Programmation Objet Avancée

Kevin Heraud

Cours 1 : rappels, héritage et polymorphisme

Département Informatique
Université de Tours

Présentation

- Worldline
- Dev, Lead dev, architecte,... chez ITA
- heraud.kevin@gmail.com
- aicfr#0766

Objectifs du cours

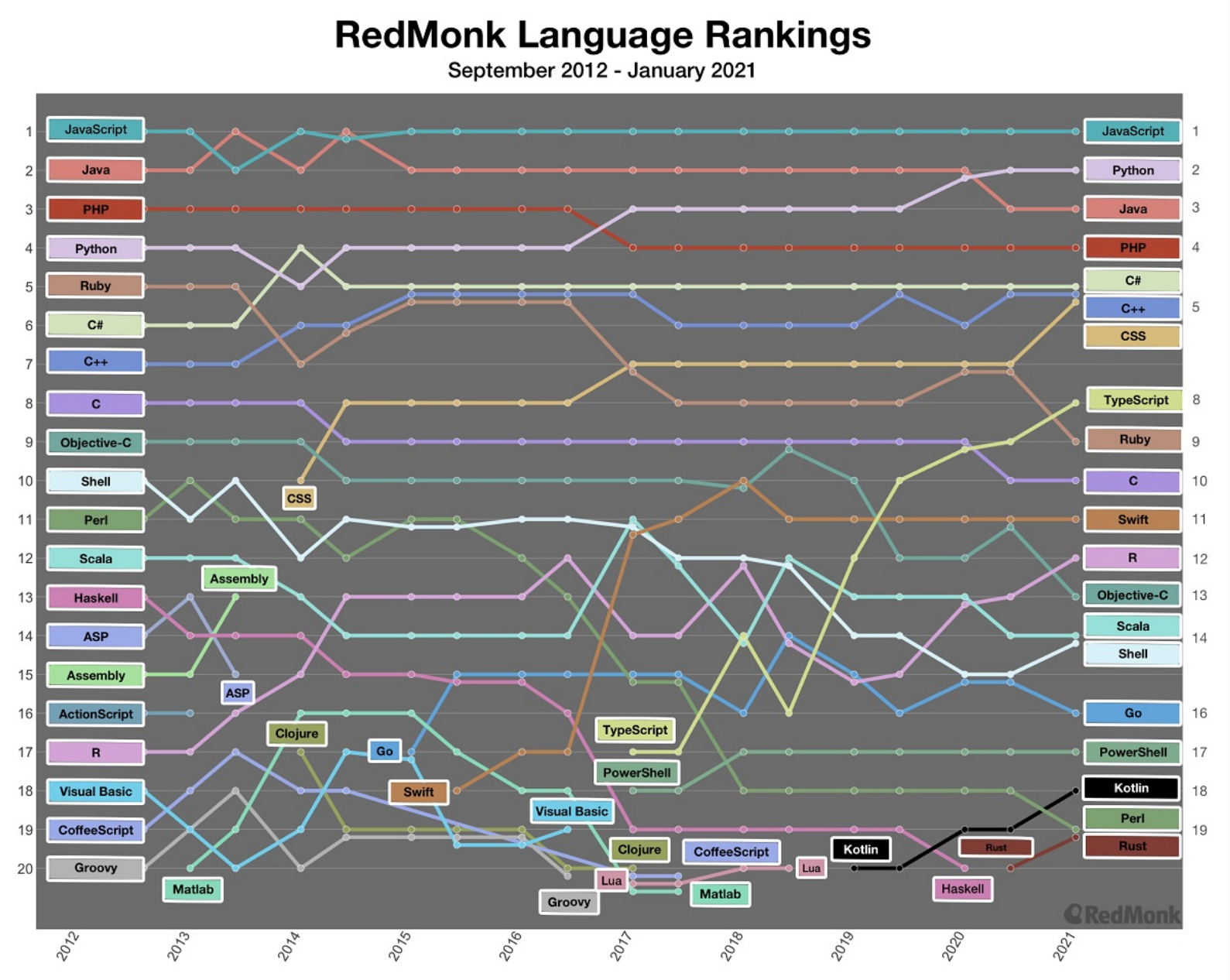
- rappels de L1 : paradigme de programmation objet, classe, objet
- comprendre les notions avancées d'héritage, de classes abstraites, d'interface
- savoir utiliser les structures comme les `ArrayList` et les `HashMap`
- manipuler les exceptions, les fichiers et les bases de données
- Implémenter une interface graphique simple

Évaluation des connaissances

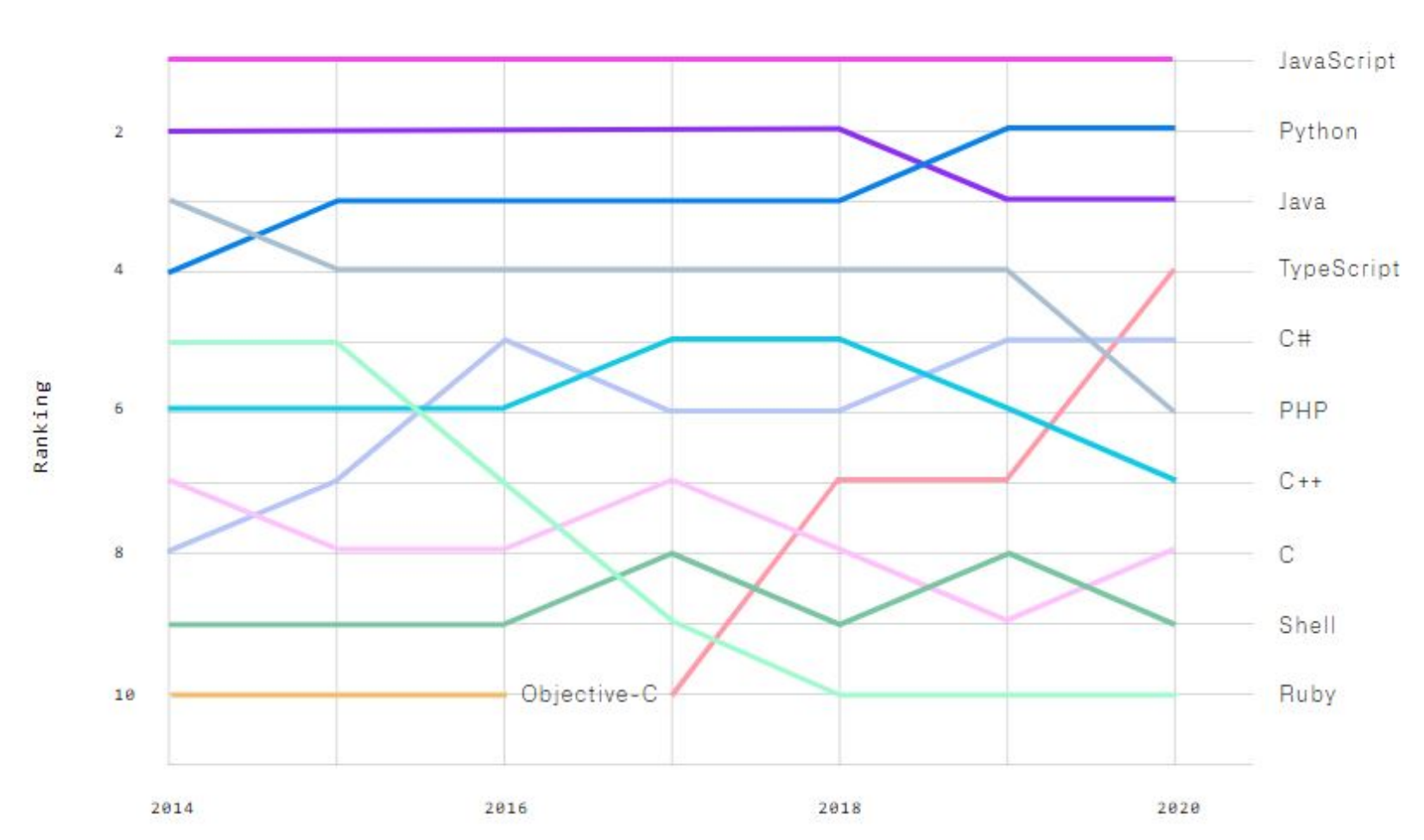
- 2 TP notés sur les 1ères parties du cours : semaines 4 et 8
- 1 projet commun avec le *Génie Logiciel* en TP : suivi séances semaines 6 et 10

$$\text{note finale} = 0.6 \times (\frac{1}{2}(TP_1 + TP_2)) + 0.4 \times \text{projet}$$

Pourquoi étudier Java (source : redmonk) ?



Pourquoi étudier Java (source : github.com) ?



Plan du cours

Programmation Orientée Objet (POO)

Héritage

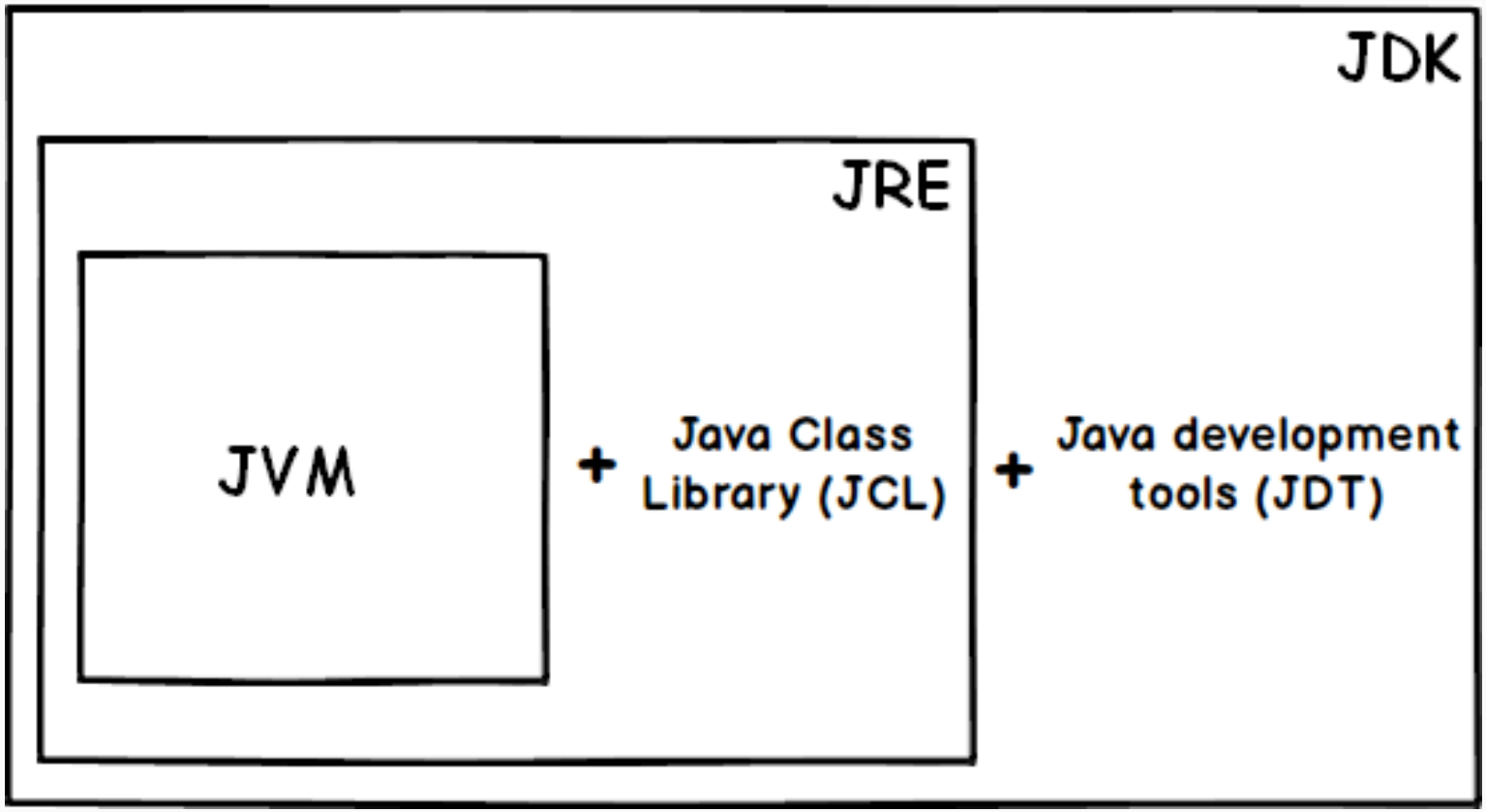
Surcharge et redéfinition de méthodes

Polymorphisme

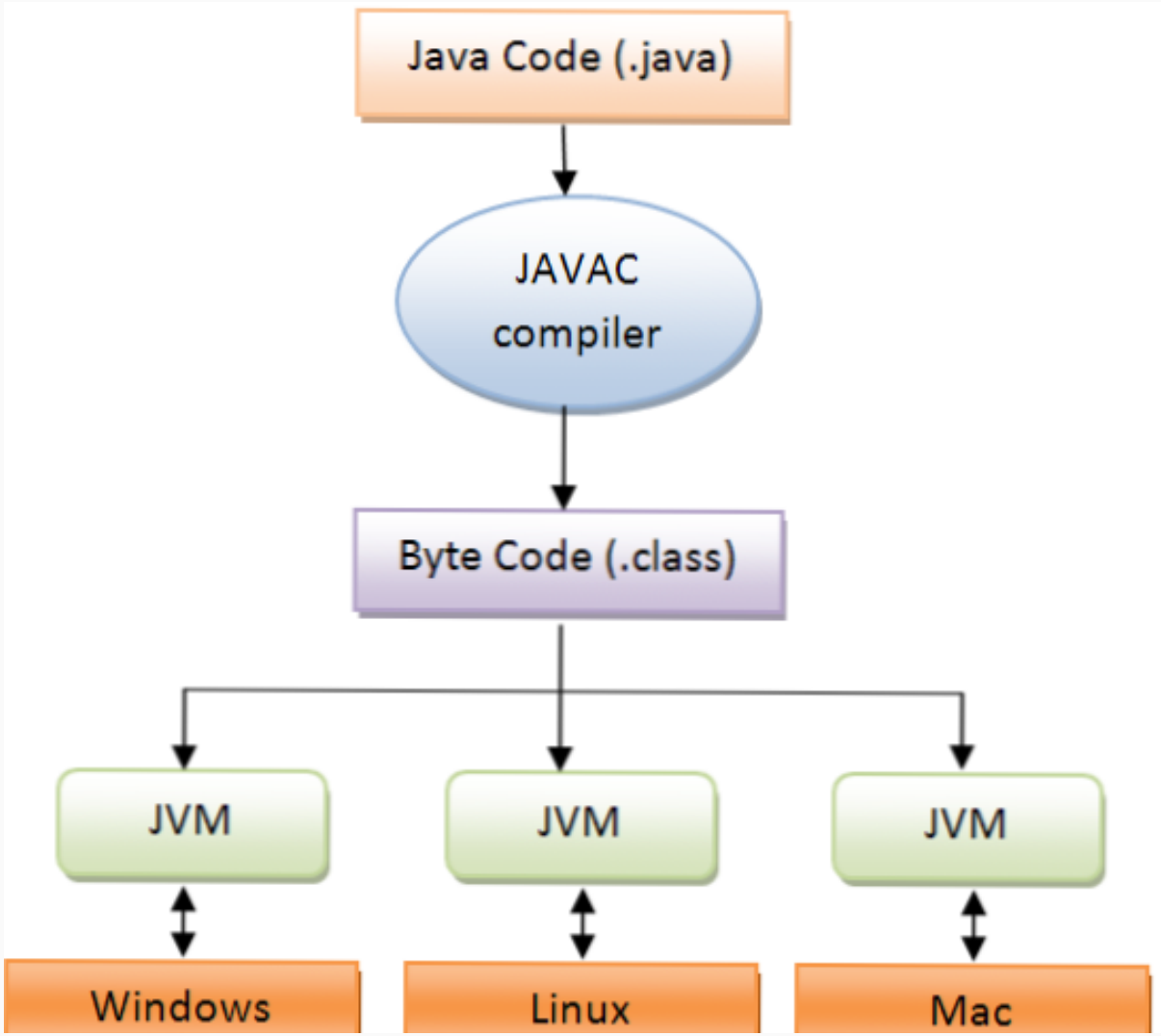
Programmation Orientée Objet (POO)

- la programmation orientée objet (POO) représente un programme sous la forme d'un ensemble d'objets qui interagissent pour résoudre un problème
- ce paradigme est proche de la représentation du monde réel, dans lequel des entités autonomes sont en relation

JVM, JRE, JDK, kézako ?



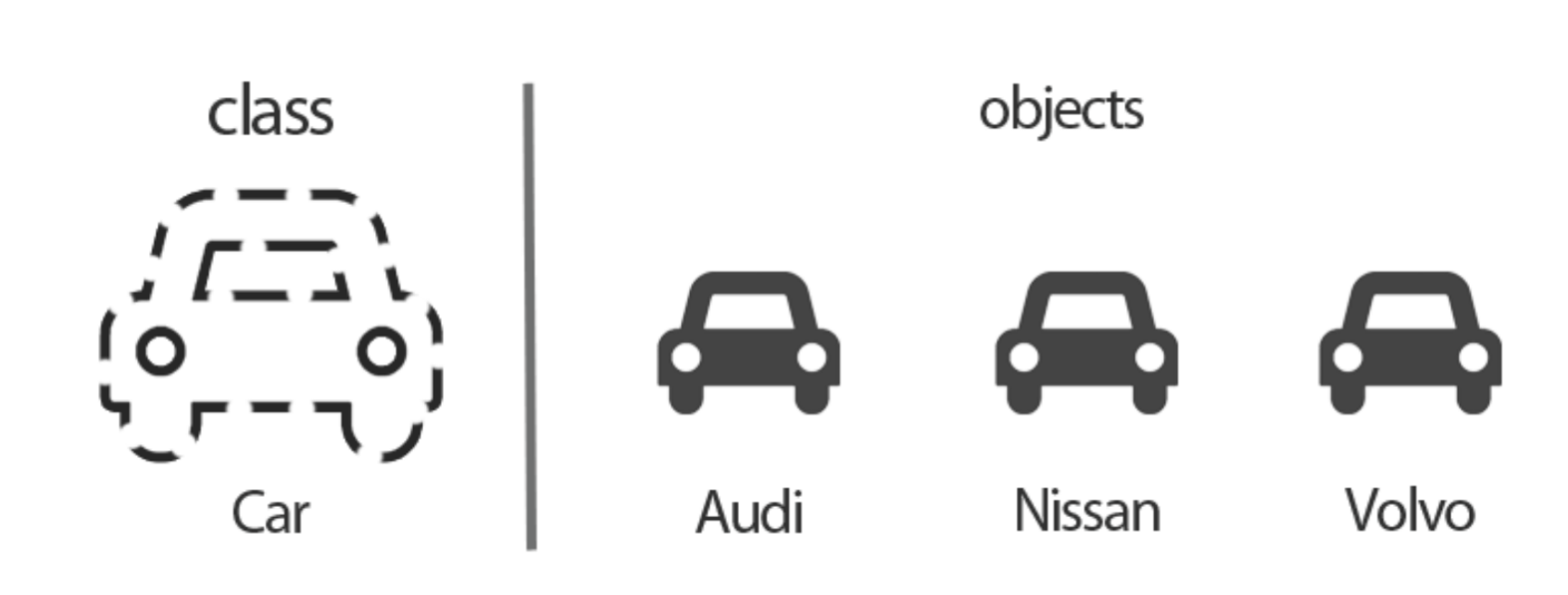
Bytecode Java



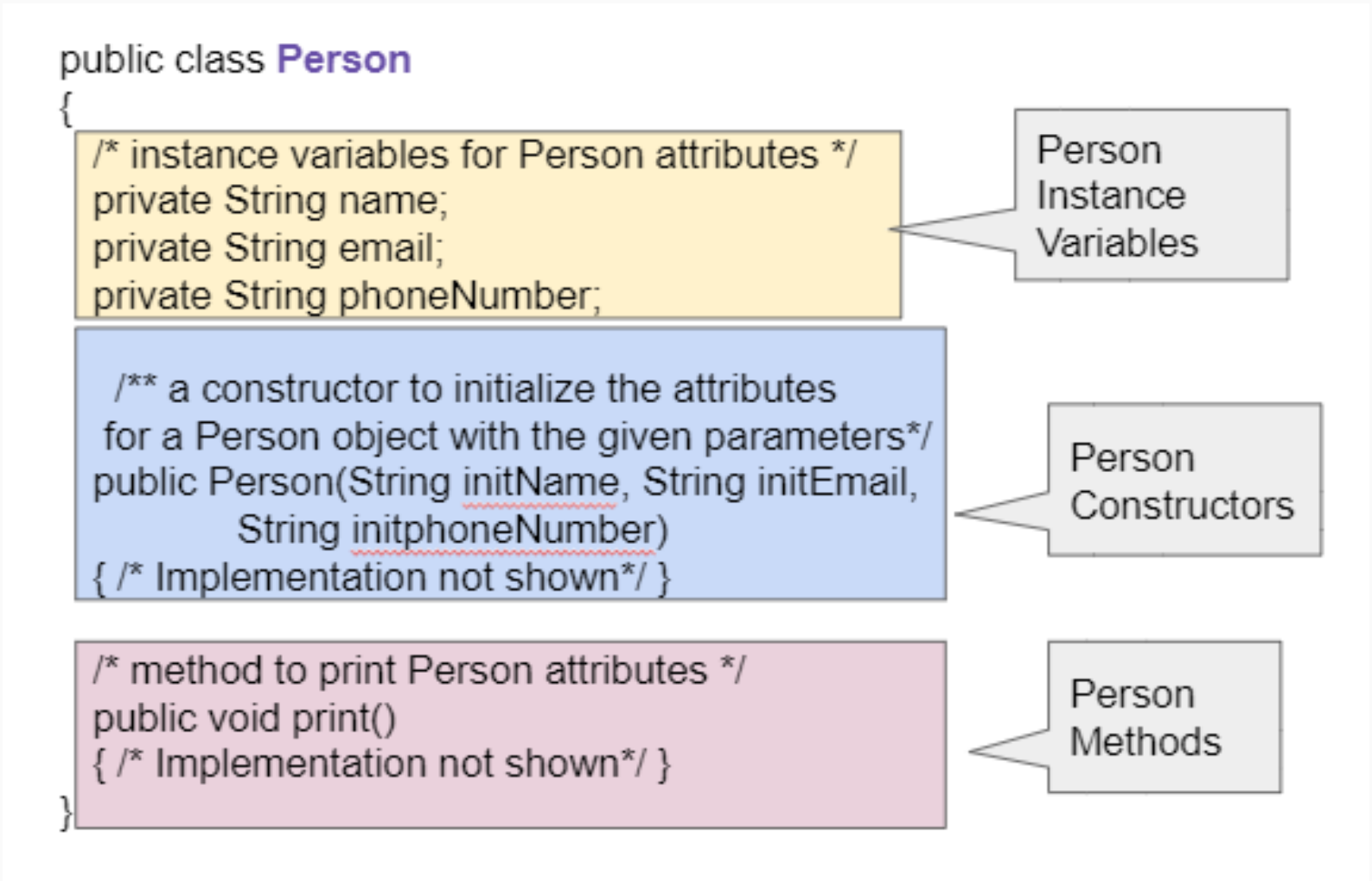
Relation entre classe et objet

- à la différence de la programmation **procédurale** qui repose sur une décomposition en sous-programmes (ou procédures) d'un programme
- la POO repose sur la définition de **classes** qui regroupent
  - des **champs** qui sont les données manipulées par la classe
  - des **méthodes** qui permettent d'interagir avec les données et/ou d'autres objets
- Une **classe** est un "plan" défini à partir duquel des objets sont créés
- un **objet** correspond à l'instanciation en mémoire d'une **classe**. Celui-ci est composé :
  - d'un **état** : Il est représenté par les attributs d'un objet
  - d'un **comportement** : Il est représenté par les méthodes d'un objet

Classe VS Objet



Anatomie d'une classe (Plain Old Java Object)



Exemple de la classe Carre

```
1  public class Carre {
2   private int largeur;
3
4   public Carre(int _largeur) {
5    largeur = _largeur;
6   }
7   public int perimetre() {
8    return 4 * largeur;
9   }
10  public int surface() {
11   return largeur * largeur;
12  }
13  public void agrandir(int coef) {
14   largeur = coef * largeur;
15  }
16 }
```

Explications

- `public class Carre {...}` définit une classe accessible de tous (`public`) nommée `Carre`. Les accolades `{` et `}` encadrent le bloc de code de la classe
- il y a **encapsulation** du champ qui est `private` : il n'est visible que des méthodes de la classe
- la méthode `public Carre(...)` est le **constructeur** qui définit comment créer et initialiser un objet de type `Carre`
- `perimetre()`, `surface()` et `agrandir(coef)` sont des **méthodes** de la classe `Carre`

Création d'objets de type Carre

- création d'un carré `c1` de largeur 10
- création d'un carré `c2` de largeur 5

```
1 Carre c1 = new Carre(10);
2 Carre c2 = new Carre(5);
```

- calcul du périmètre du carré `c1` avant et après agrandissement
- calcul de la surface du carré `c2`

```
1 int peri = c1.perimetre();
2 c1.agrandir(2);
3 int peri2 = c1.perimetre();
4 System.out.println("Avant = " + peri + ", apres = " + peri2);
5 System.out.println("Surface = " + c2.surface());
```

Variables d'instances, de classe, constantes

- Les données d'une classe sont contenues dans des variables nommées **propriétés** ou **attributs**.
- On distingue 3 types de variables

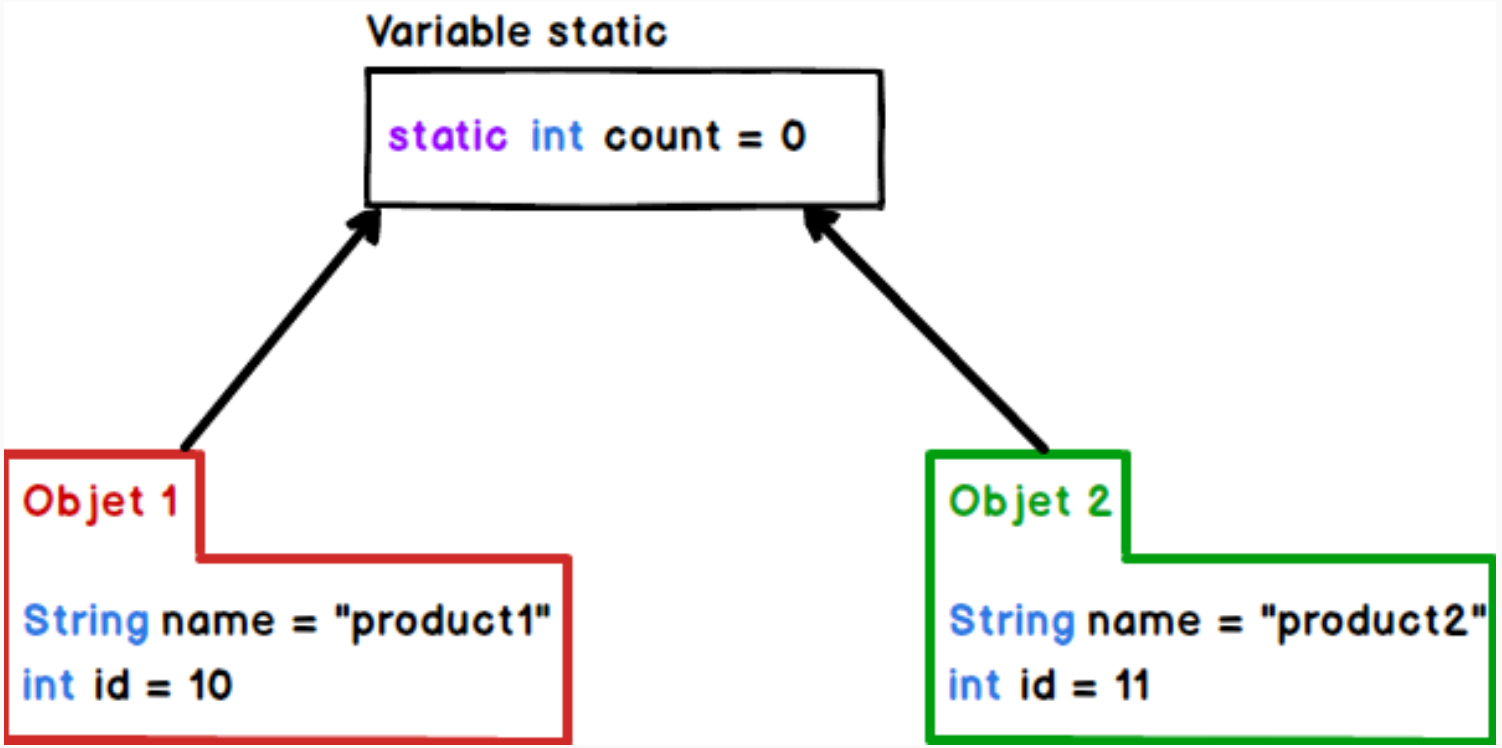
1. Les **variables d'instances** qui nécessitent simplement une déclaration de la variable dans le corps de la classe. Chaque instance de la classe a accès à sa propre occurrence de la variable.
2. Les **variables de classes** définies avec le mot-clé `static`. Toutes les instances de la classe partagent la même variable
3. Les **constantes** sont définies avec le mot clé `final` : leur valeur ne peut pas être modifiée une fois qu'elles sont initialisées

Variables d'instances, de classe, constantes - 2

Exemple :

```
1 public class MaClasse {
2   public int valeur1 ; // variable d'instance
3   static double valeur 2 ;  // variable de classe
4   final long valeur3 ;  // constante
5 }
```

Anatomie du mot clé "static"



Les "packages"

Les "packages" sont utilisés pour :

- Prévenir les conflits de noms
- Regrouper un groupe de classes, d'interfaces et de sous-package
- Fournir un contrôle d'accès

Règles de visibilité

Modifier	Class	Package	Subclasses	World
public	✅	✅	✅	✅
protected	✅	✅	✅	❌
no modifier	✅	✅	❌	❌
private	✅	❌	❌	❌

Plan du cours

Programmation Orientée Objet (POO)

Héritage

Surcharge et redéfinition de méthodes

Polymorphisme

Héritage

- le concept d'**héritage** est au cœur de la programmation objet car il permet la **réutilisation** de classes
- Une classe dérive toujours d'un autre classe, `Object` quand rien n'est spécifié
- création d'une nouvelle classe **dérivée** à partir d'une **classe de base** déjà existante
- on dit que la classe **dérivée hérite**
  - des champs
  - des méthodes

  de la classe de base
- ces champs et méthodes peuvent être complétés

Exemple simple d'héritage

- considérons la classe de base suivante

```java
public class Point
{
 private double x;
 private double y;

 public Point(double abs,double ord) {
  this.x= abs;
  this.y= ord;
 }

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

Exemple simple d'héritage - 2

- on peut définir une classe dérivée de `Point` à l'aide du mot-clé `extends`

```
1 public class Point3D extends Point{
2  private double z;
3
4  public Point3D(double abs, double ord, double prof) {
5   super(abs, ord);
6   z = prof;
7  }
8 }
```

Utilisation d'un objet dérivé

```
1 Point3D p3d = new Point3D(2.0,3.0,4.0);
2 System.out.println(p3d);       // n'affiche pas la valeur de z
3 System.out.println(p3d.x);    // ERREUR x est private
```

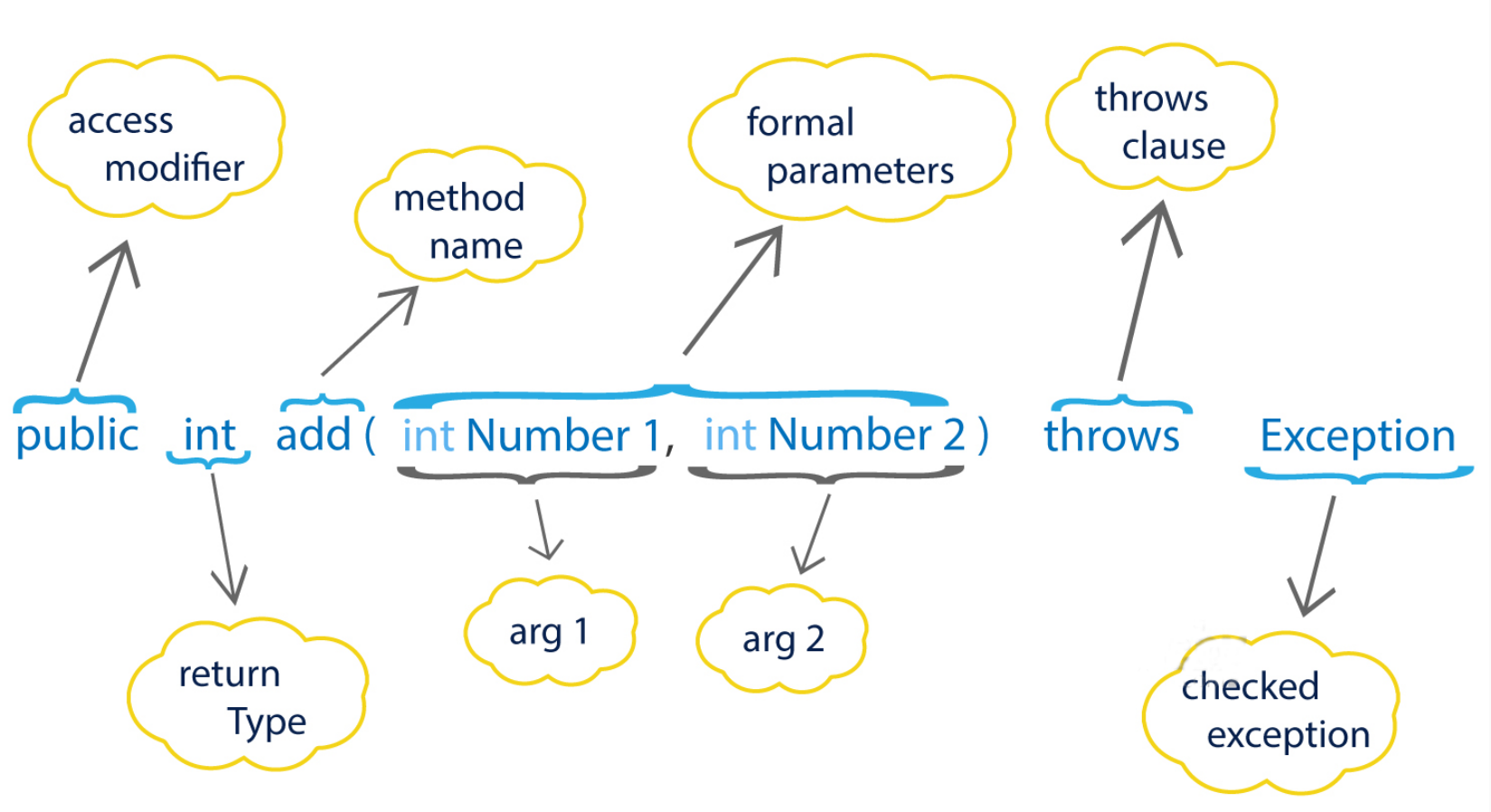
- accès aux méthodes / champs **publiques** de `Point` et `Point3D`
- une méthode de classe dérivée **n'a pas accès** aux champs / méthodes **privés** de sa classe de base
  - sinon dériver une classe suffirait à s'affranchir de l'encapsulation des champs / méthodes

Constructeur et classe fille

- Pour invoquer une méthode d'une classe mère, il suffit d'indiquer la méthode préfixée par `super`
- Le constructeur de la classe mère s'appelle en utilisant `super(paramètres)` avec les paramètres adéquats
- Dans l'exemple précédent nous avions `super(abs, ord` pour construire le `Point` qui est à la base du `Point3D`
- L'appel au constructeur de la classe mère est **obligatoire** dans une classe fille
- Cela doit être la **première instruction** du constructeur dérivé

Redéfinition d'une méthode héritée

- La **redéfinition** d'une méthode héritée doit impérativement conserver la signature de la méthode parente



- Si la signature de la méthode change, on parle de **surcharge**

Exemple de redéfinition d'une méthode héritée

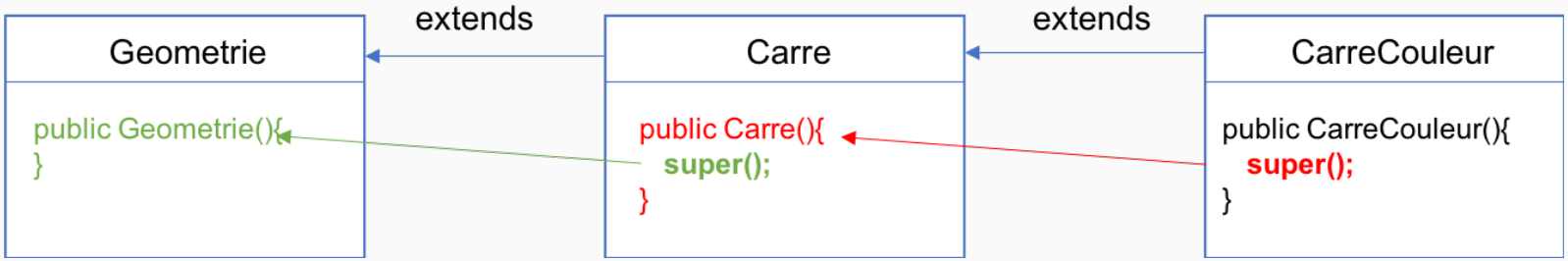
- on a vu que l'affichage du `Point3D` n'est pas satisfaisant
- on souhaite donc redéfinir cette méthode pour le `Point3D`

```
1  public String toString() {
2   return "Point (" + super.getAbscisse() + ", "
3       + super.getOrdonnee() + ", " + z + ")";
4  }
```

- Attention : cette nouvelle méthode n'est pas héritée : la classe mère ne possède pas de méthode possédant cette signature

Remarques concernant les constructeurs

- l'appel par **super** dans le constructeur d'une classe ne concerne que le constructeur de la classe de niveau **immédiatement supérieur**



- si une classe `A` ne possède pas de constructeur, toute utilisation de `super` dans une classe dérivée de `A` appelera le constructeur par défaut
- si une classe `B` possède un constructeur, toute classe dérivée de `B` doit déclarer un constructeur

Accès aux propriétés héritées

- Les variables et méthodes définies avec le modificateur d'accès `public` restent publiques à travers l'héritage et toutes les autres classes
- Une variable d'instance définie avec le modificateur `private` est bien héritée mais n'est pas accessible directement mais par les accesseurs hérités s'ils ont été définis
- Une variable définie avec le modificateur `protected` sera héritée dans toutes les classes filles qui pourront y accéder librement ainsi que les classes du même package

Plan du cours

Programmation Orientée Objet (POO)

Héritage

Surcharge et redéfinition de méthodes

Polymorphisme

Surcharge et redéfinition de méthodes

- **surcharge** d'une méthode à l'intérieur d'une même classe
  - même nom, entrée différente, sortie possiblement différente
- **redéfinition** : surcharge d'une méthode fournie par une classe ascendante (ie. plus haute dans la hiérarchie de l'héritage)
  - même nom, entrée et sortie identiques

```
1 public class Point {
2     public String toString() {}
3 }
4
5 public class Point3D extends Point {
6     public String toString() {...} // redefinition
7     public String toString(String msg) {...} // surcharge
8 }
```

Exemples de surcharges et redéfinition

```
1  class A {
2   public int method(int n){...}
3  }
4  // surdefinition-surcharge
5  class B extends A {
6   public double method(String s){...} // entree differente
7                                       // => sortie possiblement differente
8  }
9  // redefinition : entree et sortie identiques
10 class C extends A {
11  public int method(int n){...}
12 }
13 // ERREUR : entree identique mais sortie differente
14 class D extends A {
15  public float method(int n){...}
16 }
```

Redéfinitions et droits d'accès

- une redéfinition **peut** changer les droits d'accès d'une méthode uniquement pour en ajouter
  - `public` → `private` : erreur car une classe ascendante aurait plus de droits qu'une classe qui en hérite
  - `private` → `public` : possible, on parle d'extension des droits d'accès

```
1  class A {
2   public int f(int n){...}
3   private double g(int n){...}
4  }
5  class B extends A{
6   private int f(int n){...} // ERREUR
7  }
8  class C extends A{
9   public double g(int n){...} // OK extension des droits d'acce
10 }
```

Plan du cours

Polymorphisme

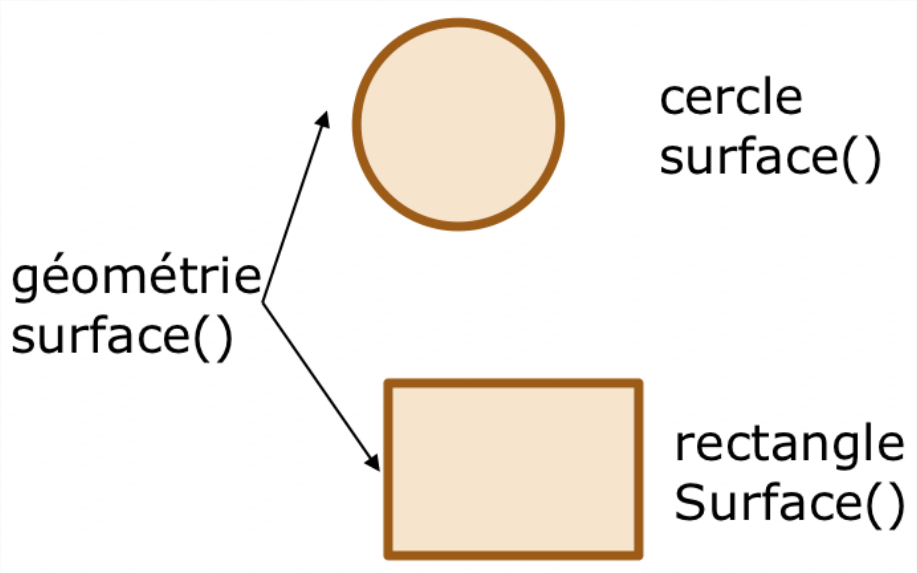
- le **polymorphisme** est un mécanisme fondamental de Java qui complète l'héritage
- il repose sur
  - le concept d'héritage, permettant d'hériter des attributs et des méthodes d'une autre classe
  - le **choix d'une méthode** à l'exécution et non pas à la compilation appelée *ligature dynamique*
- en d'autres termes, il permet de déclarer des objets d'une classe de base pour finalement instancier des objets de classes héritées

```
1 Point p = new Point3D(); // p de type Point contient
2                          // une reference a un objet Point3D
```

Avantages du polymorphisme

- Permet de réaliser des traitements analogues sur des objets possédant la même classe de base

Avantages du polymorphisme - 2

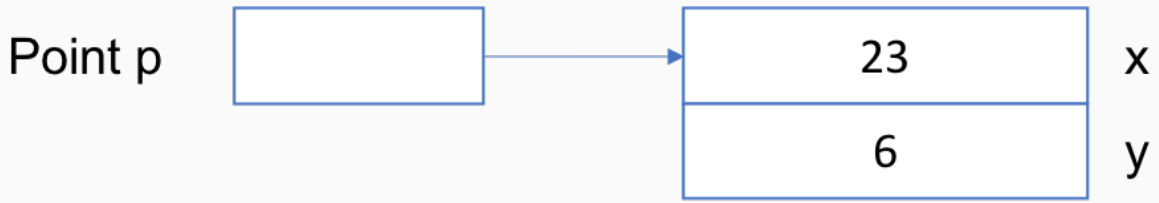



```
1 class Geometrie{ public double surface(){...} }
2 class Cercle extends Geometrie { public double surface(){...}
3 class Rectangle extends Geometrie { public double surface(){..
```

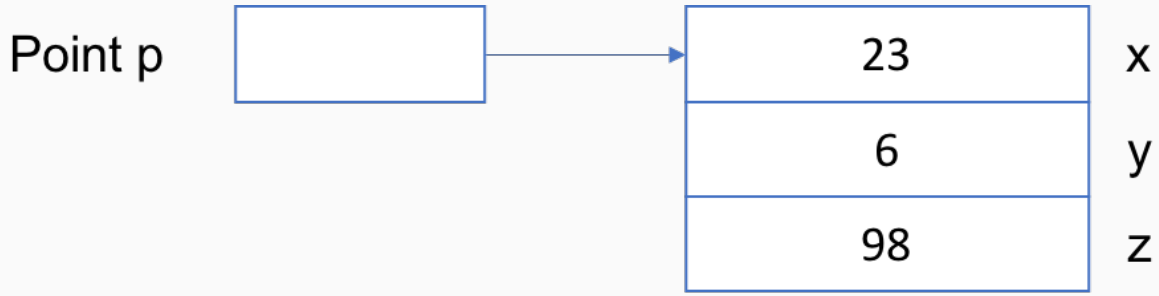
Le polymorphisme en pratique

- du fait de l'héritage, tout `Point3D` est un `Point`
- l'inverse n'est pas vrai !

```
1 Point p = new Point(23, 6); // Point(double x,double y)
```




```
1 p = new Point3D(23, 6, 98); // Point3D(double x,
2                             // double y,double z)
```



Exemple avec un tableau

- le polymorphisme permet de travailler avec des tableaux d'objets **hétérogènes** !
- il suffit pour cela que les types de classes des éléments du tableau héritent tous d'une même classe parente

```
1  Point[] courbe = new Point[3];
2  courbe[0] = new Point(3, 4);
3  courbe[1] = new Point3D(1, 2, 1);
4  courbe[2] = new Point3D(4, 5, 6);
5  for (Point p : courbe) System.out.println(p);
6                                          // utilise la methode
7                                          // affiche de Point
8                                          // ou de Point3D
```

Limites de l'héritage et du polymorphisme

- par exemple, considérons la méthode `equals()` définie pour les `Point` et les `Point3D`

```
1  class Point {
2   public boolean equals(Point p2){
3    return (x == p2.x) && (y == p2.y);
4   }
5  }
6  class Point3D {
7   public boolean equals(Point3D p2){
8    return (x == p2.x) && (y == p2.y) && (z == p2.z);
9   }
10 }
```

Limites de l'héritage et du polymorphisme - 2

```
1  Point p1, p2;
2  p1.equals(p2); // true ssi p1.x == p2.x et p1.y == p2.y
3
4  Point3D p3d1, p3d2;
5  p3d1.equals(p3d2); // true ssi idem Point et p3d1.z == p3d2.z
6
7  Point p3 = new Point3D(1,2,3);
8  Point p4 = new Point3D(1,2,5);
9  p3.equals(p4);
10     // true car utilisation de la methode egale(Point)
```

Limites de l'héritage et du polymorphisme - 3

- au moment de la compilation, `p3.equals(p4)` est associé à la méthode `equals(Point)`
- la classe `Point3D` ne possède pas de méthode `equals(Point)`
- on appelle donc la méthode de la classe parente la plus proche ⇒ `equals(Point)` de la classe `Point`

Limites de l'héritage et du polymorphisme - 4

- une solution au problème précédent consiste à doter la classe `Point3D` d'une méthode de signature `equals(Point)` et d'utiliser le transtypage `(Point3D)`

```
1 class Point3D extends Point {
2  public boolean equals(Point p){
3   Point3D p3d = (Point3D)p;
4   return (x == p3d.x) && (y == p3d.y) && (z == p3d.z);
5  }
6 }
```

La super classe Object

- `Object` est une classe particulière dont dérive implicitement toutes les classes simples
- nous pourrions donc écrire `class Point extends Object`
- on peut utiliser un `Object` pour représenter tout autre type de classe
- en supposant que l'on ait une méthode `deplace(double dx, double dy)` dans la classe `Point`

```
1 Point p = new Point();
2 Object o = p;     // OK
3 o.deplace(3,4);  // ERREUR pas de methode deplace dans Object
4 Point p1 = (Point) o;
5 p1.deplace(3,4); // OK grace a la conversion de type
6 ((Point)o).deplace(3,4); // OK, idem
7       // mais sans nouvelle reference
```

Les méthodes de Object

- la classe `Object` possède notamment 2 méthodes dont héritent toutes les classes simples et que l'on peut redéfinir
- la méthode d'affichage `toString()`
  - retourne une chaîne de caractères contenant le nom de la classe concernée et l'adresse de l'objet en hexadécimal précédé de `@`

```
1 Point3D p3d = new Point3D(1, 2, 4);
2 System.out.println(p3d.toString());
3 // affiche p. ex. Point3D@33909752
4 // sauf si on a redefinit toString() !
```

- et la méthode de comparaison `equals()`
  - qui compare les **adresses** des deux objets concernés

```
1 Point3D p1 = new Point3D(1,2,3);
2 Point3D p2 = new Point3D(1,2,3);
3 p1.equals(p2); // false car les adresses sont differentes
```