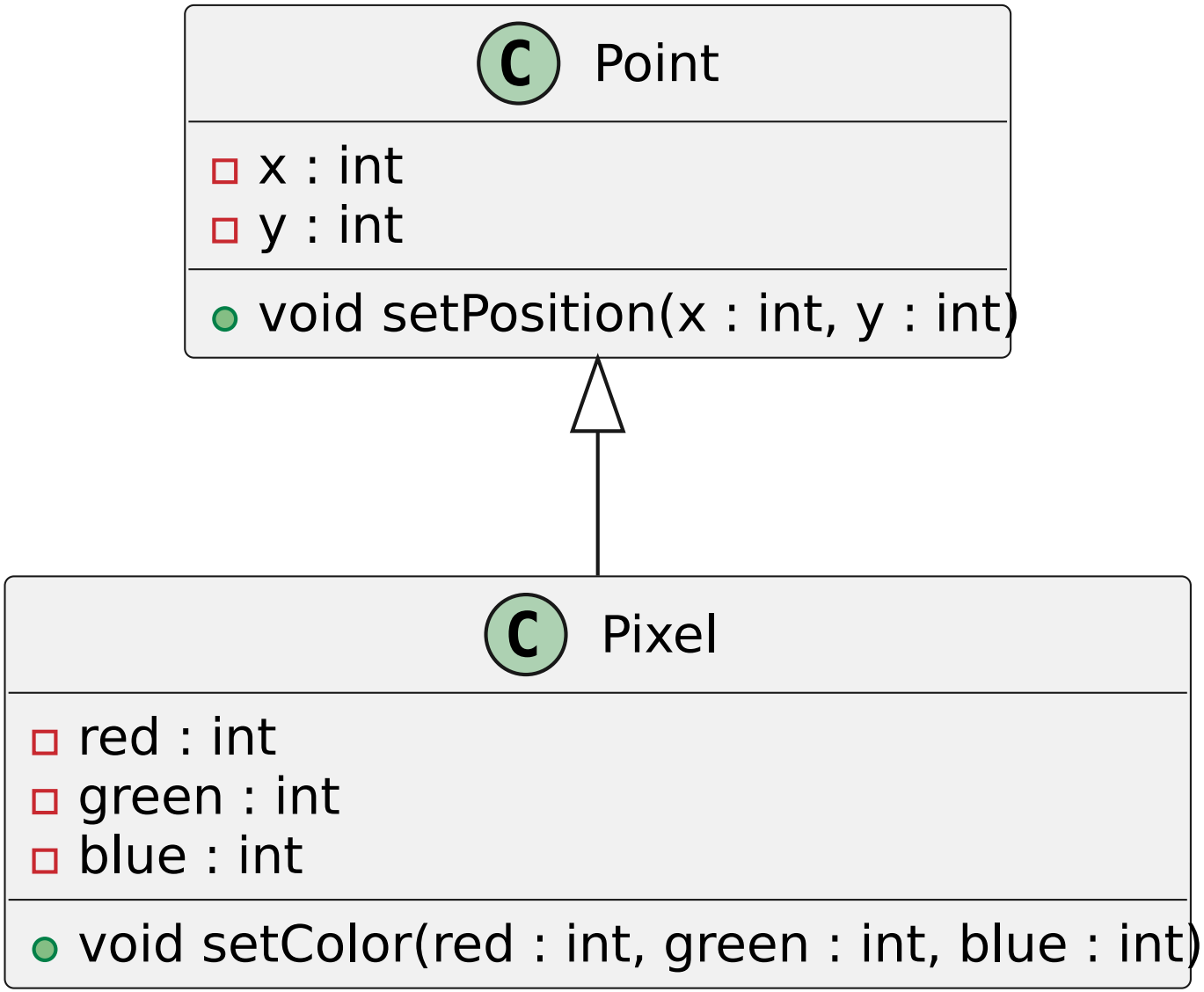
Supposons que nous ayons la classe `Point` suivante :

```
public class Point {
  private int x, y;
  public void setPosition(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

Il est possible d'ajouter de nouveaux services en utilisant l'extension :

```
public class Pixel extends Point {
  private int r, g, b;
  public void setColor(int r, int g, int b) {
    this.r = r; this.g = g; this.b = b; }
}
```

Diagramme


Point
x : int
y : int
void setPosition(x : int, y : int)
Pixel
red : int
green : int
blue : int
void setColor(red : int, green : int, blue : int)

Extension pour ajouter des nouveaux services

Les services (méthodes et attributs) de `Point` sont disponibles dans `Pixel` :

```
Pixel pixel = new Pixel();
pixel.setPosition(4,8);
System.out.println(pixel.x); // → 4
System.out.println(pixel.y); // → 8
pixel.setColor(200, 200, 120);
```

Évidemment, les services de `Pixel` ne sont pas disponibles dans `Point` :

```
Point point = new Point();
point.setPosition(4, 8);
System.out.println(point.x); // → 4
System.out.println(point.y); // → 8
point.setColor(200, 200, 120); // impossible !
```

Redéfinition de méthode

Supposons que nous ayons la classe Point suivante :

```
public class Point {
  public int x, y;
  public void setPosition(int x, int y) {
    this.x = x; this.y = y;
  }
  public void clear() {
    x = 0; y = 0;
  }
}
```

Redéfinition de méthode

Il est possible de redéfinir la méthode `clear` dans `Point` :

```
public class Pixel extends Point {
  public int red, green, blue;
  public void setColor(int r, int g, int b) {
    this.r = r; this.g = g; this.b = b;
    }
  public void clear(){
    x = 0; y = 0;
    red = 0; green = 0; blue = 0;
  }
}
```

Redéfinition avec super

```
public class Point {
  public int x, y;
  public void setPosition(int x, int y) {
    this.x = x; this.y = y;
  }
  public void clear() {
    setPosition(0, 0);
  }
}
```

Le mot-clé super permet d'utiliser la méthode clear de Point :

```
public class Pixel extends Point { public int r, g, b;
public void setColor(int red, int green, int blue) {
  this.red = red; this.green = green; this.blue = blue; }
public void clear() { super.clear(); setColor(0, 0, 0); }
```

Le mot-clé super

```
public class Point {
  public int x, y;
  public void setPosition(int x, int y) {
    this.x = x; this.y = y; }
}
```

Si la méthode n'a pas été redéfinie, le mot clé super est inutile :

```
public class Pixel extends Point {
  public int red, green, blue;
  public void setColor(int red, int green, int blue) {
    this.red = red; this.green = green; this.blue = blue;
  }
  public void clear() {
    /*super.*/setPosition(0, 0);
    setColor(0, 0, 0);
  }
```

Les constructeurs et le mot-clé super

```
public class Point {
  public int x, y;
  public Point(int x, int y) {
    this.x = x; this.y = y;
  }
}
```

La classe `Point` n'a pas de constructeur sans paramètre, il faut donc indiquer comment initialiser la partie de Pixel issue de la classe `Point` :

```
public class Pixel extends Point {
  public int red, green, blue;
  public Pixel(int x, int y, int r, int g, int b) {
    super(x, y); // appel du constructeur de Point
    this.red = r; this.green = g; this.blue = b;
  }
}
```

Les constructeurs

Supposons que nous ayons la classe `Point` suivante :

```
public class Point {
  public int x, y;
  public Point() { x = 0; y = 0; }
  public Point(int x, int y) { this.x = x; this.y = y; }
}
```

Par défaut, le constructeur sans paramètre est appelé :

```
public class Pixel extends Point {
  public int red, green, blue;
  public Pixel(int r, int g, int b) {
    // appel du constructeur sans paramètre de Point
    this.red = r; this.green = g; this.blue = b;
  }
}
```

Les constructeurs

```
public class Point {
  public int x, y;
  //public Point() { x = 0; y = 0; }
  public Point(int x, int y) { this.x = x; this.y = y; }
}
```

Ici, vous devez préciser les paramètres du constructeur avec super :

```
public class Pixel extends Point {
  public int red, green, blue;
  public Pixel(int r, int g, int b) {
    // erreur de compilation
    // (aucun constructeur sans paramètre)
    this.red = r; this.green = g; this.blue = b;
  }
}
```

Transtypages et polymorphisme

Aucune méthode ou attribut ne peut être supprimée lors d'une extension. Par exemple, Pixel possède toutes les attributs et méthodes de `Point` (même si certaines méthodes ont pu être redéfinies).

Par conséquent, l'**upcasting** est toujours autorisé :

```
Point point = new Pixel();
point.setPosition(2,4);
System.out.println(point.x + " " + point.y);
point.clear();
```

Remarques

- Le code exécuté lors d'un appel de méthode est déterminé à l'exécution en fonction de la référence présente dans la variable.
- Le typage des variables permet de vérifier à la compilation l'existence des attributs et des méthodes.

Upcasting (transtypage ascendant)

Définition d'upcasting

Considérer une instance d'une classe comme une instance d'une de ses **super-classes**, c'est-à-dire :

- une classe étendue par la classe
- une interface implémentée par la classe

Remarques

- Après l'*upcasting*, les services supplémentaires de la classe (méthodes et attributs non disponibles dans la super-classe) ne sont plus accessibles.
- Lors de l'appel des méthodes, si la classe a redéfini (*overrides*) la méthode, c'est le code de la classe qui est exécuté.

La classe Object

Par défaut, les classes étendent la classe `Object` de Java. Par conséquent, l'*upcasting* vers la classe `Object` est toujours possible :

```
Pixel pixel = new Pixel();
Object object = pixel;
Object[] array = new Object[10];
for (int i = 0; i < t; i++) {
  if (i%2==0) array[i] = new Point();
  else array[i] = new Pixel();
}
```

Notez que `object.setPosition(2,3)` n'est pas autorisé dans le code ci-dessus, car la classe `Object` ne possède pas la méthode `setPosition` et seul le type de la variable compte pour déterminer si l'appel d'une méthode ou l'utilisation d'un attribut est autorisé.

Méthodes de la classe `Object`

- `protected Object clone()`: Creates and returns a copy of this object.
- `boolean equals(Object obj)`: Indicates whether some other object is “equal to” this one.
- `Class<?> getClass()`: Returns the runtime class of this Object.
- `int hashCode()`: Returns a hash code value for the object.
- `String toString()`: Returns a string representation of the object.

Il y a aussi des méthodes `wait` et `notify` pour attendre sur un objet et réveiller des *thread* en attentes sur un objet.

La méthode toString() de la classe Object

Par transitivité de l'extension, toutes les méthodes et attributs de la classe Object sont disponibles sur toutes les instances :

```
Object object=new Object(); Point point=new Point(2,3);
System.out.println(object.toString());
// → java.lang.Object@19189e1
System.out.println(point.toString());
// → test.Point@7c6768
```

La méthode toString est utilisée par Java pour convertir une référence en chaîne de caractères :

```
Object object = new Object();
Point point = new Point(2,3);
String string = object+";"+point;
System.out.println(string);
// → java.lang.Object@19189e1;test.Point@7c6768
```

La méthode `toString()` et le polymorphisme (1/3)

Évidemment, il est possible de redéfinir la méthode `toString` dans `Point` :

```
public class Point {
  // extends Object de manière implicite
  public int x, y;
  public Point(int x, int y) {
    this.x = x; this.y = y;
  }

  public String toString() {
    return "("+x+","+y+")";
  }
}
```

La méthode toString() et le polymorphisme (2/3)

Évidemment, il est aussi possible de redéfinir la méthode toString dans Pixel :

```
public class Pixel extends Point {
  public int red, green, blue;
  public Pixel(int x, int y, int r, int g, int b) {
    super(x, y);
    this.red = r; this.green = g; this.blue = b;
  }
  public String toString() {
    return super.toString()
      + ", #"+red+","+green+","+blue;
  }
}
```

La méthode `toString()` et le polymorphisme (3/3)

Le code de la méthode est appelée correspond toujours à la dernière redéfinition de la méthode par rapport à la classe réelle de l'objet (pas la classe de la variable).

```
Point point = new Point(2,3);
Object objectPoint = point;
Pixel pixel = new Pixel(0, 0, 10, 20, 30);
Point pointPixel = pixel;
Object objectPixel = pixel;
System.out.println(point); // → (2,3)
System.out.println(objectPoint); // → (2,3)
System.out.println(pixel); // → (0,0), #10,20,30
System.out.println(pointPixel); // → (0,0), #10,20,30
System.out.println(objectPixel); // → (0,0), #10,20,30
```