

The image features a light gray background with decorative geometric elements in the corners. The top-left corner contains a cluster of squares in shades of red, orange, and black. The bottom-right corner features a cluster of squares in shades of green, teal, and black. The text 'Initiation Java' is centered in a bold, black, sans-serif font.

# Initiation Java

# Plan

**1**

**Pourquoi  
java**

**2**

**Les  
types**

**3**

**Conditions  
& Boucles**

**4**

**Les méthodes**

**5**

**Gestion  
des erreurs**

**6**

**Les tableaux**





**1**

**Pourquoi Java ?**



# Pourquoi Java ?

1. Java est un langage de programmation populaire et largement utilisé: Du Web au mobile en passant par le bureau, Java se retrouve dans pratiquement tous les systèmes d'exploitation.
2. Java est un solide précurseur d'autres langages de programmation: Les avantages de l'apprentissage de Java sont fondamentaux pour apprendre les langages de codage ultérieurs tels que C, C #, C ++, Python et autres.
3. Java est polyvalent: Non seulement Java est universellement utilisé dans la technologie, mais il est facilement évolutif et, surtout, très portable, car il doit être exécuté via une machine virtuelle Java (JVM) compatible multiplateforme

# Pourquoi Java ?

1. Les développeurs Java bénéficient du support de la communauté: Besoin d'aide ? La communauté Java a ce qu'il vous faut. Java dispose de forums actifs, de bibliothèques open source et de groupes d'utilisateurs Java pour tous les niveaux de compétence.
2. Les développeurs Java gagnent des salaires élevés
3. Java est là pour rester: Le paysage technologique est en constante évolution, les développeurs cherchant à apprendre les nouveaux langages et frameworks les plus en vogue pour leur donner un avantage



**2**

# Les types



# Les types

Les types de données sont divisés en deux groupes :

- Types de données primitifs - `byte short int long float double boolean char`
- Types de données non primitifs - tels que String, Arrays et Classes

## Primitive Data Type

### Integer

byte (1 byte)

long (8 bytes)

short (2 bytes)

int (4 bytes)

int num = 56

### Float

float (4 bytes)

double (8 bytes)

float num = 56851.3285

### Character

char (2 byte)

char ch = 'X'

### Boolean

bool (1 byte,  
but makes use  
of 1 bit of it)

boolean b = true



# Les types

La principale différence entre les types de données primitifs et non primitifs est la suivante :

- Les types primitifs sont prédéfinis (déjà définis) en Java. Les types non primitifs sont créés par le programmeur et ne sont pas définis par Java sauf pour String
- Les types non primitifs peuvent être utilisés pour appeler des méthodes pour effectuer certaines opérations, contrairement aux types primitifs.
- Un type primitif a toujours une valeur, tandis que les types non primitifs peuvent être `.null`

# Les types

- Un type primitif commence par une lettre minuscule, tandis que les types non primitifs commencent par une lettre majuscule.
- La taille d'un type primitif dépend du type de données, tandis que les types non primitifs ont tous la même taille.

# Classes wrapper

Une classe Wrapper est une classe dont l'objet encapsule des types de données primitifs. Lorsque nous créons **un objet** dans une classe wrapper, il contient un champ et dans ce champ, nous pouvons **stocker des types de données primitifs**.

En d'autres termes, **nous pouvons encapsuler une valeur primitive dans un objet de classe wrapper**.

Primitive Data Type	Wrapper Class
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean



# Classes wrapper

La conversion automatique des types primitifs en l'objet de leurs classes wrapper correspondantes est connue sous le nom d'autoboxing. Par exemple – conversion de int en Entier, long en Long, double en Double etc.

```
public class AutoBoxingTest {  
    public static void main(String args[]) {  
        int num = 10;  
        Integer obj = Integer.valueOf(num);  
        System.out.println(num + " " + obj);  
    }  
}
```



**3**

# Conditions & boucles

# Les conditions

## If..else

```
if (condition) {  
    // block of code to be executed if the condition is true  
} else {  
    // block of code to be executed if the condition is false  
}
```

```
int time = 20;  
if (time < 18) {  
    System.out.println("Good day.");  
} else {  
    System.out.println("Good evening.");  
}  
// Outputs "Good evening."
```



# Les conditions

## Opérateur ternaire

```
variable = (condition) ? expressionTrue : expressionFalse;
```

```
int time = 20;  
String result = (time < 18) ? "Good day." : "Good evening.";  
System.out.println(result);
```

# Les conditions

## Switch

```
switch(expression) {  
  case x:  
    // code block  
    break;  
  case y:  
    // code block  
    break;  
  default:  
    // code block  
}
```

```
int day = 4;
switch (day) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    case 3:
        System.out.println("Wednesday");
        break;
    case 4:
        System.out.println("Thursday");
        break;
    case 5:
        System.out.println("Friday");
        break;
    case 6:
        System.out.println("Saturday");
        break;
    case 7:
        System.out.println("Sunday");
        break;
}
```





# Les boucles

## For

Utilisée quand le nombre d'itérations est fixe

```
for (statement 1; statement 2; statement 3) {  
    // code block to be executed  
}
```

```
for (int i = 0; i < 5; i++) {  
    System.out.println(i);  
}
```

# Les boucles

## While

Utilisée quand le nombre d'itérations n'est pas fixe

```
while (condition) {  
    // code block to be executed  
}
```

```
int i = 0;  
while (i < 5) {  
    System.out.println(i);  
    i++;  
}
```

# Les boucles

## Do..while

Utilisée quand le nombre d'itérations n'est pas fixe et que vous devez exécuter la boucle au moins une fois

```
do {  
    // code block to be executed  
}  
while (condition);
```

```
int i = 0;  
do {  
    System.out.println(i);  
    i++;  
}  
while (i < 5);
```



# Les boucles

## Foreach

Utilisée exclusivement pour boucler les éléments d'un tableau

```
for (type variableName : arrayName) {  
    // code block to be executed  
}
```

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};  
for (String i : cars) {  
    System.out.println(i);  
}
```



# 4

## Les méthodes

# Les méthodes

Une méthode est un bloc de code qui ne s'exécute que lorsqu'il est appelé.

Les méthodes sont utilisées pour effectuer certaines actions, et elles sont également appelées fonctions.

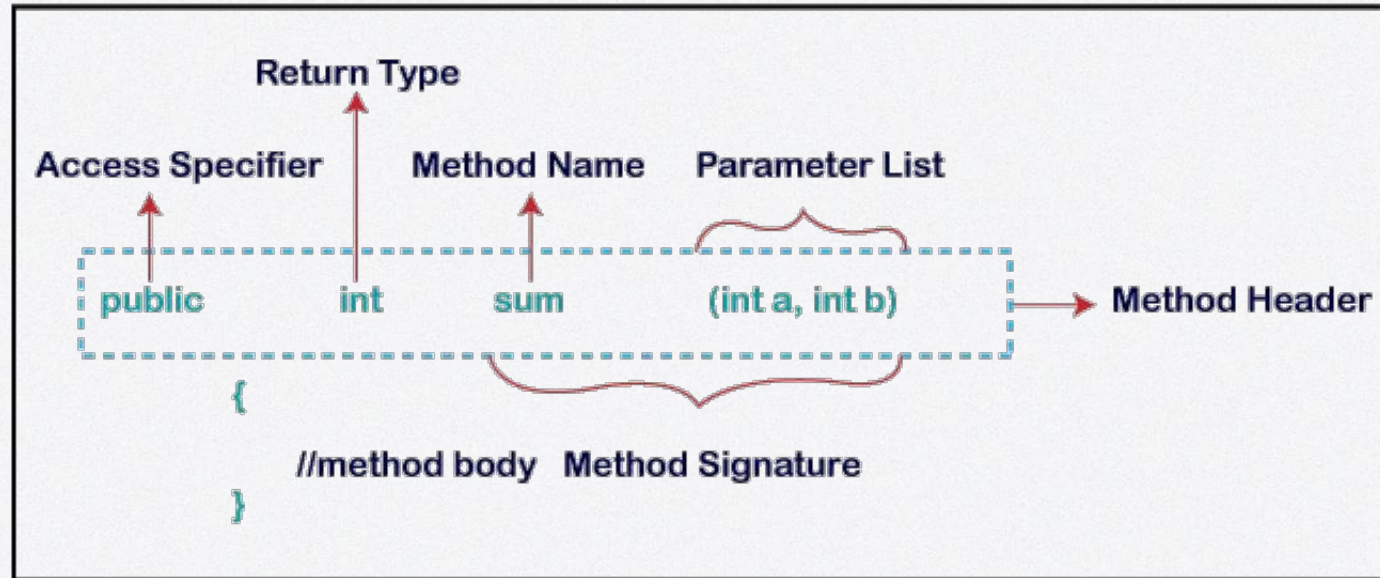
Pourquoi utiliser des méthodes? Pour **réutiliser** le code : définir le code une fois et utilisez-le plusieurs fois.

```
public class Main {  
    static void myMethod() {  
        // code to be executed  
    }  
}
```



# Les méthodes

## Method Declaration



# Les méthodes: Specificateur d'accès

**Access Specifier:** Le spécificateur ou modificateur d'accès est le type d'accès de la méthode. Il spécifie la visibilité de la méthode. Java fournit quatre types de spécificateur d'accès :

- **Public:** La méthode est accessible par toutes les classes lorsque nous utilisons un spécificateur public dans notre application.
- **Private:** Lorsque nous utilisons un spécificateur d'accès privé, la méthode n'est accessible que dans les classes dans lesquelles elle est définie.
- **Protected:** Lorsque nous utilisons un spécificateur d'accès protégé, la méthode est accessible dans le même package ou sous-classes dans un package différent.
- **Default:** Lorsque nous n'utilisons aucun spécificateur d'accès dans la déclaration de méthode, Java utilise par défaut un spécificateur d'accès par défaut. Il n'est visible qu'à partir du même package.

# Les méthodes: Nommage

Lors de la définition d'une méthode, le nom de la méthode doit par une lettre minuscule. Si le nom de la méthode comporte plus de deux mots, le premier mot doit être un verbe suivi d'un adjectif ou d'un nom.

Dans le nom de la méthode multi-mots, la première lettre de chaque mot doit être en majuscules, à l'exception du premier mot

**Nom de la méthode en un seul mot** : `sum()`, `area()`

**Nom de la méthode multi-mots** : `areaOfCircle()`, `stringComparision()`



# Les méthodes: Nommage

C'est la technique de camelCase.



# Les méthodes

```
public class Main {  
    static void myMethod() {  
        System.out.println("I just got executed!");  
    }  
  
    public static void main(String[] args) {  
        myMethod();  
        myMethod();  
        myMethod();  
    }  
}  
  
// I just got executed!  
// I just got executed!  
// I just got executed!
```

# Différence entre les méthodes statiques et les méthodes d'instance

Les méthodes d'instance sont des méthodes qui nécessitent la création d'un objet de sa classe avant de pouvoir être appelé. Les méthodes statiques sont les méthodes en Java qui peuvent être appelées sans créer d'objet de classe.

1. La méthode static est déclarée avec un mot-clé static. La méthode Instance n'est pas avec un mot-clé statique.
2. Méthode statique signifie qui existera en tant que copie unique pour une classe. Mais les méthodes d'instance existent sous forme de copies multiples en fonction du nombre d'instances créées pour cette classe.
3. Les méthodes statiques ne peuvent pas accéder directement aux méthodes d'instance et aux variables d'instance. La méthode Instance peut accéder directement aux variables statiques et aux méthodes statiques.



# Différence entre les méthodes statiques et les méthodes d'instance

```
class A {  
    void fun1() {  
        System.out.println("Hello I am Non-Static");  
    }  
    static void fun2() {  
        System.out.println("Hello I am Static"); }  
}  
class Person {  
    public static void main(String args[]) {  
        A obj=new A();  
        obj.fun1(); // Call non static method  
        A.fun2(); // Call static method  
    }  
}
```

**Output is:**

Hello I am Non-Static  
Hello I am Static



**5**

# **Gestion des erreurs**

# Erreur = exception

L'exception est une **condition anormale**.

En Java, une exception est un événement qui perturbe le flux normal du programme. C'est un objet qui est lancé au moment de l'exécution. Lorsqu'une erreur se produit, Java **s'arrête** normalement et génère un **message d'erreur**. Le terme technique pour cela est: Java lèvera une **exception** (lancer une erreur).



# Les exceptions

```
public class Main {  
    public static void main(String[ ] args) {  
        int[] myNumbers = {1, 2, 3};  
        System.out.println(myNumbers[10]); // error!  
    }  
}
```

# Les exceptions: Try..catch

```
try {  
    // Block of code to try  
}  
catch(Exception e) {  
    // Block of code to handle errors  
}
```

# Les exceptions: Try..catch

```
public class Main {  
    public static void main(String[] args) {  
        try {  
            int[] myNumbers = {1, 2, 3};  
            System.out.println(myNumbers[10]);  
        } catch (Exception e) {  
            System.out.println("Something went wrong.");  
        }  
    }  
}
```



# Les exceptions: Throw

L'instruction vous permet de créer une erreur personnalisée.

L'instruction est utilisée avec un type d'exception. Il existe de nombreux types d'exceptions disponibles en Java

- `ArithmeticException`,
- `FileNotFoundException`,
- `ArrayIndexOutOfBoundsException`,
- `SecurityException`
- ...

# Les exceptions: Throw

```
public class Main {  
    static void checkAge(int age) {  
        if (age < 18) {  
            throw new ArithmeticException("Access denied - You must be at least 18 years old.");  
        }  
        else {  
            System.out.println("Access granted - You are old enough!");  
        }  
    }  
  
    public static void main(String[] args) {  
        checkAge(15); // Set age to 15 (which is below 18...)  
    }  
}
```



**6**

# **Les tableaux**



# Les tableaux

Les tableaux sont utilisés pour stocker plusieurs valeurs dans une seule variable, au lieu de déclarer des variables distinctes pour chaque valeur.

```
String[] aArray = new String[5];  
String[] bArray = {"a", "b", "c", "d", "e"};  
String[] cArray = new String[]{"a", "b", "c", "d", "e"};
```

# Top méthodes pour les tableaux

## 1. Imprimer un tableau

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};  
for (String i : cars) {  
    System.out.println(i);  
}
```

# Top méthodes pour les tableaux

## 1. Imprimer un tableau

```
int[] intArray = { 1, 2, 3, 4, 5 };  
String intArrayString = Arrays.toString(intArray);
```

*// print directly will print reference value*

```
System.out.println(intArray);
```

*// [I@7150bd4d*

```
System.out.println(intArrayString);
```

*// [1, 2, 3, 4, 5]*



# Top méthodes pour les tableaux

## 2. Créer un ArrayList à partir d'un tableau

```
String[] stringArray = { "a", "b", "c", "d", "e" };  
ArrayList<String> arrayList = new  
ArrayList<String>(Arrays.asList(stringArray));  
System.out.println(arrayList);  
// [a, b, c, d, e]
```

# Top méthodes pour les tableaux

## 3. Remplissage d'un tableau

```
int ar[] = {2, 2, 2, 2, 2, 2, 2, 2, 2};  
// Fill from index 1 to index 4.  
//Arrays.fill(tableau, startingIndex, ending Index, value);  
Arrays.fill(ar, 1, 5, 10);  
System.out.println(Arrays.toString(ar));
```

# Top méthodes pour les tableaux

4. Vérifier si un tableau contient une certaine valeur

```
String[] stringArray = { "a", "b", "c", "d", "e" };  
boolean b = Arrays.asList(stringArray).contains("a");  
System.out.println(b);  
// true
```



# Top méthodes pour les tableaux

## 5. Concaténer deux tableaux

```
int[] firstArray = {23,45,12,78,4,90,1};  
int[] secondArray = {77,11,45,88,32,56,3};  
int fal = firstArray.length;  
int sal = secondArray.length;  
int[] result = new int[fal + sal];  
//arraycopy(Object src, int srcPos, Object dest, int  
destPos, int length)  
System.arraycopy(firstArray, 0, result, 0, fal);  
System.arraycopy(secondArray, 0, result, fal, sal);  
System.out.println(Arrays.toString(result));
```

# Top méthodes pour les tableaux

6. Joindre les éléments du tableau fourni en une seule chaîne

```
// Apache common lang  
String j = String.join(new String[] { "a", "b", "c" }, ", ");  
System.out.println(j);  
// a, b, c
```

# Top méthodes pour les tableaux

## 7. Convertir un arraylist en un tableau

```
String[] stringArray = { "a", "b", "c", "d", "e" };  
ArrayList<String> arrayList = new  
ArrayList<String>(Arrays.asList(stringArray));  
String[] stringArr = new String[arrayList.size()];  
arrayList.toArray(stringArr);  
for (String s : stringArr)  
    System.out.println(s);
```



# Top méthodes pour les tableaux

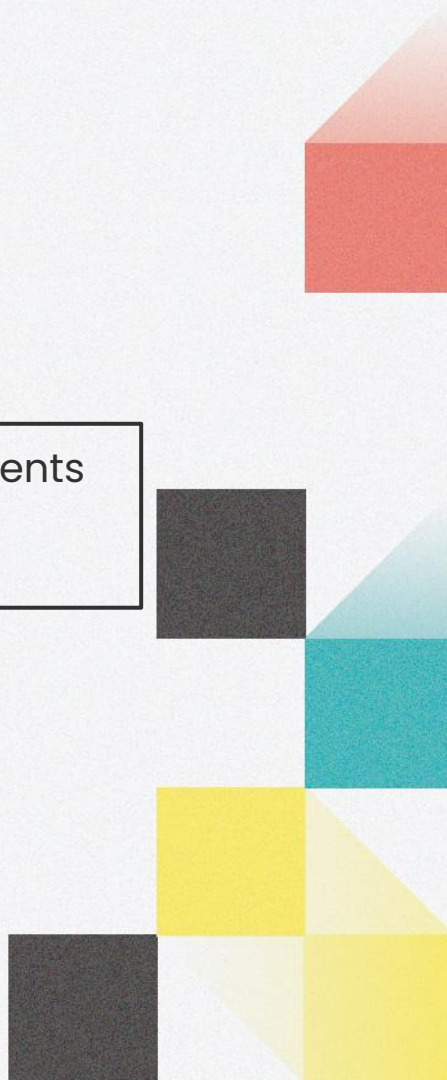
## 8. Inverser un tableau

```
Integer intArray[] = { 1, 2, 3, 4, 5 };  
Collections.reverse(Arrays.asList(intArray));  
System.out.println(Arrays.toString(intArray));  
//[5, 4, 3, 2, 1]
```



# Exercice 1

Écrire un programme Java qui vérifie que les éléments d'un tableau d'entiers sont différents de 0 ou -1.  
[1, 2, 5, 6, 0, 1, 3, 7]

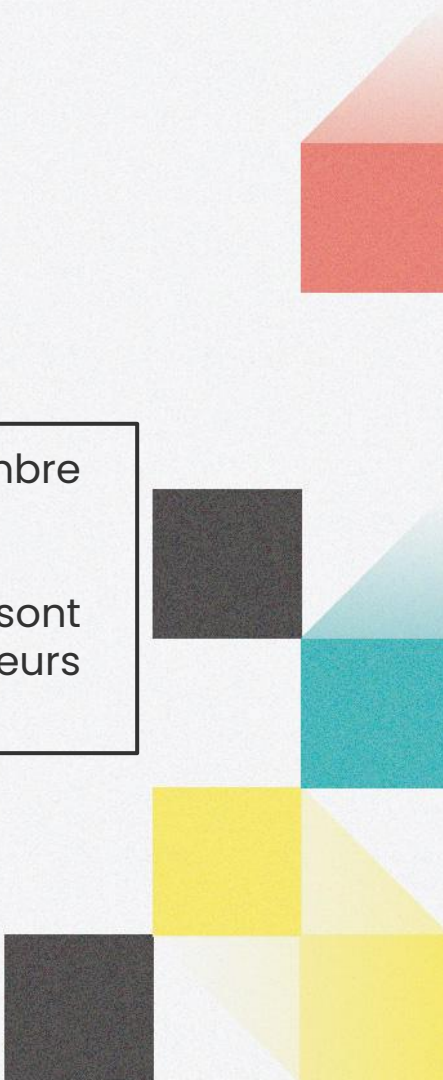




# Exercice 2

Écrire un programme Java qui vérifie si un nombre donné est un nombre moche (ugly number).

Dans le système de nombres, les nombres moches sont des nombres positifs ayant 2, 3 ou 5 comme facteurs premiers.





# Exercice 3

Écrire un programme Java qui trouve le produit maximum de deux entiers dans un tableau donné d'entiers.

Entrée : `nums = { 2, 3, 5, 7, -7, 5, 8, -5 }`

Sortie : La paire est (7, 8), Produit maximum : 56

# Exercice 4

Écrire un programme Java pour mélanger un tableau donné d'entiers.

Entrée : `nums = { 1, 2, 3, 4, 5, 6 }`

Sortie : `Shuffle Array : [4, 2, 6, 5, 1, 3]`

# Exercice 5

Écrire un programme Java qui réorganise un tableau donné d'éléments uniques de sorte qu'un élément sur deux du tableau soit supérieur à ses éléments gauche et droit.

Entrée : `nums = { 1, 2, 4, 9, 5, 3, 8, 7, 10, 12, 14 }`

Sortie : Le tableau avec chaque seconde élément est supérieur à ses éléments gauche et droit :

`[1, 4, 2, 9, 3, 8, 5, 10, 7, 14, 12]`





**7**

**User input (Scanner)**

# User input

La classe Scanner utilisée pour obtenir l'entrée utilisateur et se trouve dans le `package.Scannerjava.util`

```
import java.util.Scanner;

class Main {
    public static void main(String[] args) {
        Scanner input= new Scanner(System.in);
        System.out.println("Enter username");

        String userName = input.nextLine();
        System.out.println("Username is: " + userName);
    }
}
```

# User input

Method	Description
<code>nextBoolean()</code>	Reads a <code>boolean</code> value from the user
<code>nextByte()</code>	Reads a <code>byte</code> value from the user
<code>nextDouble()</code>	Reads a <code>double</code> value from the user
<code>nextFloat()</code>	Reads a <code>float</code> value from the user
<code>nextInt()</code>	Reads a <code>int</code> value from the user
<code>nextLine()</code>	Reads a <code>String</code> value from the user
<code>nextLong()</code>	Reads a <code>long</code> value from the user
<code>nextShort()</code>	Reads a <code>short</code> value from the user



```
import java.util.Scanner;
class Main {

    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.println("Enter name, age and
salary:");
        String name = input .nextLine();
        int age = input .nextInt();
        double salary = input .nextDouble();
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Salary: " + salary);
    }
}
```



# 7

## Les listes (ArrayList)

# Les listes (ArrayList)

C'est un tableau redimensionnable, qui se trouve dans le package `ArrayList` `java.util`.

Les éléments peuvent être ajoutés et supprimés d'un quand vous le souhaitez.

La différence entre une liste et un tableau en Java, c'est que la taille d'un tableau ne peut pas être modifiée.

```
import java.util.ArrayList; // import the ArrayList class
```

```
ArrayList<String> cars = new ArrayList<String>(); // Create an ArrayList object
```



# Les opérations sur les listes

Ajouter des éléments: **add()**

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> cars = new ArrayList<String>();
        cars.add("Volvo");
        cars.add("BMW");
        cars.add("Ford");
        cars.add("Mazda");
        System.out.println(cars);
    }
}
```

# Les opérations sur les listes

Imprimer la liste avec une boucle for

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> cars = new ArrayList<String>();
        cars.add("Volvo");
        cars.add("BMW");
        cars.add("Ford");
        cars.add("Mazda");

        for (int i = 0; i < cars.size(); i++) {
            System.out.println(cars.get(i));
        }
    }
}
```

# Les opérations sur les listes

Accéder à un élément: **get()**

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> cars = new ArrayList<String>();
        cars.add("Volvo");
        cars.add("BMW");
        cars.add("Ford");
        cars.add("Mazda");
        System.out.println(cars.get(0));
    }
}
```



# Les opérations sur les listes

Ajouter tous les éléments: **addAll()**

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> cars_one = new
ArrayList<String>();
        cars_one.add("Volvo");
        cars_one.add("BMW");
        ArrayList<String> cars_two = new
ArrayList<String>();
        cars_two.add("Ford");
        cars_two.add("Mazda");
        cars_one.addAll(cars_two );
    }
}
```

# Les opérations sur les listes

Modifier un élément: **set()**

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> cars = new ArrayList<String>();
        cars.add("Volvo");
        cars.add("BMW");
        cars.add("Ford");
        cars.add("Mazda");
        cars.set(0, "Opel");
    }
}
```

# Les opérations sur les listes

Supprimer tous les éléments: **removeAll()**

```
public static void main(String [] args) {
    ArrayList<String> days = new ArrayList<>();

    days.add("Monday");
    days.add("Tuesday");
    days.add("Wednesday");
    System.out.println("Days: " + days);

    days.removeAll(days);
    System.out.println("ArrayList after
removeAll(): " + days);
}
```



# Les opérations sur les listes

Vider toutes la liste: **clear()**

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> cars = new ArrayList<String>();
        cars.add("Volvo");
        cars.add("BMW");
        cars.add("Ford");
        cars.add("Mazda");
        cars.clear();
    }
}
```

# Les opérations sur les listes

Obtenir la taille de la liste: **size()**

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> cars = new ArrayList<String>();
        cars.add("Volvo");
        cars.add("BMW");
        cars.add("Ford");
        cars.add("Mazda");
        cars.size();
    }
}
```

# Les opérations sur les listes

Ordonner la liste dans un ordre croissant: **sort()**

```
import java.util.ArrayList;
import java.util.Collections;

public class Main {
    public static void main(String args[]) {
        ArrayList<String> list = new ArrayList<String>();
        list.add("Data Science");
        list.add("Testing");
        list.add("C#");
        list.add("Basic Language");
        Collections.sort(list);
    }
}
```



# Les opérations sur les listes

Ordonner la liste dans un ordre décroissant: **reverseOrder()**

```
import java.util.ArrayList;
import java.util.Collections;

public class Main {
    public static void main(String args[]) {
        ArrayList<String> list = new ArrayList<String>();
        list.add("Data Science");
        list.add("Testing");
        list.add("C#");
        list.add("Basic Language");
        Collections.sort(list, Collections.reverseOrder());
    }
}
```

# Les opérations sur les listes

Vérifier l'existence d'un élément dans une list: **contains()**

```
public class Main {  
    public static void main(String args[]) {  
        ArrayList<String> aList = new ArrayList<String>();  
        aList.add("A");  
        aList.add("B");  
        aList.add("C");  
        aList.add("D");  
        aList.add("E");  
        if(aList.contains("C"))  
            System.out.println("The element C is available in the  
ArrayList");  
        else  
            System.out.println("The element C is not available in the  
ArrayList");  
        if(aList.contains("H"))  
            System.out.println("The element H is available in the  
ArrayList");  
        else  
            System.out.println("The element H is not available in the  
ArrayList");  
    }  
}
```

# Exercice 1

Écrire un programme Java ayant une `arrayList` `programmingLanguages` contenant les éléments suivants C, C++, Java, Kotlin, Python, Perl et Ruby.

- Supprimer le 5ème élément.
- Supprimer l'élément "kotlin".
- Supprimer les langages script: Python, Perl et Ruby.
- Vider toute la liste.



# Exercice 2

Écrire un programme Java qui compare deux listes. C'est à dire vérifier si les éléments de la première liste existe dans la deuxième.

1,2,3,4,5

1,4,7,8,9

1 existe? Oui


5 existe? Non

- Si l'element existe: imprimer "nom\_element exists ? Oui"
- Sinon imprimer "nom\_element exists ? Non"



# Exercice 3

Écrire un programme Java qui enregistre les numéros de téléphones fournis par l'utilisateur dans un répertoire et les affiche.



The image features a light gray background with decorative geometric elements in the corners. The top-left corner contains a cluster of squares in shades of red, orange, and black. The bottom-right corner features a cluster of squares in shades of green, teal, and black. The text 'La POO' is centered in the middle of the image.

**La POO**



# Plan

**1**

Pourquoi la  
POO ?

**2**

Classe  
et objet

**3**

Principes  
de la POO





**1**

**Pourquoi la POO ?**

# Pourquoi la POO?

La programmation orientée objet (POO) est l'un des paradigmes de programmation les plus utilisés.

Pourquoi est-il largement utilisé?

- Bien adapté à la création d'applications complexes
- Permet la réutilisation du code, augmentant ainsi la productivité
- De nouvelles fonctionnalités peuvent être facilement intégrées au code existant
- Réduction des coûts de production et de maintenance





**2**

# **Classe et objet**

# Qu'est-ce que classe en Java ?

Les classes sont un **Blueprint** ou un ensemble d'instructions pour **créer un type d'objet spécifique**. La classe en Java détermine le comportement d'un objet et ce qu'il contiendra.

Une classe est un groupe d'objets qui ont des propriétés communes. Il s'agit d'un modèle ou d'un plan à partir duquel les objets sont créés.

Une classe en Java peut contenir :

- Champs
- Méthode
- Constructeurs
- Classe et interface imbriquées

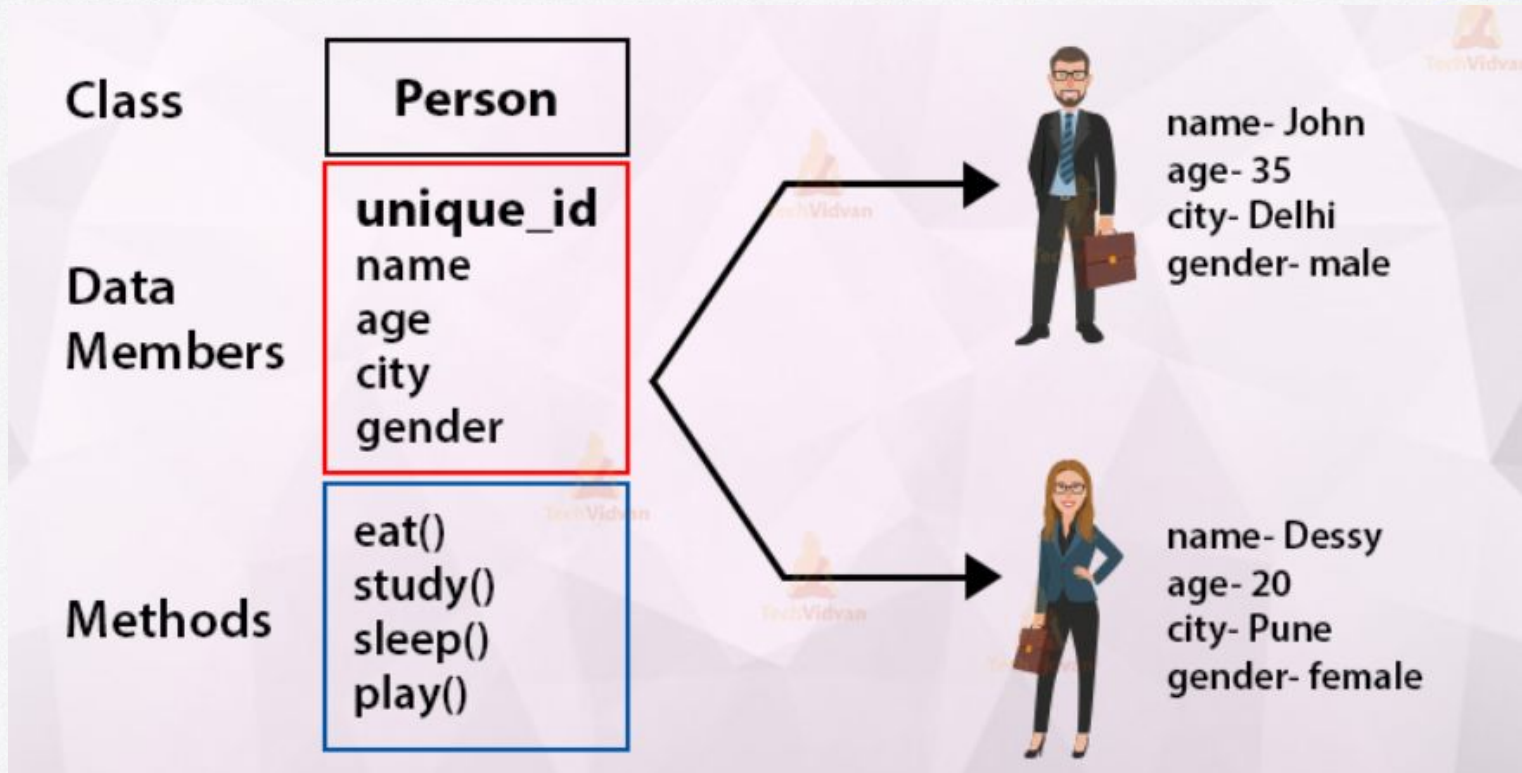
# Qu'est-ce qu'un objet ?

L'objet est une **instance d'une classe**. Un objet dans OOPS n'est rien d'autre qu'un composant autonome qui se compose de méthodes et de propriétés pour rendre un type particulier de données utiles. Par exemple nom de couleur, table, sac, aboiement.

Du point de vue de la programmation, un objet dans OOPS peut inclure une structure de données, une variable ou une fonction. Il a un emplacement de mémoire alloué.







```
public class Person
    private String name;
    private int age;

    public person(String name, int age) {
        this.name = name;
        this.age = age
    }
    void eat(){
        //methodBody
    }
    void study(){
        //methodBody
    }
    void play(){
        //methodBody
    }
}
```

# Le constructeur



Un constructeur est une méthode invoquée lors de **la création d'un objet**. Cette méthode effectue les opérations nécessaires à l'initialisation d'un objet. Chaque constructeur doit **avoir le même nom que la classe** où il est défini et n'a aucune valeur de retour.



# Types de constructeur

```
public class School
```

```
{
```

```
// Zero parameter constructor.
```

```
School() {
```

```
// Body of constructor 1.
```

```
}
```

```
// One parameter constructor.
```

```
School(String name) {
```

```
// Body of constructor 2.
```

```
}
```

```
// Two parameters constructor.
```

```
School(String name, int rollNo) {
```

```
// Body of constructor 3.
```

```
}
```

```
.....
```

```
}
```

Three  
constructors  
overloaded  
having a  
different  
parameter  
list

Fig: Overloaded constructors based on parameter list.

```
// Class Declaration
```

```
class Dog {
```

```
    // Instance Variables
```

```
    String breed;
```

```
    String size;
```

```
    int age;
```

```
    String color;
```

← Les attributs

```
    // method 1
```

```
    public String getInfo() {
```

← Une méthode

```
        return ("Breed is: "+breed+" Size is:"+size+" Age is:"+age+" color is: "+color);
```

```
    }
```

```
}
```

```
public class Execute{
```

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

```
        Dog maltese = new Dog();
```

← Création d'un objet avec le mot clé **new**

```
        maltese.breed="Maltese";
```

```
        maltese.size="Small";
```

```
        maltese.age=2;
```

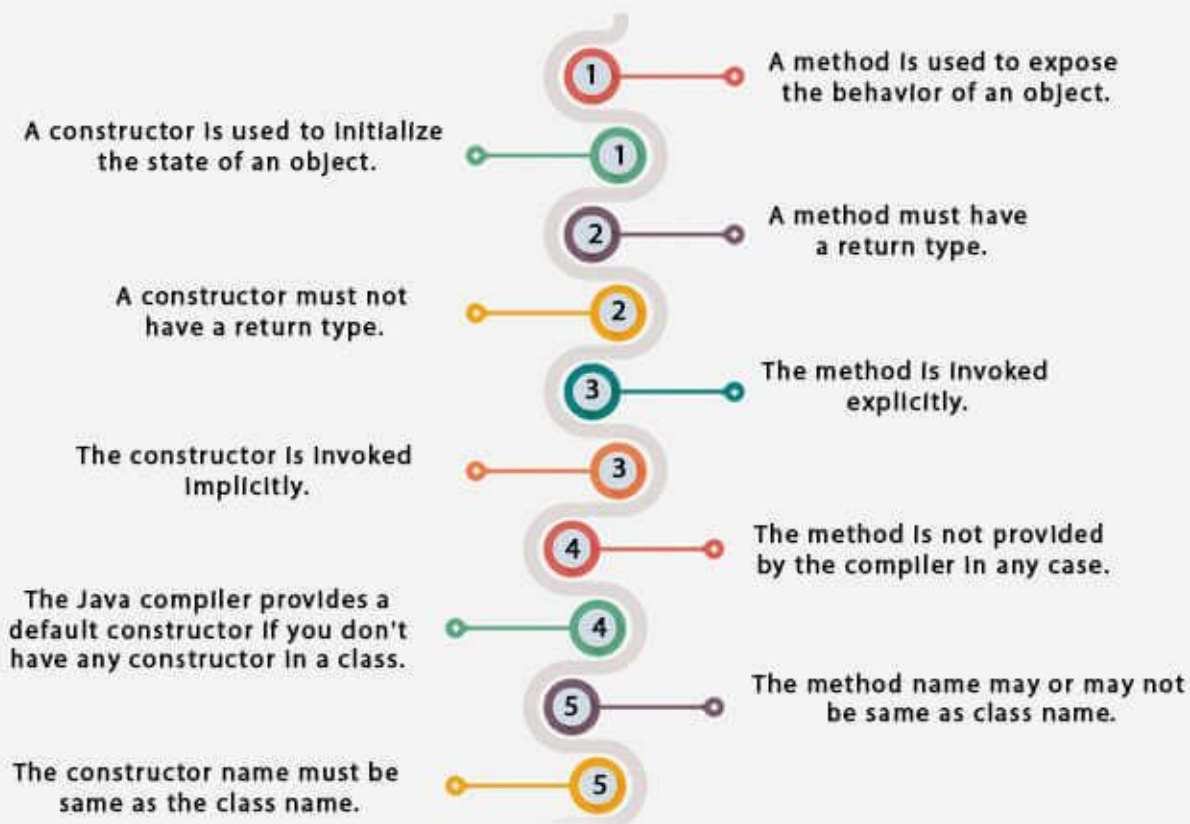
```
        maltese.color="white";
```

```
        System.out.println(maltese.getInfo());
```

```
    }
```

```
}
```

# Difference between constructor and method in Java







# 2

## Principes de la POO

# 1. Encapsulation

La signification de Encapsulation, est de s'assurer que les données « sensibles » sont **cachées** aux utilisateurs. Pour ce faire, il faut:

- déclarer les variables/attributs de classe comme **private**
- fournir des méthodes publiques **get** and **set** pour accéder à la valeur d'une variable et la mettre à jour

```
public class Person {  
    private String name; // private = restricted access  
  
    // Getter  
    public String getName() {  
        return name;  
    }  
  
    // Setter  
    public void setName(String newName) {  
        this.name = newName;  
    }  
}
```

# 1. Encapsulation

```
public class Main {  
    public static void main(String[] args) {  
        Person myObj = new Person();  
        myObj.name = "John"; // error  
        System.out.println(myObj.name); // error  
    }  
}
```

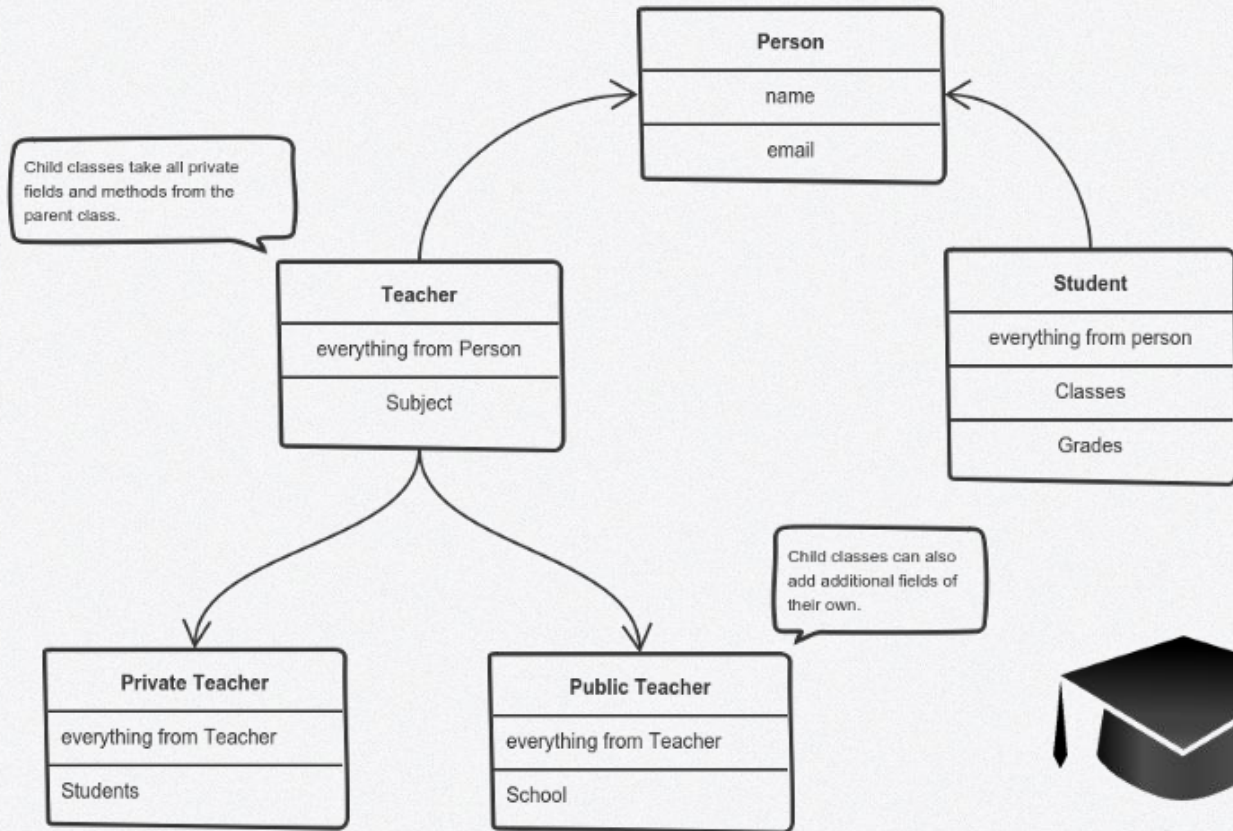


```
public class Main {  
    public static void main(String[] args) {  
        Person myObj = new Person();  
        myObj.setName("John"); // Set the value of the name variable to "John"  
        System.out.println(myObj.getName());  
    }  
}
```





## 2. Héritage



## 2. Héritage

```
class Vehicle {  
    protected String brand = "Ford";  
  
    public void honk() {  
        System.out.println("Tuut, tuut!");  
    }  
}  
  
class Car extends Vehicle {  
    private String modelName = "Mustang";  
  
    public static void main(String[] args) {  
        Car myCar = new Car();  
        myCar.honk();  
  
        System.out.println(myCar.brand + " " + myCar.modelName);  
    }  
}
```

## 2. Héritage

Le mot-clé **Super** en Java est une variable de référence qui est utilisée pour faire référence à l'objet de classe parente immédiat.

Chaque fois que vous créez l'instance de sous-classe, une instance de classe parente est créée implicitement qui est référencée par une variable de super référence.

Utilisation du mot clé super:

1. super peut être utilisé pour faire référence à une variable d'instance de classe parente immédiate.
2. super peut être utilisé pour appeler la méthode de classe parente immédiate.
3. super() peut être utilisé pour appeler le constructeur de classe parente immédiate.



## 2. Héritage

```
/* superclass Person */
class Person {
    Person() {
        System.out.println("Person class Constructor");
    }
}

/* subclass Student extending the Person class */
class Student extends Person{
    Student(){
        // invoke or call parent class constructor
        super();
        System.out.println("Student class Constructor");
    }
}

/* Driver program to test*/
class Test {
    public static void main(String[] args){
        Student s = new Student();
    }
}
```

## 2. Héritage

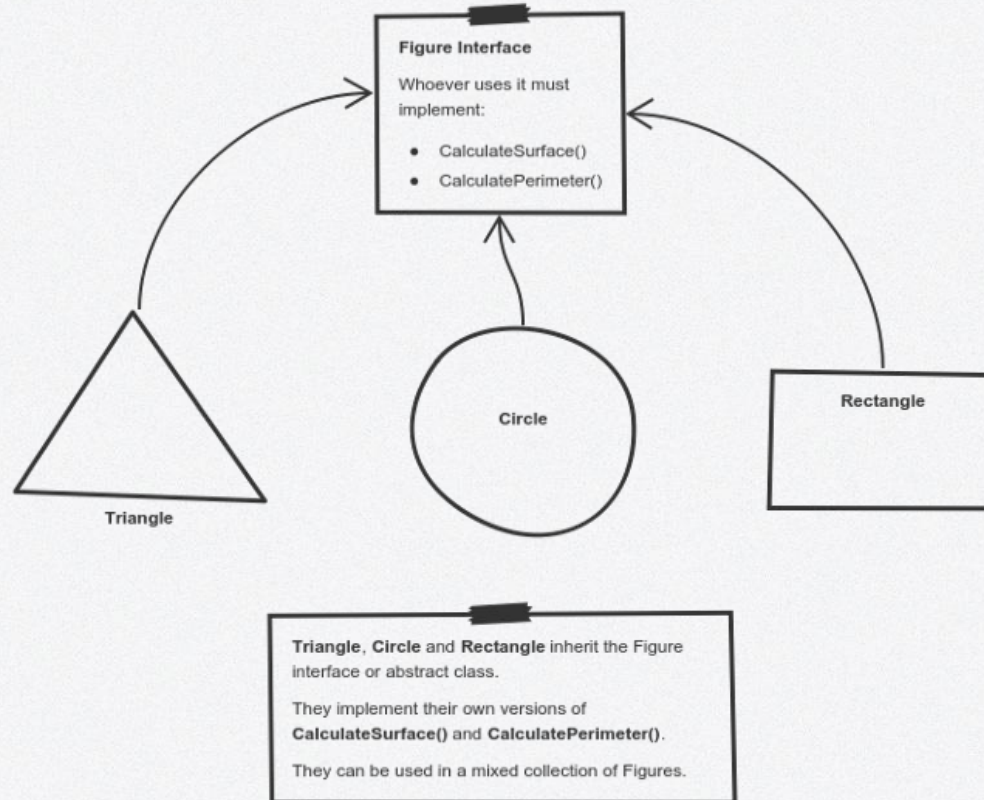
```
class Person{
    int id;
    String name;
    Person(int id,String name){
        this.id=id;
        this.name=name;
    }
}

class Emp extends Person{
    float salary;
    Emp(int id,String name,float salary){
        super(id,name);//reusing parent constructor
        this.salary=salary;
    }
    void display(){System.out.println(id+" "+name+" "+salary);}
}

class Execute{
    public static void main(String[] args){
        Emp e1=new Emp(1,"sarah",45000f);
        e1.display();
    }
}
```

# 3. Polymorphisme

Le mot polymorphisme signifie avoir de nombreuses formes.



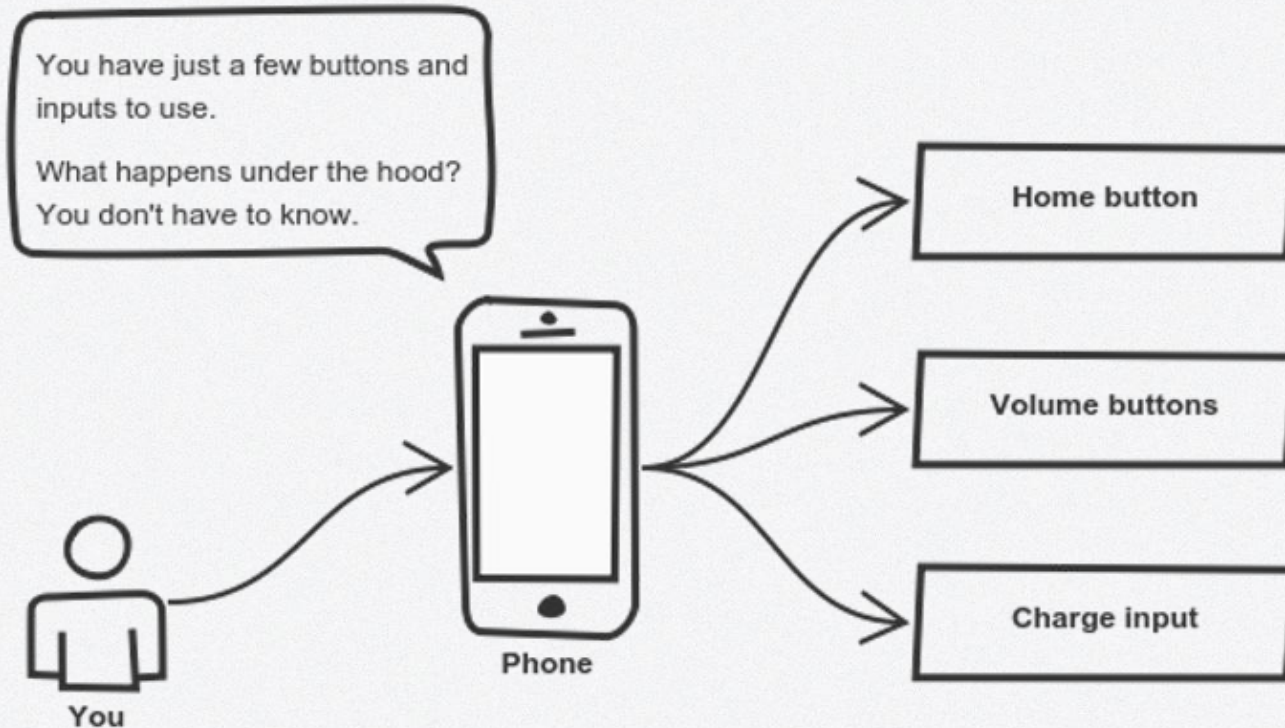


### 3. Polymorphisme

```
class Animal {
    public void animalSound() {
        System.out.println("The animal makes a sound");
    }
}
class Pig extends Animal {
    public void animalSound() {
        System.out.println("The pig says: wee wee");
    }
}
class Dog extends Animal {
    public void animalSound() {
        System.out.println("The dog says: bow wow");
    }
}
class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Animal(); // Create a Animal object
        Animal myPig = new Pig(); // Create a Pig object
        Animal myDog = new Dog(); // Create a Dog object
        myAnimal.animalSound();
        myPig.animalSound();
        myDog.animalSound();
    }
}
```

## 4. Abstraction

L'abstraction est un processus qui consiste à masquer les détails de l'implémentation et à n'afficher que les fonctionnalités à l'utilisateur.



## 4. Abstraction: Les classes

Une classe qui est déclarée comme abstraite est connue sous le nom de classe abstraite. Elle peut avoir **des méthodes abstraites et non abstraites**.

**Elle ne peut pas être instanciée.**

Points à retenir

- Une classe abstraite doit être déclarée avec un mot-clé `abstract`.
- Elle peut avoir des méthodes abstraites et non abstraites.
- Elle ne peut pas être instanciée.
- Elle peut également avoir des constructeurs et des méthodes statiques.
- Elle peut avoir des méthodes finales qui forceront la sous-classe à ne pas modifier le corps de la méthode.



## 4. Abstraction: Les classes

```
abstract class Animal {
    // Abstract method (does not have a body)
    public abstract void animalSound();
    // Regular method
    public void sleep() {
        System.out.println("Zzz");
    }
}

class Pig extends Animal {
    public void animalSound() {
        // The body of animalSound() is provided here
        System.out.println("The pig says: wee wee");
    }
}

class Main {
    public static void main(String[] args) {
        Pig myPig = new Pig(); // Create a Pig object
        myPig.animalSound();
        myPig.sleep();
    }
}
```

# La différence entre l'abstraction et l'héritage

## La Classe abstraite :

- L'abstraction masque les détails de l'implémentation et affiche uniquement les fonctionnalités à l'utilisateur.
- L'abstraction permet de réduire la complexité du code.
- Nous ne pouvons pas créer d'objets d'une classe abstraite.

## Héritage:

- L'héritage est la méthodologie de création d'une nouvelle classe à l'aide des propriétés et des méthodes d'une classe existante.
- L'héritage permet d'améliorer la réutilisabilité du code.
- Nous pouvons créer l'objet de la classe parente.

## 4. Abstraction: Les interfaces

L'interface en Java est un mécanisme pour réaliser **l'abstraction**. Il ne peut y avoir **que des méthodes abstraites** dans l'interface Java, **pas dans le corps** de la méthode.

Elle est utilisée pour réaliser l'abstraction et l'héritage multiple en Java.

```
// interface
interface Animal {
    public void animalSound();
    public void run();
}
```



## 4. Abstraction: Les interfaces

```
interface Animal {
    public void animalSound(); // interface method (does not have a body)
    public void sleep(); // interface method (does not have a body)
}

class Pig implements Animal {
    public void animalSound() {
        System.out.println("The pig says: wee wee");
    }
    public void sleep() {
        System.out.println("Zzz");
    }
}

class Main {
    public static void main(String[] args) {
        Pig myPig = new Pig(); // Create a Pig object
        myPig.animalSound();
        myPig.sleep();
    }
}
```

# Exercice 1

Écrire un programme Java ayant

- une classe "**Car**" dont les attributs sont **model** et **price** et les fonctionnalités **start**, **stop** et **move**.
  - La méthode start retourne la chaîne de caractère: "Car starts"
  - La méthode stop retourne la chaîne de caractère: "Car stops"
  - La méthode move retourne la chaîne de caractère: "Car moves"
- Une classe "**driver**" dont les attributs sont **name** et **age** et la fonctionnalité **drive**
  - La méthode move retourne la chaîne de caractère: "Driver drives car"

## Exercice 2

On veut réaliser un programme de gestion des recettes de cuisine, qui sera installé sur des appareils électroménagers pour leur permettre de faire la cuisine de façon autonome.

La classe Ingrédient est donnée ci-dessous :

```
class Ingredient{  
    String nom_aliment, etat;  
    int quantite;  
    String unite;  
    Ingredient(String n, String e, int q, String unite){  
        this.nom_aliment = n;  
        this.etat = e;  
        this.quantite = q;  
        this.unite = unite;  
    }  
}
```



# Exercice 2

## Partie 1

- Écrire une classe Plat qui représente les plats, chaque plat ayant **un nom** et **une liste d'ingrédients**.
- On doit pouvoir créer un plat avec son **nom**.
- Il faut également avoir des **accesseurs** sur le nom du plat et les ingrédients, et **pouvoir ajouter un ingrédient** à un plat.
- Écrire également une méthode main qui crée un plat appelé choucroute contenant comme ingrédients : 500g de choucroute cuite, 150g de lard cuit et entier et 2 saucisses entières et cuites

## Exercice 2

### Partie 2 (indice: héritage)

On veut faire la distinction entre les ingrédients qu'**on peut cuire** et ceux qu'**on peut découper**.

- Un ingrédient qu'on peut cuire doit avoir une méthode cuire() qui le fait passer dans l'état "cuit" et une température de cuisson.
- Un ingrédient qu'on peut découper doit avoir une méthode decouper() qui le fait passer dans l'état "découpé".

## Exercice 3

On veut réaliser un programme qui représente des montres et les personnes qui les portent.

- Une montre donne l'heure et les minutes.
- On peut initialiser une montre à partir d'un couple heure/ minute donnée.

Écrire une classe qui représente les montres telles que décrites au dessus.

- Une personne a un nom et peut éventuellement porter une montre.
- Une personne peut porter une montre (`porterMontre()` ) si elle n'en a pas déjà une.
- Une personne peut enlever sa montre (`enleverMontre()` )
- Une personne peut donner l'heure (`donnerHeure()`)

Écrire une classe qui représente les personnes telles que décrites au dessus.



## Exercice 4

Ecrire un programme java qui gère le coût des enseignants d'une université. Chaque enseignant a un nom, un prénom et un certain nombre d'heures de cours assurées dans l'année.

Il existe plusieurs catégories d'enseignants.

- Les enseignants-chercheurs sont payés avec un salaire fixe (2000 euros par mois pour simplifier) plus le revenu des heures complémentaires (40 euros de l'heure). Les heures complémentaires sont les heures effectuées au-delà des 192h. Ainsi, un enseignant-chercheur qui donne 200h de cours dans l'année recevra  $2000 \times 12 + 8 \times 40 = 24320$  euros sur l'année.
- Les vacataires sont des enseignants venant d'un organisme extérieur à l'université. Cet organisme est décrit par une chaîne de caractères. Les vacataires sont payés 40 euros de l'heure pour toutes leurs heures. Les étudiants de 3e cycle (doctorants) peuvent assurer des cours et sont payés 30 euros de l'heure, mais ne peuvent dépasser 96h de cours.
- Un doctorant qui fait plus de 96h ne recevra que le salaire correspondant à 96h de cours.
- Finalement, sur tous les salaires versés, on applique des charges égales à un certain pourcentage du salaire, ce pourcentage est le même pour tous les enseignants (par exemple 100%, ce qui signifie que le montant des charges est égal au montant des salaires).

On veut, pour chaque enseignant, pouvoir connaître ce qu'il coûte à l'université (salaire + charges).