# Notions de base de la POO en Java et les génériques de Java

# Introduction

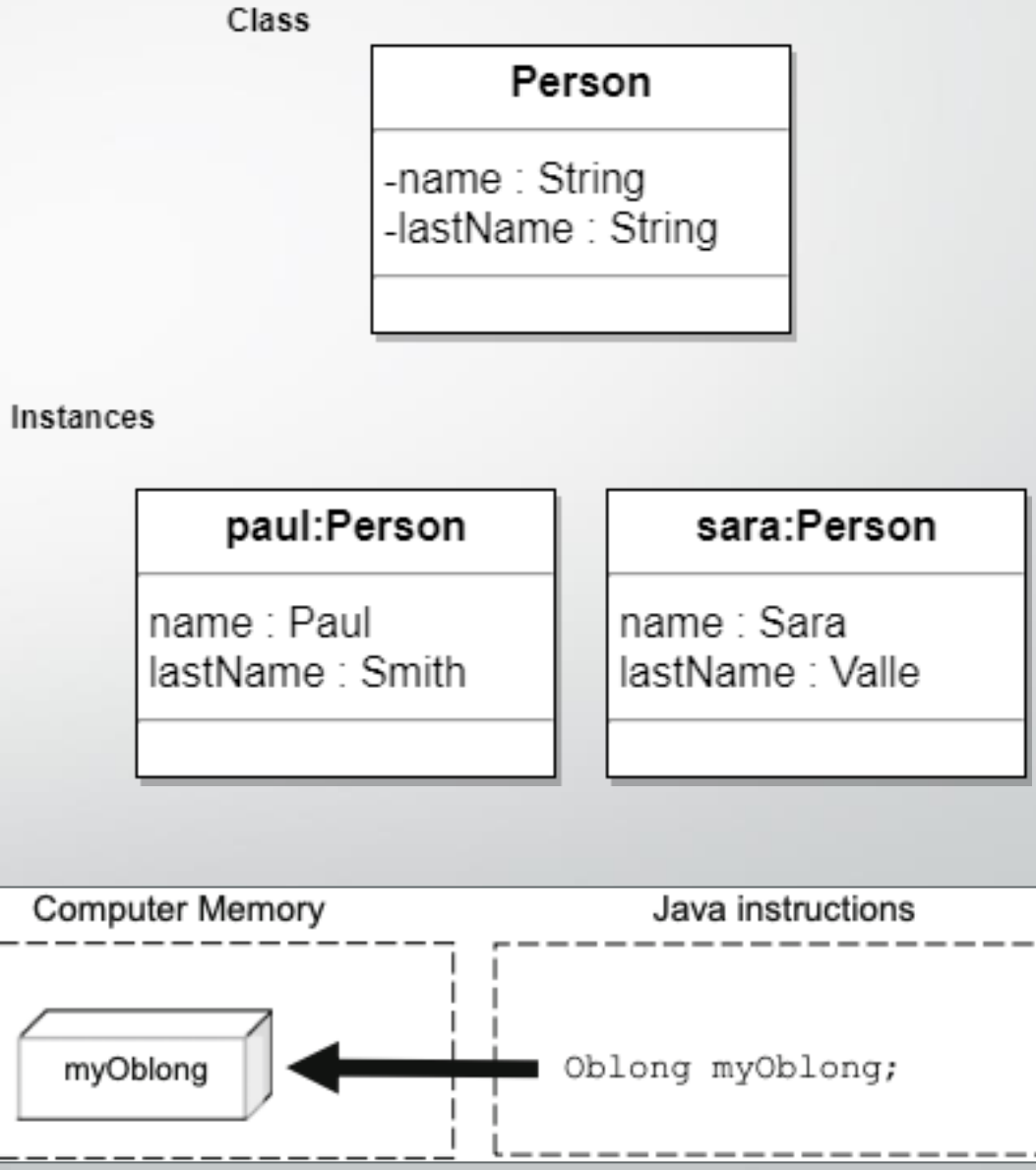
**Programmation OO: assembler des petits éléments (objets) autonomes pour en construire de plus grands et des programmes.**

Ecrire un programme en OO: définir des objets (classes) et les utiliser ou réutiliser des objets existants.

L'utilisation d'un objet se fait de manière organisée via les méthodes (et/ou les attributs) prévues par l'objet.

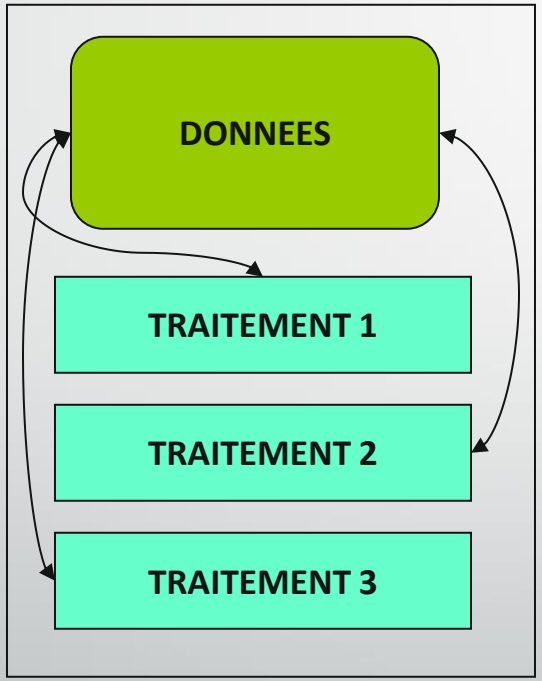
**On ne sépare plus les données des traitements portant sur ces données. Objet = Données + Programmes**.

*Encapsulation*: on "cache" des données à l'intérieur d'une classe en ne permettant de les manipuler qu'au travers des méthodes de la classe.



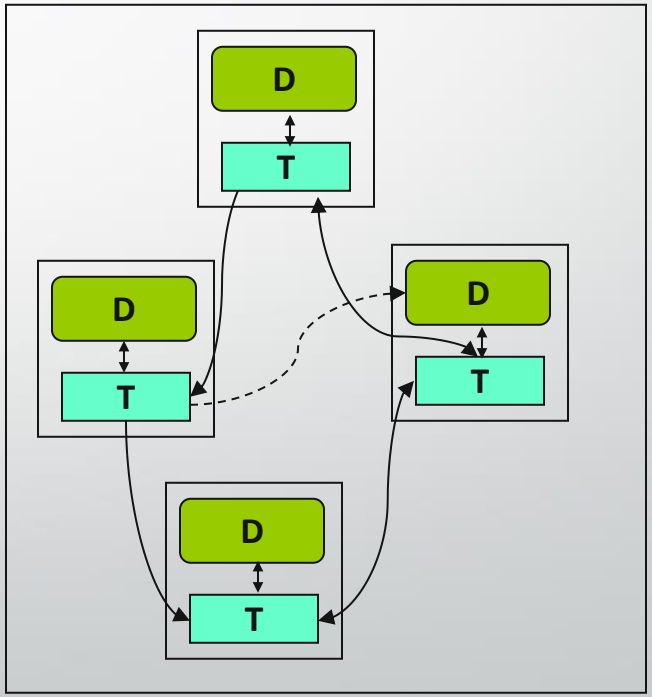
# Programmation impérative vs programmation OO

**Programme impératif**: **ensemble de fonctions** effectuant des opérations sur des données communes: données et traitements sont séparés




Approche impérative : « que doit faire mon programme ? »

**Programme OO**: ensemble d'objets contenant des données ET des traitements sur ces données. Ces objets s'utilisent entre eux.




Approche OO : « **de quoi doit être composé mon programme ?** »

# Objet

Approche OO : « **de quoi doit être composé mon programme ?** »

*Cette composition est conséquence d'un choix de modélisation fait pendant la conception*

Exemple: objets d'une bibliothèque


Germinal
E. Zola


Le seigneur des anneaux
J.R.R.Tolkien

Le Monde


M. Martin
Bibliothécaire


A. Dupont
Directrice


A. Deschamps
Lecteur


A. Durand
Lectrice

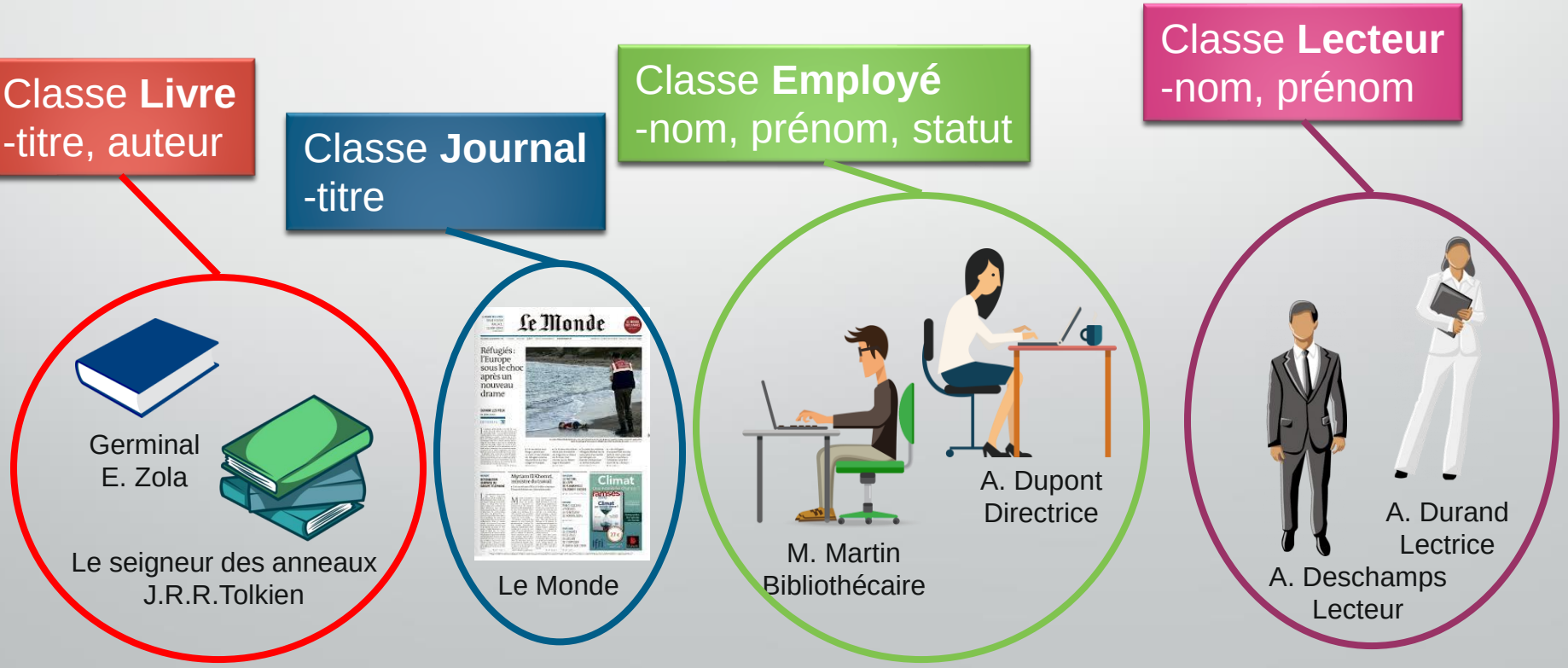
# Classe

Une ***classe*** est un **modèle commun à un ensemble d'objets** de mêmes caractéristiques.

Une ***instance*** (de classe) est l'autre nom d'un objet réel. C'est une manifestation concrète d'une classe.

Une classe est une représentation abstraite d'un objet et une instance en est une représentation concrète.

Ex: on a une classe Employé (avec un nom, prénom, statut) et 2 objets (instances): A. Dupont, directrice et M. Martin, bibliothécaire.



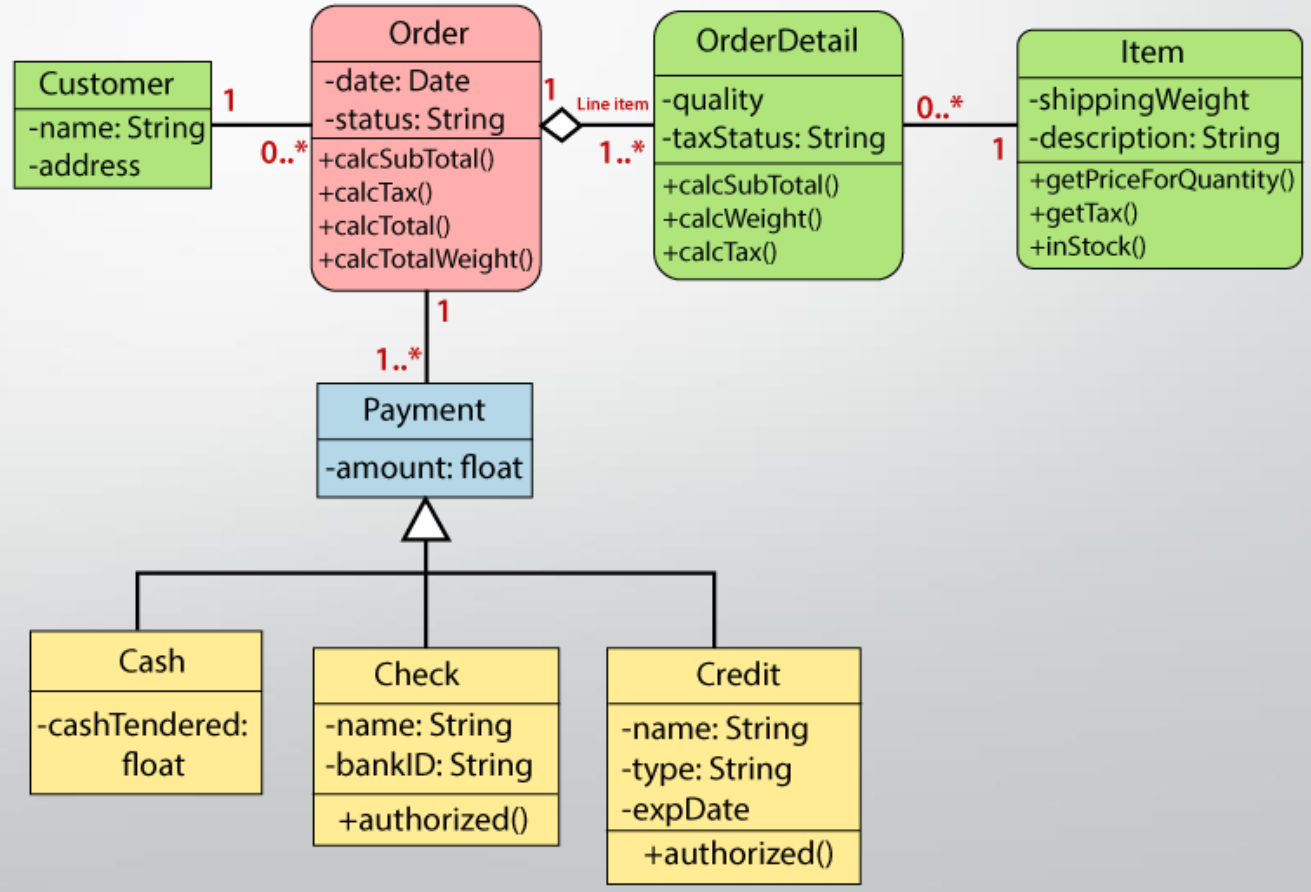
# Contenu d'une classe

Une classe définit donc les caractéristiques communes à un ensemble d'objets similaires. Elle se compose de 2 parties :

**Les attributs (données)** : caractéristiques individuelles qui différencient un objet d'un autre. Ex: un Lecteur a pour attribut: un **nom**, un **prénom**

**Les méthodes (traitements)** : ce sont les traitements auxquels un objet peut répondre.

Ex : un « Lecteur » peut **emprunter** ou **retourner** un ouvrage. Un « Current_Account » sait ouvrir, fermer, …et faire un trf.



# Création d'une instance

Pour obtenir un nouvel objet d'une classe on utilise l'opérateur **`new`** <Classe>**`()`**

Ex: **`Lecteur l = new Lecteur();`**

A partir d'un objet on peut accéder aux attributs et/ou méthodes grâce à une notation pointée (**operateur de sélection**): <var objet>**`.`**<membre>:

Ex: **`l.nom = "Deschamps";`** Remarque: on évite !

**`l.emprunte(o);`**

A l'intérieur d'une même classe on peut accéder directement aux attributs et méthodes (pas besoin de notation pointée) mais il est parfois nécessaire de faire référence à l'objet courant. On utilise alors le mot clé **`this`** qui désigne l'objet qui doit résoudre la sélection (souvent l'objet courant).

# Constructeur

**Quel est l'effet de `new` ?** Allocation dans le tas (heap) de l'espace nécessaire (dépendant du constructeur utilisé pour initialiser l'objet – ses attributs par exemple) pour le nouvel objet.

Par défaut toute classe possède un constructeur (sans arguments) qui initialise toutes les variables d'instances (attributs) à des valeurs par défaut suivant leurs types:

nombres: **0** booléen: **`false`** objets: **`null`**

On peut définir ses propres constructeurs, ils se définissent comme des méthodes de même nom que la classe mais **sans type de retour**.

Exemple: définition d'un constructeur dans la classe Lecteur:

```
public class Lecteur {
  public String nom;
  public String prenom;
  public Lecteur(String nom, String prenom) {
    this.nom = nom;
    this.prenom = prenom;
  }
}
```

utilisation de **`this`** pour atteindre chaque attribut

Utilisation:

```
Lecteur l = new Lecteur("Deschamps", "Antoine");
```

# Example: The Contacts Manager (i)

Assume you're writing a Java program that is responsible for storing and managing all your friends' contact information.

We'll **start by creating a class** that is responsible **for storing** all contact information of a **single person**, it will look something like this:

**Contact.java**

```
public class Contact {
    String name;
    String email;
    String phoneNumber;
}
```

All fields, no methods, since a contact object itself won't be "doing" much actions itself in the scope of this program, it's merely a slightly more advanced data type that can store a few strings in 1 variable.

**Note:** Notice how we used a String to store the phone number instead of using int!

# Example: The Contacts Manager (ii)

Now, let's create the class that will be in charge of adding and searching for contacts: **ContactsManager**.

**ContactsManager.java**

```
public class ContactsManager {

}
```

# Example: The Contacts Manager (ii)

Now, let's create the class that will be in charge of adding and searching for contacts: **ContactsManager**.

This class will be **storing the contacts in an array (an array of Contacts)**, another field represents the number of friends added to the array.

**ContactsManager.java**

```
public class ContactsManager {
   Contact [] myFriends;
   int friendsCount;
}
```

# Example: The Contacts Manager (ii)

Now, let's create the class that will be in charge of adding and searching for contacts: **ContactsManager**.

This class will be **storing the contacts in an array (an array of Contacts)**, another field represents the number of friends added to the array.

This class has a default constructor (ContactsManager) that will initialize myFriends and friendsCount fields.

**ContactsManager.java**

```
public class ContactsManager {
   Contact [] myFriends;
   int friendsCount;

   ContactsManager (){
      this.friendsCount = 0;
      this.myFriends = new Contact[500];
   }
}
```

# Example: The Contacts Manager (ii)

Now, let's create the class that will be in charge of adding and searching for contacts: **ContactsManager**.

This class will be **storing the contacts in an array (an array of Contacts)**, another field represents the number of friends added to the array.

This class has a default constructor (ContactsManager) that will initialize myFriends and friendsCount fields.

The first method we will create in the ContactsManager class is the addContact method which will add a Contact object to the Contact array myFriends.

**ContactsManager.java**

```
public class ContactsManager {
   Contact [] myFriends;
   int friendsCount;

   ContactsManager (){
      this.friendsCount = 0;
      this.myFriends = new Contact[500];
   }

  void addContact(Contact contact){
      this.myFriends[friendsCount] = contact;
      this.friendsCount ++;
   }
}
```

# Example: The Contacts Manager (ii)

Now, let's create the class that will be in charge of adding and searching for contacts: **ContactsManager**.

This class will be **storing the contacts in an array (an array of Contacts)**, another field represents the number of friends added to the array.

This class has a default constructor (ContactsManager) that will initialize myFriends and friendsCount fields.

The first method we will create in the ContactsManager class is the addContact method which will add a Contact object to the Contact array myFriends.

Let's add another method called searchContact to search through the array using a name String and return a Contact object once a match is found.

**ContactsManager.java**

```
public class ContactsManager {
   Contact [] myFriends;
   int friendsCount;

   ContactsManager (){
      this.friendsCount = 0;
      this.myFriends = new Contact[500];
   }

  void addContact(Contact contact){
      this.myFriends[friendsCount] = contact;
      this.friendsCount ++;
   }

   Contact searchContact (String searchName){
      for (int i = 0; i < this.friendsCount; i++){
         if (myFriends[i].name.equals(searchName)){
            return myFriends[i];
         }
      }
      return null;
   }
}
```

# Example: The Contacts Manager (iii)

To be able to run this program, we need the main method, so let's create another class called Main that will hold this method.

**Main.java**

```
public class Main {
    public static void main(String[] args) {

    }
}
```

# Example: The Contacts Manager (iii)

To be able to run this program, we need the main method, so let's create another class called Main that will hold this method.

This means that once this program runs, the main method will start which will create the ContactManager object myContactManager and thus ready to be used.

**Main.java**

```
public class Main {
   public static void main(String[] args) {
      // Create the ContactsManager object
      ContactsManager myCManager = new ContactsManager();

   }
}
```

# Example: The Contacts Manager (iii)

To be able to run this program, we need the main method, so let's create another class called Main that will hold this method.

This means that once this program runs, the main method will start which will create the ContactManager object myContactManager and thus ready to be used.

Then, let's **create three new Contact** variables and set the name to a friend's name, their email and their phoneNumber. Next, **call the addContact method** in myContactsManager to add each contact.

**Main.java**

```
public class Main {
    public static void main(String[] args) {
        // Create the ContactsManager object
        ContactsManager myCManager = new ContactsManager();
        // Create a new Contact object for James
        Contact friendJames = new Contact();
        friendJames.name = "James";
        friendJames.email = "james@ensta-bretagne.fr";
        friendJames.phoneNumber = "0012223333";
        myCManager.addContact(friendJames);
        // Create a new Contact object for Paul
        Contact friendPaul = new Contact();
        friemdPaul.name = "Paul";
        friemdPaul.email = "paul@ensta-bretagne.fr";
        friemdPaul.phoneNumber = "987654321";
        myCManager.addContact(friendPaul);
        // Create a new Contact object for Jessica
        Contact friendJessica = new Contact();
        friendJessica.name = "Jessica";
        friendJessica.phoneNumber = "5554440001";
        myCManager.addContact(friendJessica);

    }
}
```

# Example: The Contacts Manager (iii)

To be able to run this program, we need the main method, so let's create another class called Main that will hold this method.

This means that once this program runs, the main method will start which will create the ContactManager object myContactManager and thus ready to be used.

Then, let's **create three new Contact** variables and set the name to a friend's name, their email and their phoneNumber. Next, **call the addContact method** in myContactsManager to add each contact.

To finish, **search for a contact using the method searchContact and print out their phone number**.

**Main.java**

```
public class Main {
    public static void main(String[] args) {
        // Create the ContactsManager object
        ContactsManager myCManager = new ContactsManager();
        // Create a new Contact object for James
        Contact friendJames = new Contact();
        friendJames.name = "James";
        friendJames.email = "james@ensta-bretagne.fr";
        friendJames.phoneNumber = "0012223333";
        myCManager.addContact(friendJames);
        // Create a new Contact object for Paul
        Contact friemdPaul = new Contact();
        friemdPaul.name = "Paul";
        friemdPaul.email = "paul@ensta-bretagne.fr";
        friemdPaul.phoneNumber = "987654321";
        myCManager.addContact(friemdPaul);
        // Create a new Contact object for Jessica
        Contact friendJessica = new Contact();
        friendJessica.name = "Jessica";
        friendJessica.phoneNumber = "5554440001";
        myCManager.addContact(friendJessica);
        //Let's search of a contact and display their phone number
        Contact result = myCManager.searchContact("Jessica");
        System.out.println(result.phoneNumber);
    }
}
```

# Example: The Contacts Manager

**Contact.java**

```
public class Contact {
   String name;
   String email;
   String phoneNumber;
}
```

**ContactsManager.java**

```
public class ContactsManager {
   Contact [] myFriends;
   int friendsCount;
   ContactsManager (){
      this.friendsCount = 0;
      this.myFriends = new Contact[500];
   }
   void addContact(Contact contact){
      this.myFriends[friendsCount] = contact;
      this.friendsCount ++;
   }
   Contact searchContact (String searchName){
      for (int i = 0; i < this.friendsCount; i++){
         if (myFriends[i].name.equals(searchName)){
            return myFriends[i];
         }
      }
      return null;
   }
}
```

**Main.java**

```
public class Main {
   public static void main(String[] args) {
      // Create the ContactsManager object
      ContactsManager myCManager = new ContactsManager();
      // Create a new Contact object for James
      Contact friendJames = new Contact();
      friendJames.name = "James";
      friendJames.email = "james@ensta-bretagne.fr";
      friendJames.phoneNumber = "0012223333";
      myCManager.addContact(friendJames);
      // Create a new Contact object for Paul
      Contact friemdPaul = new Contact();
      friemdPaul.name = "Paul";
      friemdPaul.email = "paul@ensta-bretagne.fr";
      friemdPaul.phoneNumber = "987654321";
      myCManager.addContact(friemdPaul);
      // Create a new Contact object for Jessica
      Contact friendJessica = new Contact();
      friendJessica.name = "Jessica";
      friendJessica.phoneNumber = "5554440001";
      myCManager.addContact(friendJessica);
      //Let's search of a contact and display their phone number
      Contact result = myCManager.searchContact("Jessica");
      System.out.println(result.phoneNumber);
   }
}
```

# Définition de plusieurs constructeurs

Tout comme pour les méthodes il est possible de définir plusieurs constructeurs avec des arguments de type différent. On appelle cela la *surcharge*.

```
public class Lecteur {
  public String nom;
  public String prenom;

  public Lecteur(String nom, String prenom) {
    this.nom = nom;
    this.prenom = prenom;
  }

  public Lecteur(String nom) {
    this(nom, null);
  }

}
```

On fait appel à l'autre Constructeur; cela doit être la 1ère instruction.

# Access modifiers

A well-designed Java code is one that wouldn't even allow you to create bugs by mistake. One thing that can help is **using the correct access modifiers** and so **avoid** that ourselves or others to **directly access what is private or protected**!

```
public class Book{
    private String title;
    private String author;
    public boolean isBorrowed;
    public Book(String title, String author){
        this.title = title;
        this.author = author;
    }
}
```

**This way you can guarantee that once a book object has been created, the title and author will never change!**

However, the isBorrowed field can be modified by other classes and so, you will be able to do something like this from anywhere in your project:

```
book.isBorrowed = true;
```

**Which is risky because you could end up mistakingly setting the boolean to true when you only meant to check if it was true or false!**

# Niveaux de Protection

Le but de la protection est de limiter la visibilité (et donc l'accessibilité) d'un attribut ou d'une méthode. Java propose **4 niveaux de protection** (de la moins restrictive à la plus restrictive) :

1. **`public`**: est visible partout.
2. **Par défaut**: est un élément **`public`** qui n'est visible que dans les classes du même *package*.
3. **`protected`**: n'est visible que depuis les sous-classes.
4. **`private`**: n'est pas visible hors de la classe.

Comme on l'a vu:

```
[<protection>] class <nom> {
    … définition d'attributs …
    … définition de méthodes …
}
```

Exemple:

```
public class Lecteur {

  public String nom;
  private String prenom;

  void emprunte(Ouvrage o) {
  }

  void retourne(Ouvrage o) {
  }
}
```

# **QUIZ QUESTION** about Protection

Package1

```
public class C1 {
   public int a;
   protected int b;
   int c;
   private int d;
}
```

```
class C3{

}
```

```
class C2 extends C1{

}
```

Package2

```
public class C4 extends C1
{

}
```

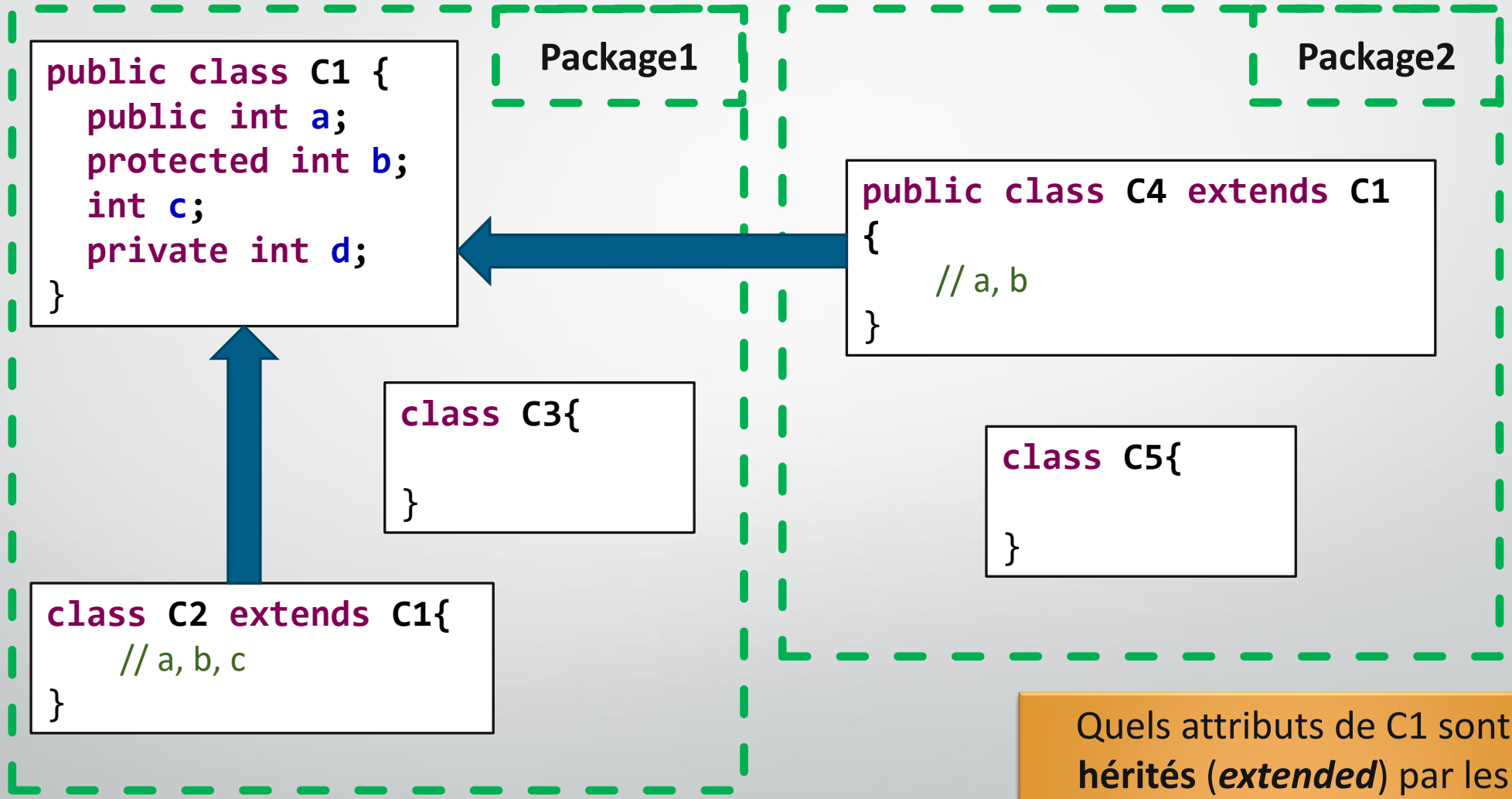
```
class C5{

}
```

Quels attributs de C1 sont **hérités** (***extended***) par les autres classes?

Par défaut (sans déclaration de protection): acces interne au package

# QUIZ ANSWER about Protection




Quels attributs de C1 sont **hérités** (***extended***) par les autres classes?

Par défaut (sans déclaration de protection): acces interne au package

# **QUIZ QUESTION** about Protection

Package1

```
public class C1 {
   public int a;
   protected int b;
   int c;
   private int d;
}
```

```
class C3{

}
```

```
class C2 extends C1{
    // a, b, c
}
```

Package2

```
public class C4 extends C1
{
    // a, b
}
```

```
class C5{

}
```

Quels attributs de C1 sont **visibles** (**accessibles**) par les autres classes?

Par défaut (sans déclaration de protection): acces interne au package

# QUIZ ANSWER about Protection

Package1

```
public class C1 {
   public int a;
   protected int b;
   int c;
   private int d;
}
```

```
class C3{
  // a, c
}
```

```
class C2 extends C1{
     // a, b, c
}
```

Package2

```
public class C4 extends C1
{
      // a, b
}
```

```
class C5{
      // a
}
```

Quels attributs de C1 sont **visibles** (**accessibles**) par les autres classes?

Par défaut (sans déclaration de protection): acces interne au package

# It's strongly recommended in Java to label ALL fields as **private**

For example, when defining a Book class, instead of writing:

```
class Book{
    String title;
    String author;
    boolean isBorrowed;
}
```

You should define every attribute private, and only initialize them in the constructor and access/modify them with public methods that return/change the value of such hidden field.

This way, you can guarantee that once a book object has been created, the title and author will never change! And every time you change the isBorrowed you are conscient of that!

```
public class Book{
    private String title;
    private String author;
    private boolean isBorrowed;

    public Book(String title, String author){
        this.title = title;
        this.author = author;
    }

    public void borrowBook(){
        isBorrowed = true;
    }
    public void returnBook(){
        isBorrowed = false;
    }
    public boolean isBookBorrowed(){
        return isBorrowed;
    }
}
```

**Encapsulation** is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction.

**Encapsulation** is the mechanism of wrapping the data and code acting on the data together as a single unit.

**To achieve encapsulation in Java :**

- Declare the variables of a class as **private.**
- Provide **public setter and getter methods** to modify and view the variables values.

**Benefits of Encapsulation**

- Attributes can be made read-only or write-only.
- A class can have total control (implement verifications) over what is stored in its attributes.

```
public class Employee {
  private String name;
  private double salary;

  public String getName() {
    return name;
  }

  public String getSalary() {
    return salary;
  }

  public void setSalary(double sal) {
    if (sal > 0.0){
        this.salary = sal;
    }
    …
  }
}
```

The **private methods** are usually known as **helper methods**, since they can only be seen and called by the same class, they are usually there to organize your code and keep it simple and more readable.

```
class Person{
   private String userName;
   private String SSN;
   private String getId(){
      return SSN + "-" + userName;
   }
   public String getUserName(){
      return userName;
   }
   public boolean isSamePerson(Person p){
      if(p.getId().equals(this.getId()){
         return true;
      }
      else{
         return false;
      }
   }
}
```

The class Person has both fields set to **private** because if they were public, then any other class will be able to change such sensitive information. Setting them to private means that only methods and constructors inside this class can do so!

**The method getId() was also set to private so that no other class can know the social security number of any person!**

However, we were still able to **use that method internally** when comparing this person with another person object in the **isSamePerson(Person p)** method

# Five **recommendations to correctly encapsulate** your Java code

1. Always try to declare all fields as **private**
2. Create a constructor that accepts those private fields as inputs
3. Create a public method that *sets* each private field, this way you will ***know*** when you are changing a field and ***control*** who and how is trying to change it. These methods are called **setters**
4. Create a public method that *returns* each private field, so you can ***read the value without mistakingly changing it*** and ***control*** who and how is trying to access it. These methods are called **getters**
5. Set any of your methods to **public** that are considered **actions** and set any of your methods to **private** that are considered **helpers**.

# Objets et références

Lors de la création d'un objet avec **`new`** on récupère une référence (c-à-d un pointeur) sur une zone mémoire contenant les membres (variables d'instances et méthodes) de l'objet. Ainsi tous les objets sont manipulés implicitement via des pointeurs.

**Attention**: les opérateurs d'affectation **`=`** et de comparaisons **`==`** et **`!=`** ne travaillent que sur les références, pas sur les contenus.

```
Lecteur x = new Lecteur("Deschamps", "Antoine");
Lecteur y = new Lecteur("Deschamps", "Antoine");

if (x == y)                vaut faux !
    …
```

Il faut définir une méthode (ou mieux redéfinir **`equals`**) qui permet vraiment de comparer 2 objets.

# Variables de classe

Aux attributs d'une classe on les qualifie de **variables d'instance** car elles n'existent qu'une fois que l'on a instancié la classe.

On a parfois besoin de **variables qui soient communes à toutes les instances d'une classe**. On parle alors de **variable de classe**. Il faut voir une variable de classe comme une variable globale visible par tous les objets issus de cette classe.

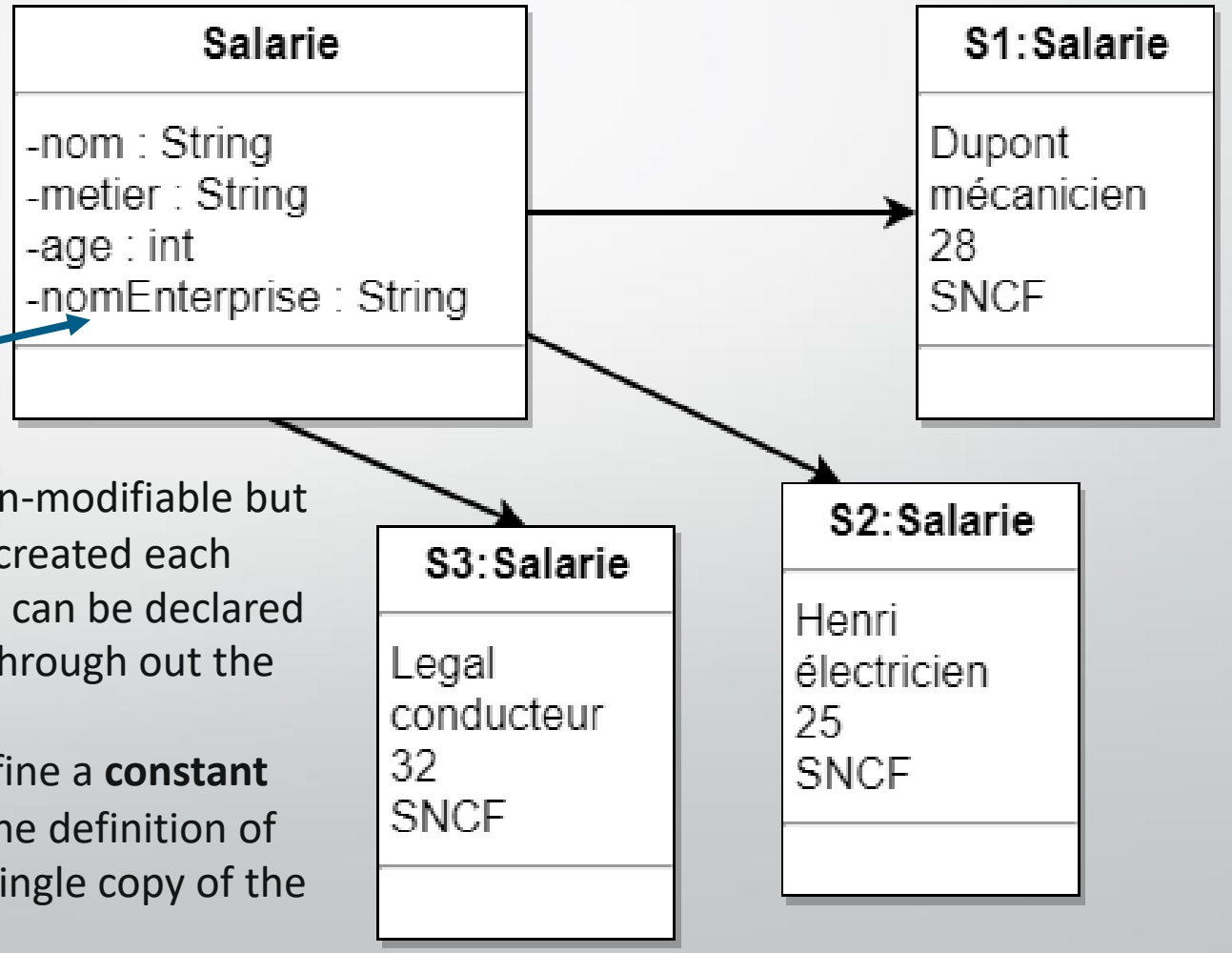
**On déclare une telle variable avec le mot clé `static`.**

Exemple: on veut définir une classe **`Ouvrage`** de telle sorte que tout ouvrage crée se voit attribué un numéro séquentiel (1, 2, …).

```
public class Ouvrage {
  private static int noOuvrage = 0;
  protected int numero;
  protected String titre;

  public Ouvrage(String titre) {
    numero = ++noOuvrage; //un numéro séquentiel
    this.titre = titre;
  }
}
```

# **QUIZ QUESTION:** Identifier les **variables que pourront être de classe** et les **variables que devront être d'instance** dans la classe Salarie




Is a ***final*** variable (its value is un-modifiable but a separate copy of the variable is created each time you create a new object) but can be declared as ***static*** (have only one copy through out the program) to optimize memory...
Or both (***final static***) to define a **constant** attribute (because, according to the definition of the constant you need to have a single copy of the variable throughout the program)

# Variables de classe: initialisation

**Une variable de classe est allouée et initialisée au début du programme (lorsque la classe est chargée en mémoire)**. On peut initialiser une variable de classe lors de sa déclaration:

```
private static int noOuvrage = 0;
```

Lorsque on a une initialisation complexe on peut déclarer un **bloc d'instruction pour réaliser cette initialisation**. On déclare un tel bloc avec **`static {…}`**. Il est exécuté au chargement de la classe.

```
private static int tbl[] = new int [10];
static {
  for(int i = 0; i < tbl.length; i++)
  tbl[i] = 2 * i;
}
```

# QUIZ QUESTION

```
public class Person {
   public static int instanceCount; //=0;
   public int localCount; //= 0;

  public Person(){
      instanceCount++;
      localCount++;
   }
}
```

```
public class Main {
   public static void main(String[] args) {
      Person person1 = new Person();
      Person person2 = new Person();
      Person person3 = new Person();
      Person person4 = new Person();
      // Print the values of both counters
      System.out.println("(" + person4.localCount + "," +
Person.instanceCount + ")");
   }
}
```

What will the above code print for (**`localCount`**, **`instanceCount`**)?

A. (4,4)

B. (1,4)

C. (4,1)

D. (1,1)

And what is the value of **`person4.`***`instanceCount`* ?

# Méthodes de classe

Similairement on peut définir des **méthodes de classe** (par opposition aux méthodes d'instances). Ces méthodes sont donc accessibles sans avoir besoin d'instancier la classe.

Elles aussi se définissent avec le mot clé **`static`** (ex: **`main`**).

Exemple: on peut connaître le nombre d'ouvrages en consultant la variable **`noOuvrage`**. On donne donc une méthode de classe:

```
public class Ouvrage {

  private static int noOuvrage = 0;

  public static int getNbOuvrage() {
    return noOuvrage;
  }
  …
}
```

# Méthodes de classe

Just like static fields, static methods also belong to the class rather than the object.

Si elles sont déclarées **`public`**, elles peuvent aussi être utilisées depuis n'importe où avec la notation pointée: <Classe>.<membre>.

```
int nbOuvrage = Ouvrage.getNbOuvrage();
```

Les classes **`Math`** et **`String`**, par exemple, fournissent plusieurs méthodes de classe.

```
double x = Math.sqrt(y);   int diff = Math.abs(a-b);
```

Static methods are ideally used to create methods that do not need to access any fields in the object, in other words, a **method that is a standalone function**.

A static method takes input arguments and returns a result based only on those input values and nothing else. **However, a static method can still access static fields**, that's because static fields also belong to the class and are shared amongst all objects of that class.

# Here's an example of a calculator implementation with some static methods:

```
public class Calculator {

    public static int add(int a, int b) {
        return a + b;
    }

    public static int subtract(int a, int b) {
        return a - b;
    }

}
```

Since both **add** and **subtract** don't need any object-specific values, they can be declared static as seen above and hence you can call them directly using the class name Calculator without the need to create an object variable at all:

```
Calculator.add(5,10);
```

# QUIZ QUESTION

```
public class Person {
  int age;
  static int averageAge = 21;

  public static void salary(){
    int aux = sum(2, 3);
    int salary = age + averageAge + aux;
  }

  public static int averageAge(){
    return averageAge;
  }

  public int sum(int x, int y){
    return  averageAge()+x+y;
  }
}
```

**Find the errors in this code**

# QUIZ QUESTION

```
public class Person {
  int age;
  static int averageAge = 21;

  public static void salary(){
    int aux = sum(2, 3);
    int salary = age + averageAge + aux;
  }

  public static int averageAge(){
    return averageAge;
  }

  public int sum(int x, int y){
    return  averageAge()+x+y;
  }
}
```

**Find the errors in this code**

Is a good practive to initialize attributs to avoir the by default “null” for the objects

**Instance methods cannot be called from static methods**

**Instance atributes cannot be called from static methods**