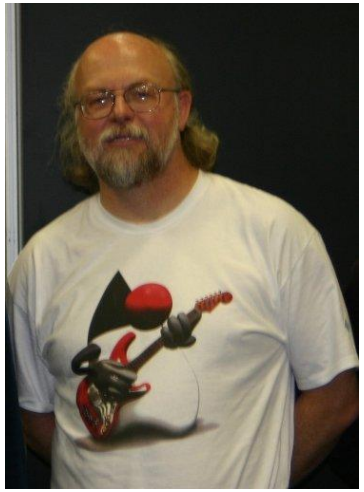
< Introduction au langage Java>

< Alejandro SEIJO>

<qu'est-ce que Java>



James Gosling

- Est un langage de programmation orienté objet créé par James Gosling, Mike Sheridan, employés de Sun Microsystems, avec le soutien de Bill Joy (cofondateur de Sun Microsystems en 1982)
- Présenté officiellement le 23 mai 1995 au SunWorld, Java est devenu l'un des langages de programmation les plus populaires et influents du monde.

<caractéristiques>

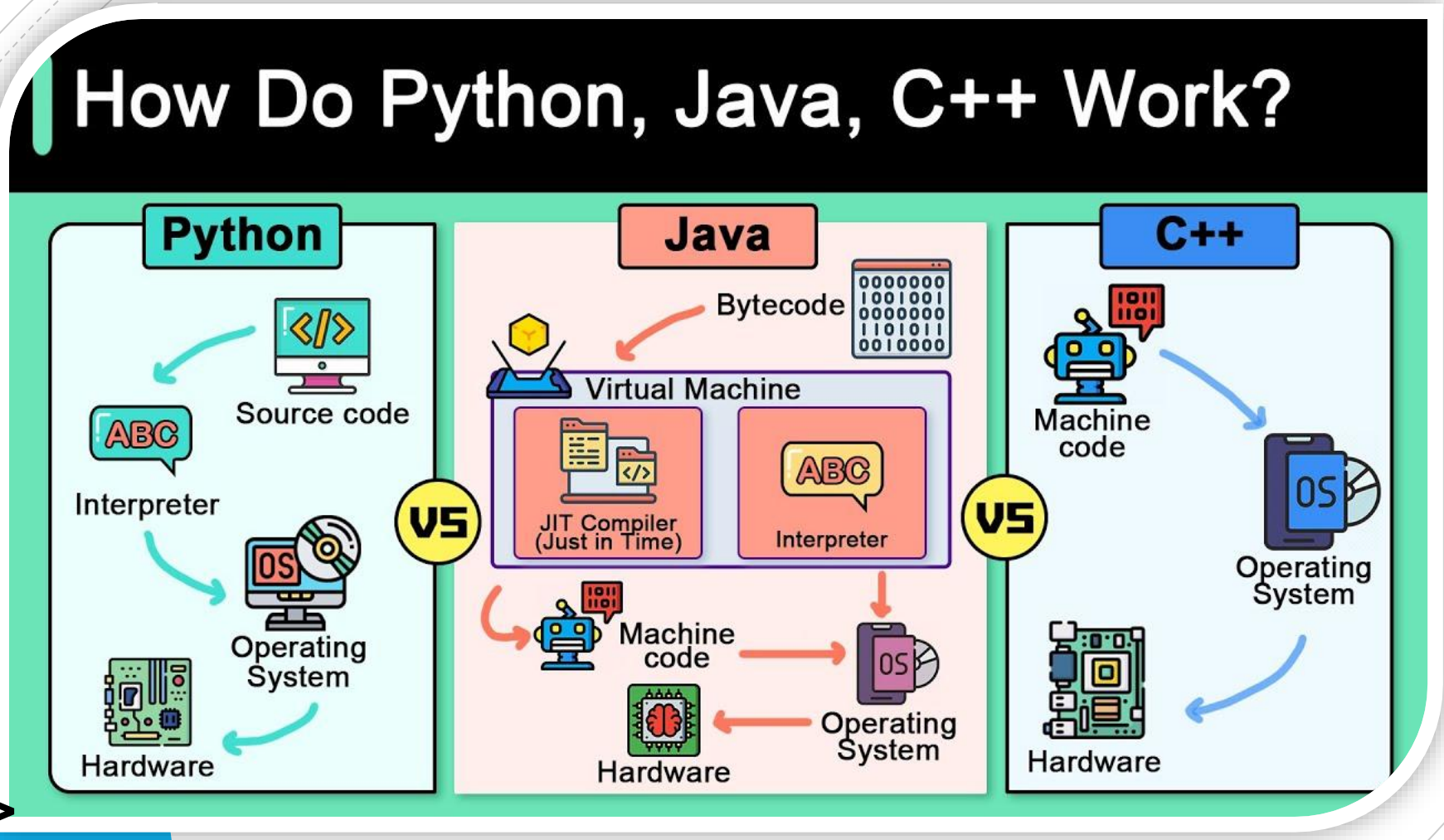
Portabilité:

- La **portabilité** en Java est l'une des caractéristiques les plus importantes et les plus vantées du langage. Elle signifie qu'un programme écrit en Java peut être exécuté sur n'importe quel système d'exploitation ou plateforme qui dispose d'une JVM (Java Virtual Machine), sans modification du code source. Cela permet aux développeurs de "coder une fois, exécuter partout" (*write once, run anywhere*).

Langage Orienté Objet

- Le langage orienté objet en Java repose sur le concept de "programmation orientée objet" (POO), qui est un paradigme de programmation utilisant des "objets" pour concevoir des applications et des logiciels. Java est un langage entièrement orienté objet, ce qui signifie que presque tout ce que vous faites en Java implique des objets.

< portabilité >



<caractéristiques>

Sécurité:

- Java est conçu avec un certain nombre de fonctionnalités de sécurité intégrées. Par exemple, la gestion de la mémoire automatisé.

Robustesse

- Le système de gestion des erreurs de Java est conçu pour rendre les programmes plus robustes. Les exceptions sont utilisées pour gérer les erreurs, ce qui permet aux programmes de continuer à s’exécuter même en cas d’erreur.

<la plateforme Java>

JDK

La **JDK** (Java Development Kit) est un ensemble d'outils et de bibliothèques nécessaires pour développer et exécuter des applications Java. Elle est fournie par Oracle et d'autres fournisseurs comme OpenJDK, et elle constitue l'un des composants essentiels pour tout développeur Java.

Le compilateur Java (javac) :

- Le compilateur traduit le code source Java (fichiers .java) en bytecode (fichiers .class) que la JVM (Java Virtual Machine) pourra exécuter. Le bytecode est un format intermédiaire indépendant du système d'exploitation, ce qui rend Java portable.

La JVM (Java Virtual Machine) :

- Bien que la JVM ne fasse techniquement pas partie de la JDK (elle fait partie de la JRE, Java Runtime Environment), elle est incluse avec la JDK. La JVM exécute le bytecode produit par le compilateur et fournit un environnement d'exécution pour les applications Java. C'est ce qui rend Java "write once, run anywhere" (écrire une fois, exécuter partout).

JRE (Java Runtime Environment) :

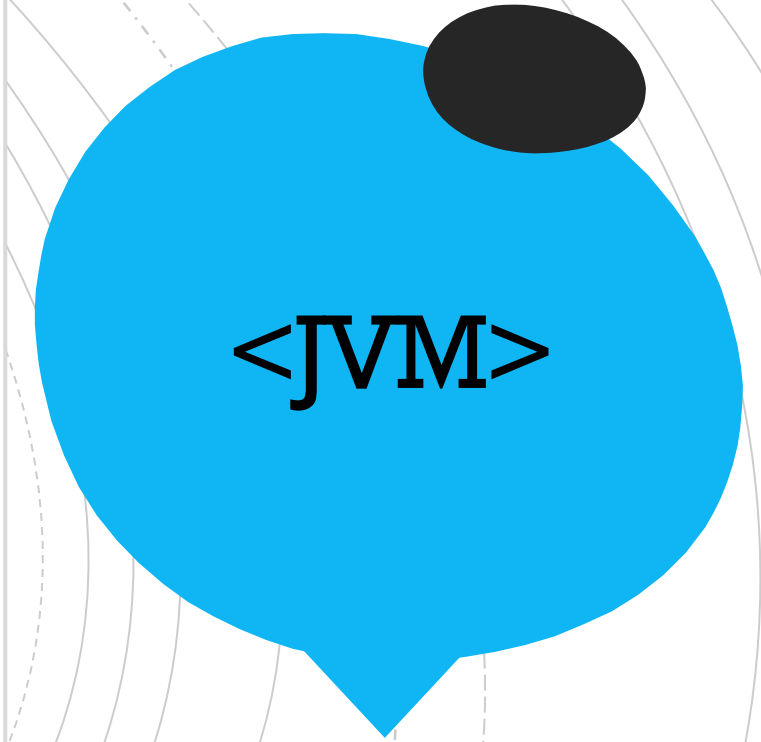
- La JRE est incluse dans la JDK et contient tout ce qui est nécessaire pour exécuter des applications Java, y compris la JVM et les bibliothèques de base. Cependant, elle ne contient pas les outils de développement comme le compilateur.

La **JVM** (Java Virtual Machine) est un composant clé de la plateforme Java. Elle permet l'exécution des programmes Java en transformant le bytecode Java en instructions machine exécutables par l'ordinateur hôte.

Fonctionnement de la JVM

- Lorsqu'un programme Java est compilé, il n'est pas transformé directement en code machine. Au lieu de cela, il est converti en bytecode, un format intermédiaire stocké dans des fichiers .class.

- La JVM charge ces fichiers .class, interprète le bytecode, et le transforme en instructions machine que le système peut exécuter. Elle utilise un interpréteur ou un compilateur à la volée (Just-In-Time Compiler ou JIT) pour optimiser l'exécution en temps réel.



Composants principaux de la JVM

- **Class Loader:**
  - La JVM utilise un class loader pour charger dynamiquement les fichiers de classes (.class) dans la mémoire lors de l'exécution. Ce composant gère également le lien entre les différentes classes.

- **Mémoire d'exécution (Runtime Data Areas) :**

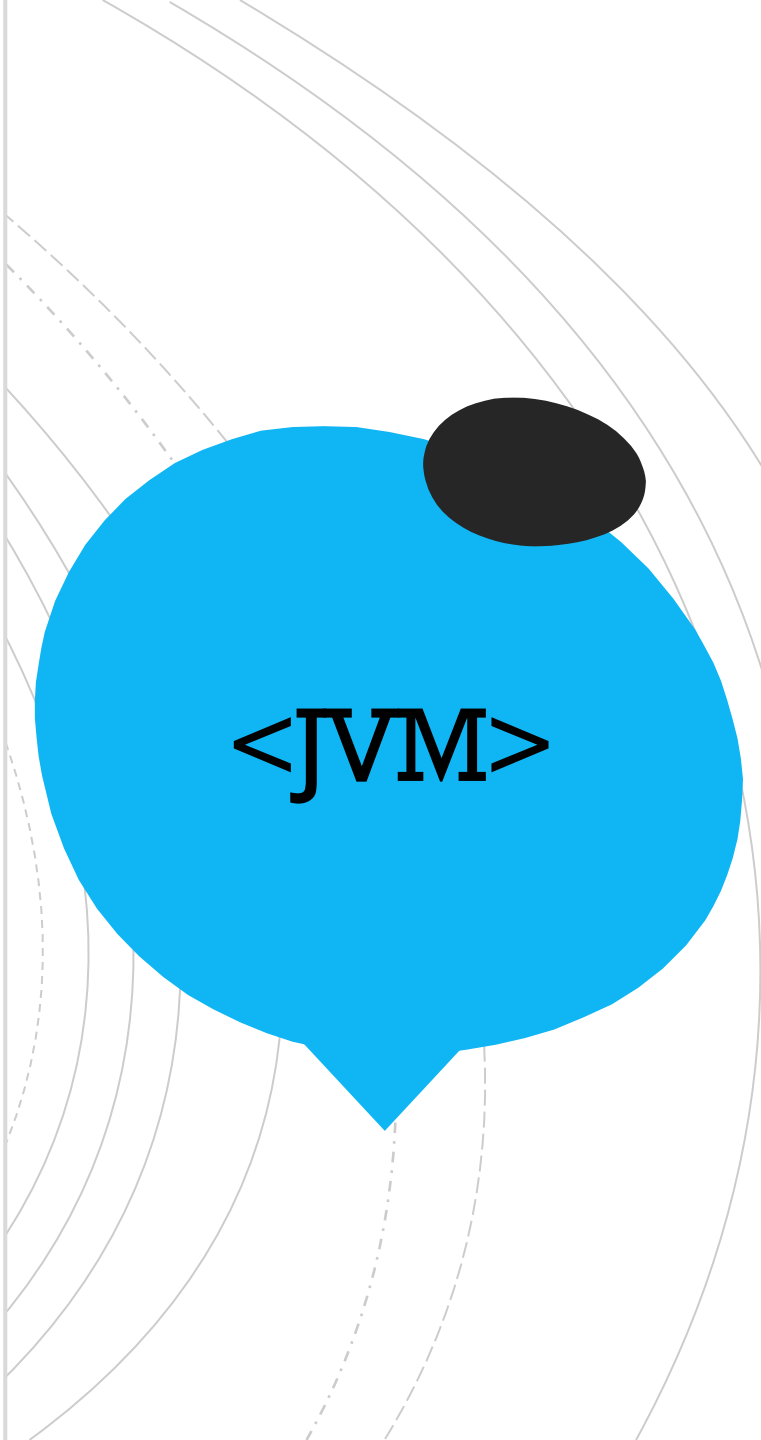
La JVM utilise plusieurs zones de mémoire pour exécuter les programmes :

- **Heap** : où sont stockés les objets et les variables de classe.
- **Stack** : où sont stockées les variables locales et les résultats partiels pour chaque thread en cours d'exécution.

- **Execution Engine** :

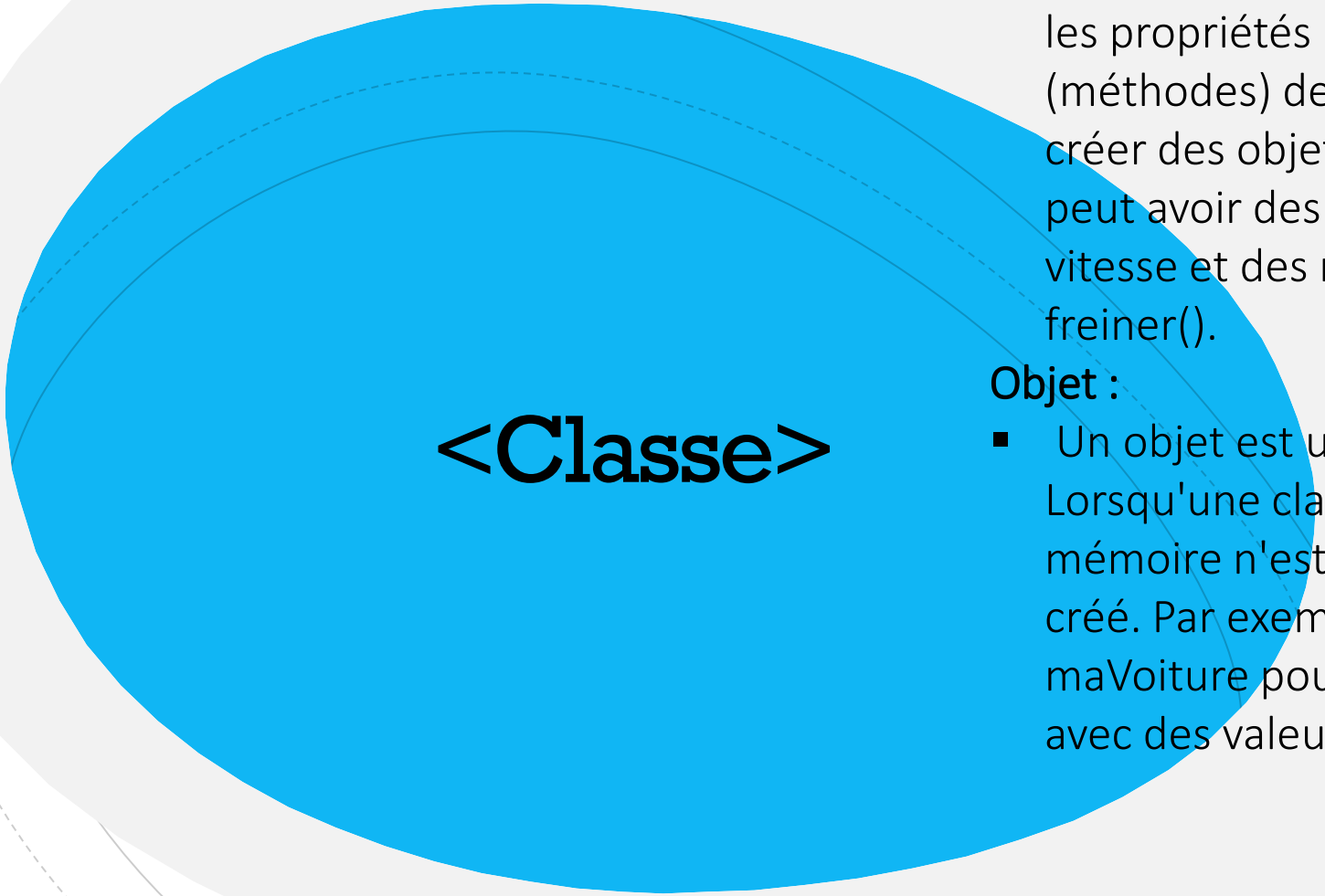
Cette composante est responsable de l'exécution du bytecode Java.

- **Interpréteur :** lit et exécute les instructions du bytecode ligne par ligne. Cependant, l'interprétation ligne par ligne peut être lente.
- **JIT (Just-In-Time Compiler) :** améliore les performances en compilant des blocs de bytecode en code machine natif au moment de l'exécution, ce qui permet une exécution plus rapide.
- **Garbage Collector (GC) :** gère automatiquement la mémoire en récupérant les objets qui ne sont plus utilisés, libérant ainsi de l'espace mémoire pour de nouveaux objets.



<commencer>

- Installer le kit de développement Java (JDK)
- Télécharger l'IDE



Classe:

- Une classe est un modèle ou un plan qui définit les propriétés (attributs) et les comportements (méthodes) des objets. Elle sert de modèle pour créer des objets. Par exemple, une classe Voiture peut avoir des attributs comme couleur, marque, vitesse et des méthodes comme accélérer(), freiner().

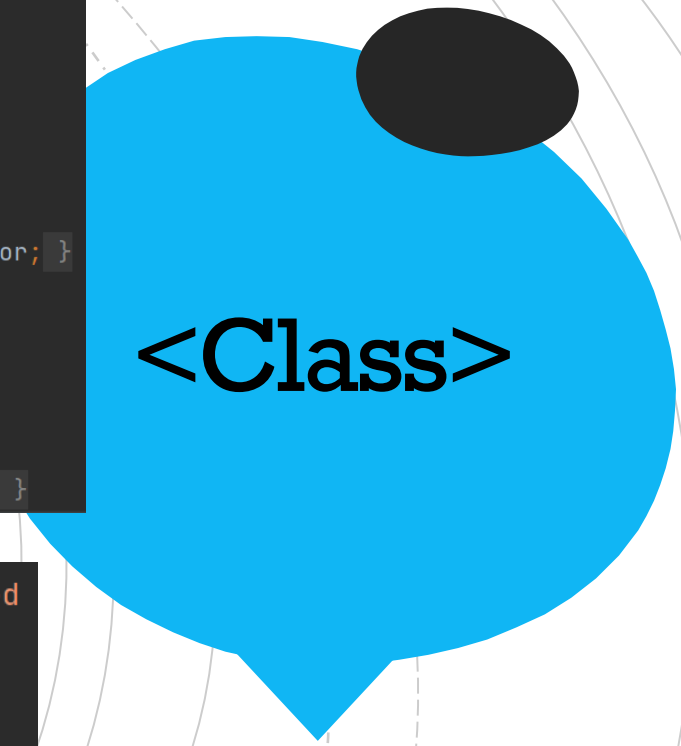
Objet :

- Un objet est une instance d'une classe. Lorsqu'une classe est définie, aucun espace mémoire n'est alloué jusqu'à ce qu'un objet soit créé. Par exemple, si Voiture est une classe, maVoiture pourrait être un objet de cette classe avec des valeurs spécifiques pour ses attributs.

```java
no usages
public class Car {
// Attributs (encapsulation)
    3 usages
    private String brand;
    3 usages
    private String color;
    5 usages
    int speed;

//    Default Constructor (empty)
    no usages
    public Car() {
    }
//    Overloaded Constructor
    no usages
    public Car(String brand, String color, int speed) {
        this.brand = brand;
        this.color = color;
        this.speed = speed;
    }
```

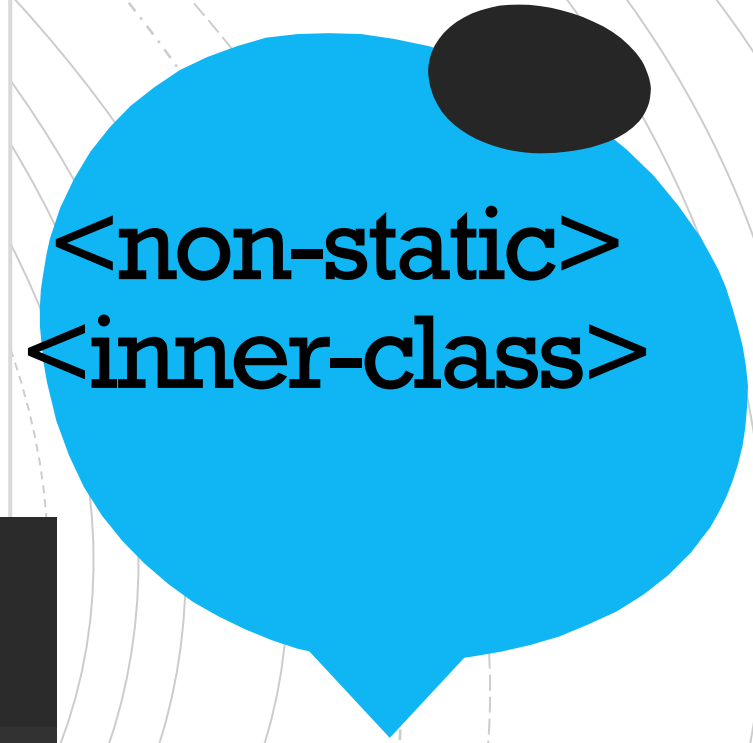
```java
//    Getters and Setters
    no usages
    public String getBrand() {
        return brand;
    }

    no usages
    public void setBrand(String brand) {
        this.brand = brand;
    }

    no usages
    public String getColor() { return color; }

    no usages
    public void setColor(String color) { this.color = color; }

    no usages
    public int getSpeed() { return speed; }

    no usages
    public void setSpeed(int speed) { this.speed = speed; }
```

```java
//    Methods for increase and decrease speed
    no usages
    public void speedUp(int increment){
        speed += increment;
    }
    no usages
    public void slowdown(int decrement){
        speed -= decrement;
    }
```



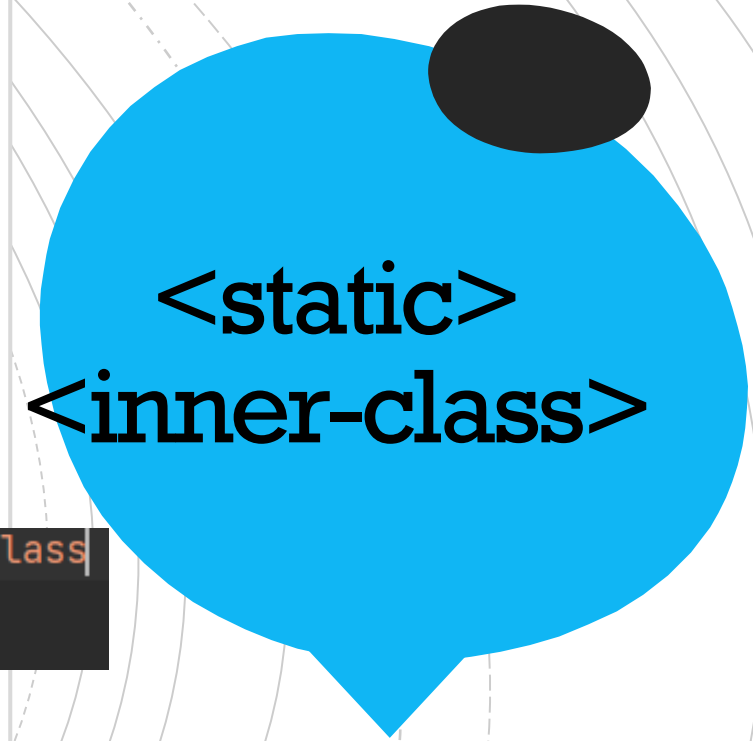
```java
//    InnerClass non-Static
    2 usages
    public class InnerClass{
//        Inner Attributs
        no usages
        int innerNumber = 8;

//        Inner Methods
    1 usage
    public void innerSound(){
        System.out.println("A race car makes: eeeeee !");
    }
}
```

```java
Car car = new Car();
car.carSound();

  InnerClass innerClass = new InnerClass(); // can not access directly to inner class
  Car.InnerClass innerClass1 = new Car.InnerClass(); // neither
Car.InnerClass innerClass = car.new InnerClass(); // access to inner class from object ("car")
innerClass.innerSound();
```

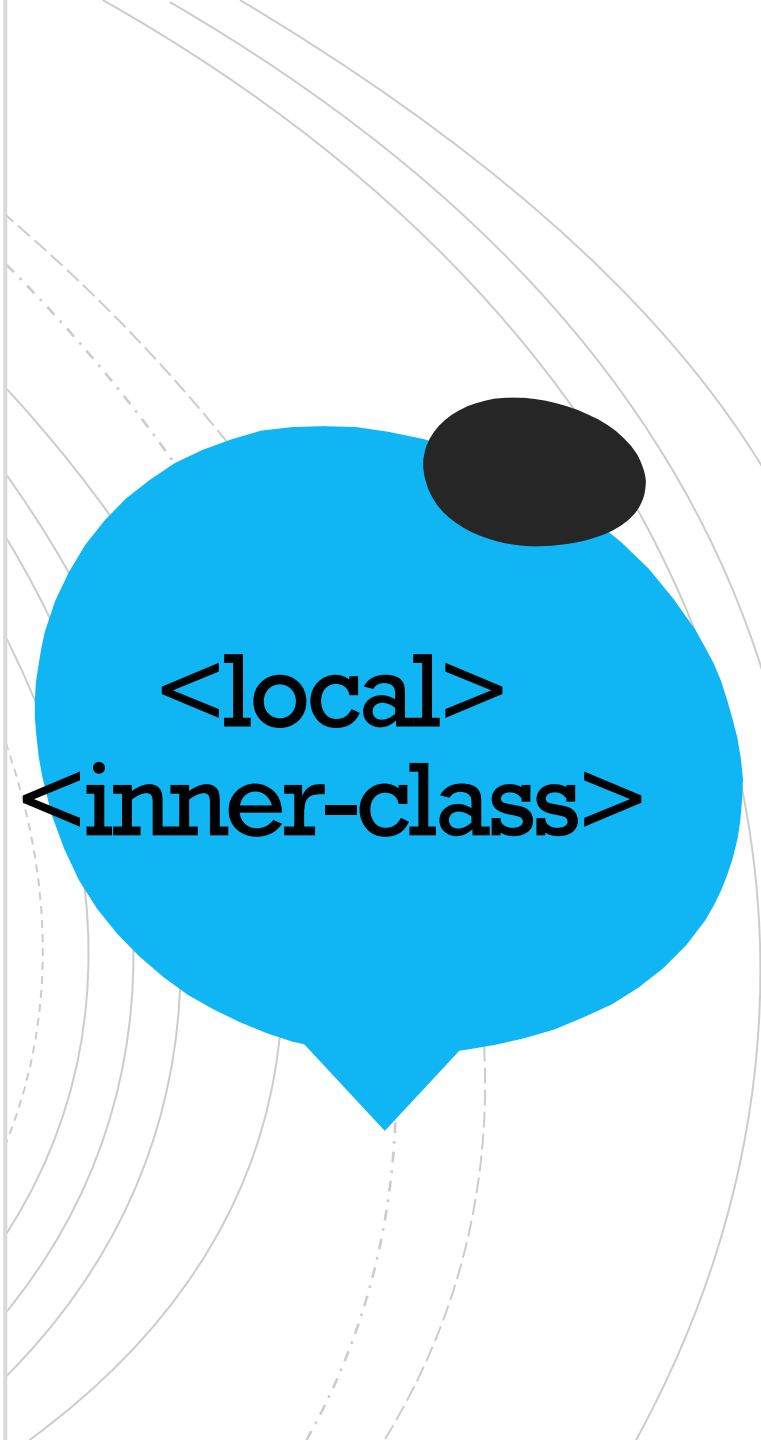


```java
//    InnerClass static
    2 usages
    public static class InnerClass{
//        Inner Attributs
        no usages
        int innerNumber = 8;

//        Inner Methods
    1 usage
    public void innerSound(){
        System.out.println("A race car makes: eeeeee !");
    }
}
```

```java
Car.InnerClass innerClass = new Car.InnerClass(); //access to static inner class
innerClass.innerSound();
```



```java
//    Local inner class
    1 usage
    public void carSound(){
        System.out.println("A car makes: vooom vooom !");

        // Declaration of local class
        2 usages
        class LocalInnerClass{
//          Attributs of local inner class
            1 usage
            String localInnerClassVariable = "This is a local inner class sound: oiiiii !";

//          Method of local inner class
            1 usage
            public void printLocalInnerClassVariable(){
                System.out.println(localInnerClassVariable);
            }
        }
//      LocalInnerClass only accessible from method
        LocalInnerClass lic = new LocalInnerClass();
        lic.printLocalInnerClassVariable();
    }
```



<notion de classe>

Encapsulation :

- L’encapsulation consiste à regrouper les données (attributs) et les méthodes qui les manipulent au sein d’une même unité, la classe. Elle protège l'état interne d'un objet en n'exposant les attributs qu'à travers des méthodes spécifiques (getters et setters).
- En Java, cela se fait en déclarant les attributs comme private et en fournissant des méthodes public (comme getNom() et setNom()) pour y accéder ou les modifier. Cela empêche l'accès direct aux attributs, garantissant ainsi leur intégrité.

Héritage :

- L’héritage est un mécanisme qui permet de créer une nouvelle classe à partir d'une classe existante. La nouvelle classe, appelée classe dérivée ou sous-classe, hérite des attributs et des méthodes de la classe de base ou super-classe.
- Cela favorise la réutilisation du code. Par exemple, si vous avez une classe Animal, vous pouvez créer des sous-classes comme Chien et Chat qui héritent des attributs et des comportements communs à tous les animaux, mais qui peuvent aussi définir leurs propres caractéristiques et méthodes spécifiques.
- En Java, l’héritage est implémenté avec le mot-clé **extends**.

Polymorphisme :

- Le polymorphisme permet à une même méthode de se comporter différemment en fonction de l'objet qui l'appelle. En Java, il existe deux types de polymorphisme :
- **Polymorphisme à la compilation (surcharge) :** Cela signifie que vous pouvez avoir plusieurs méthodes avec le même nom dans une même classe, mais avec des paramètres différents (types ou nombre). Par exemple, vous pouvez avoir plusieurs versions de la méthode additionner(int a, int b) ou additionner(double a, double b, double c).
- **Polymorphisme à l'exécution (sous-typage) :** Il permet d'utiliser un objet de sous-classe là où un objet de super-classe est attendu. Par exemple, si Chien est une sous-classe de Animal, vous pouvez traiter un objet Chien comme un Animal grâce à l'héritage, tout en appelant des méthodes spécifiques à Chien.

<notion de classe>

Abstraction :

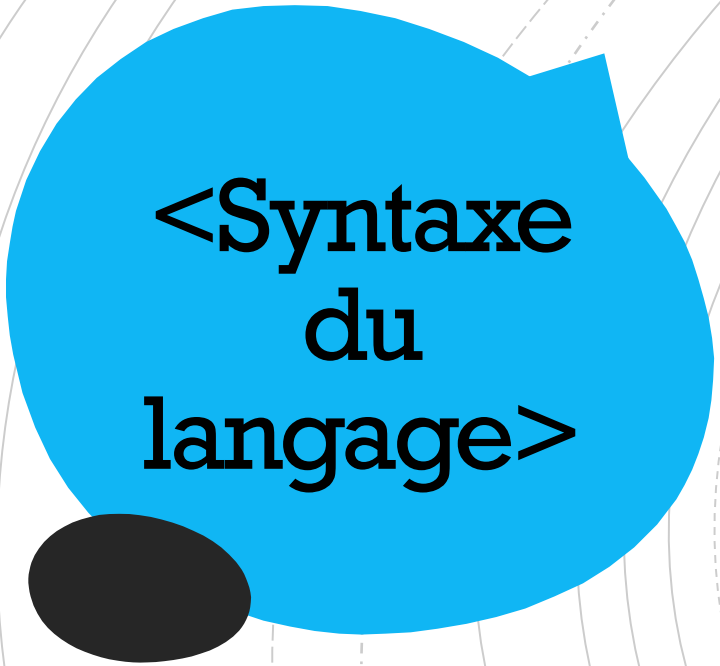
- L'abstraction est le concept qui consiste à se concentrer uniquement sur les caractéristiques essentielles d'un objet tout en cachant les détails complexes ou secondaires.
- En Java, cela se fait de deux manières :

Classes abstraites :

- Une classe abstraite est une classe qui ne peut pas être instanciée directement. Elle peut contenir des méthodes abstraites (sans implémentation) que les sous-classes doivent définir. Par exemple, une classe Forme pourrait être abstraite avec une méthode calculerAire() que des sous-classes comme Cercle ou Rectangle implémentent.

Interfaces :

- Une interface est un ensemble de méthodes que les classes peuvent implémenter. Les interfaces permettent de définir un contrat que les classes doivent respecter, sans spécifier comment les méthodes seront implémentées. En Java, une classe peut implémenter plusieurs interfaces, favorisant ainsi la flexibilité et le polymorphisme.



Le langage C a servi de base pour la syntaxe du langage Java :

- le caractère de fin d'une instruction est " ;"

```
a = c + c ;
```

- les commentaires (non traités par le compilateur) se situent entre les symboles "/*" et "*/" ou commencent par le symbole "//" en se terminant à la fin de la ligne.

```
int a ; // ce commentaire tient sur une ligne
int b ;
```

ou

```
/*Ce commentaire nécessite
2 lignes*/
int a ;
```

- les identificateurs de variables ou de méthodes acceptent les caractères {a..z}, {A..Z}, $, _. Ainsi que les caractères {0..9} s'ils ne sont pas le premier caractère de l'identificateur. Il faut évidemment que l'identificateur ne soit pas un mot réservé du langage (comme int ou for).

Ex :

mon_entier et ok4all sont des identificateurs valides mais

mon-entier et 4all ne sont pas valides pour des identificateurs.

<primitive data types >

TABLE 2.1 – Type primitifs de données en Java

Type	Classe éq.	Valeurs	Portée	Défaut
`boolean`	`Boolean`	`true` ou `false`	N/A	`false`
`byte`	`Byte`	entier signé	{-128..128}	o
`char`	`Character`	caractère	{\uoooo..\uFFFF}	\uoooo
`short`	`Short`	entier signé	{-32768..32767}	0
`int`	`Integer`	entier signé	{-2147483648..2147483647}	0
`long`	`Long`	entier signé	$\{-2^{31}..2^{31}-1\}$	0
`float`	`Float`	réel signé	$\{-3,4028234^{38}..3,4028234^{38}\}$ $\{-1,40239846^{-45}..1,40239846^{-45}\}$	0.0
`double`	`Double`	réel signé	$\{-1,797693134^{308}..1,797693134^{308}\}$ $\{-4,94065645^{-324}..4,94065645^{-324}\}$	0.0

```
int a ;
double b = 5.0 ;
a = b ;
```

est interdit et doit être écrit de la manière suivante :

```
int a ;
double b = 5.0 ;
a = (int)b ;
```

<Tableaux et matrices>

- Une variable est déclarée comme un tableau dès lors que des crochets sont présents soit après son type, soit après son identificateur. Les deux syntaxes suivantes sont acceptées pour déclarer un tableau d'entiers (même si la première, non autorisée en C, est plus intuitive) :

```
int[] mon_tableau ;

int mon_tableau2[];
```

- Un tableau a toujours une taille fixe qui doit être précisée avant l'affectation de valeurs à ses indices, de la manière suivante :

```
int[] mon_tableau = new int[20];
```

- De plus, la taille de ce tableau est disponible dans une variable length appartenant au tableau et accessible par mon_tableau.length. On peut également créer des matrices ou des tableaux à plusieurs dimensions en multipliant les crochets (ex : int[][] ma_matrice;). À l'instar du C, on accède aux éléments d'un tableau en précisant un indice entre crochets (mon_tableau[3] est le quatrième entier du tableau) et un tableau de taille n stocke ses éléments à des indices allant de O à n-1.

<Chaînes de caractères>

Les chaînes de caractères ne sont pas considérées en Java comme un type primitif ou comme un tableau. On utilise une classe particulière, nommée **String**, fournie dans le package java.lang. Les variables de type **String** ont les caractéristiques suivantes :

- leur valeur ne peut pas être modifiée (**immutability**)
- on peut utiliser l'opérateur + pour concaténer deux chaînes de caractères :

```
String s1 = ”hello” ;

String s2 = ”world” ;

String s3 = s1 + ” ” + s2 ;

//Après ces instructions s3 vaut ”hello world”
```

- l'initialisation d'une chaîne de caractères s'écrit :

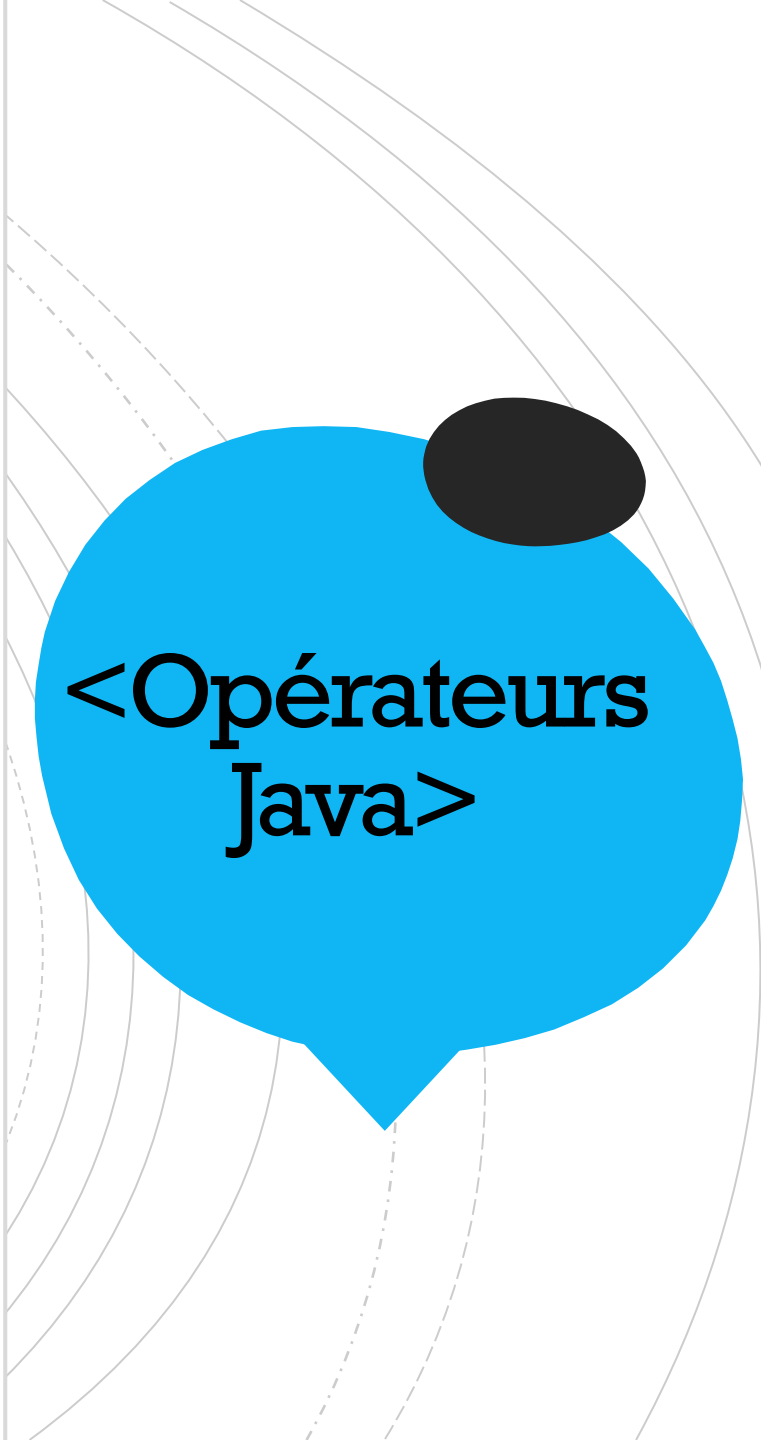
```
String s = new String(); //pour une chaine vide

String s2 = new String(”hello world”);

// pour une chaîne de valeur ”hello world”
```

- un ensemble de méthodes de la classe java.lang.String permettent d'effectuer des opérations ou des tests sur une chaîne de caractères (voir la documentaion de la classe String).

Pr.	Opérateur	Syntaxe	Résultat	Signification
1	++	++<ari>	<ari>	pré incrémentation
		<ari>++	<ari>	post incrémentation
	–	–<ari>	<ari>	pré décrémentation
		<ari>–	<ari>	post décrémentation
	+	+<ari>	<ari>	signe positif
	-	-<ari>	<ari>	signe négatif
	!	!<boo>	<boo>	complément logique
	(*type*)	(*type*)<val>	<val>	changement de type
2	*	<ari>*<ari>	<ari>	multiplication
	/	<ari>/<ari>	<ari>	division
	%	<ari>%<ari>	<ari>	reste de la division
3	+	<ari>+<ari>	<ari>	addition
	-	<ari>-<ari>	<ari>	soustraction
	+	<str>+<str>	<str>	concaténation
4	<<	<ent> << <ent>	<ent>	décalage de bits à gauche
	>>	<ent> >> <ent>	<ent>	décalage de bits à droite
5	<	<ari> < <ari>	<boo>	inférieur à
	<=	<ari> <= <ari>	<boo>	inférieur ou égal à
	>	<ari> > <ari>	<boo>	supérieur à
	>=	<ari> >= <ari>	<boo>	supérieur ou égal à
	instanceof	<val>instanceof<cla>	<boo>	test de type
6	==	<val>==<val>	<boo>	égal à
	!=	<val>!=<val>	<boo>	différent de
7	&	<ent>&<ent>	<ent>	ET bit à bit
		<boo>&<boo>	<boo>	ET booléen
8	^	<ent>^<ent>	<ent>	OU exclusif bit à bit
		<boo>^<boo>	<boo>	OU exclusif booléen
9	|	<ent>|<ent>	<ent>	OU bit à bit
		<boo>|<boo>	<boo>	OU booléen
10	&&	<boo>&&<boo>	<boo>	ET logique
11	||	<boo>||<boo>	<boo>	OU logique
12	?:	<boo>?<ins>:<ins>	<ins>	si...alors...sinon
13	=	<var>=<val>	<val>	assignation

Légende

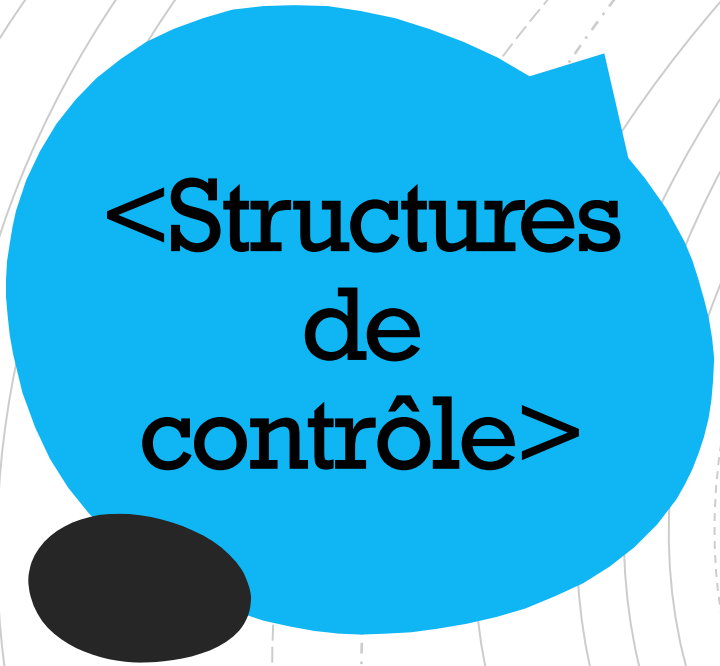
<ari> valeur arithmétique
<boo> valeur booléenne
<cla> classe
<ent> valeur entière
<ins> instruction
<str> chaîne de caractères (String)
<val> valeur quelconque
<var> variable



<Structures de contrôle>

Les structures de contrôle permettent d'exécuter un **bloc d'instructions** soit plusieurs fois (instructions itératives) soit selon la valeur d'une expression (instructions conditionnelles ou de choix multiple). Dans tous ces cas, un bloc d'instruction est:

- soit une instruction unique ;
- soit une suite d'instructions commençant par une accolade ouvrante "{" et se terminant par une accolade fermante "}".



Instructions conditionnelles

- Syntaxe :

if (<condition>) <bloc1> [else <bloc2>]

ou

<condition>?<instruction1>:<instruction2>

- <condition> doit renvoyer une valeur booléenne. Si celle-ci est vraie c'est <bloc1> (resp. <instruction1>) qui est exécuté sinon <bloc2> (resp. <instruction2>) est exécuté. La partie else <bloc2> est facultative.
- Exemple :

```
if (a == b) {
        a = 50 ;
        b = 0 ;
} else {
        a = a - 1 ;
}
```

<Structures de contrôle>

Instructions itératives (While)

Les instruction itératives permettent d'exécuter plusieurs fois un bloc d'instructions, et ce, jusqu'à ce qu'une condition donnée soit fausse. Les trois types d'instruction itératives sont les suivantes :

TantQue...Faire... L'exécution de cette instruction suit les étapes suivantes :

1. la condition (qui doit renvoyer une valeur booléenne) est évaluée. Si celle-ci est vraie on passe à l'étape 2, sinon on passe à l'étape 4;

2. le bloc est exécuté ;

3. retour à l'étape 1 ;

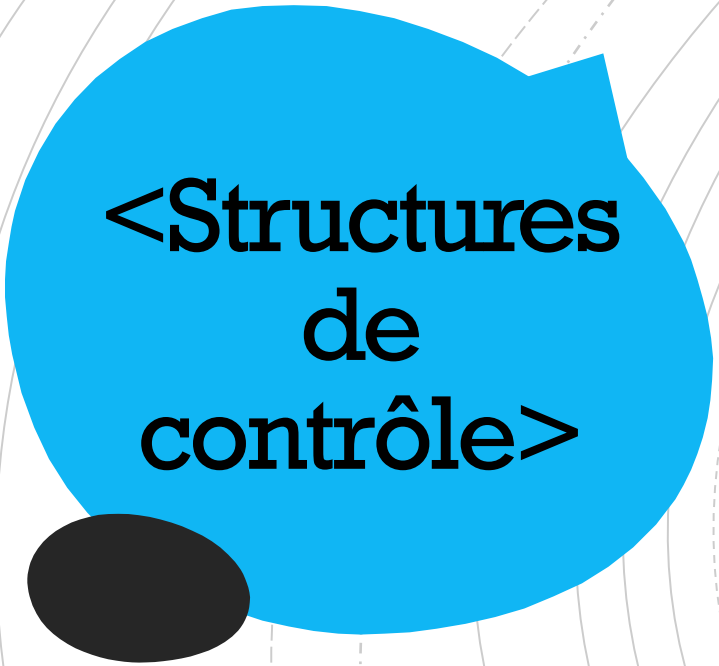
4. la boucle est terminée et le programme continue son exécution en interprétant les instruction suivant le bloc.

- Syntaxe :

while (<condition>) <bloc>

- Exemple :

```
while (a != b) a++;
```



Instructions itératives (Do While)

- Faire...TantQue... L'exécution de cette instruction suit les étapes suivantes :

1. le bloc est exécuté ;

2. la condition (qui doit renvoyer une valeur booléenne) est évaluée. Si celle-ci est vraie on retourne à l'étape 1, sinon on passe à l'étape 3;

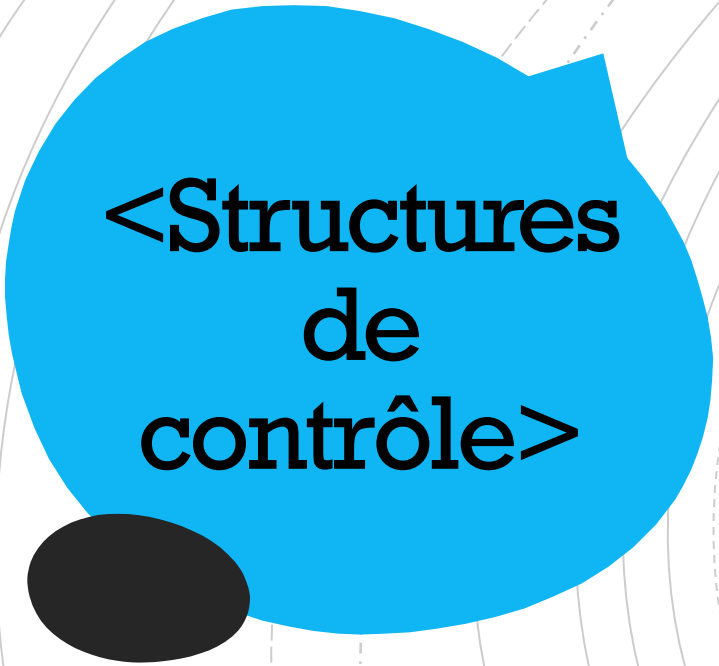
3. la boucle est terminée et le programme continue son exécution en interprétant les instruction suivant le bloc.

- Syntaxe :

do <bloc> while (<condition>);

- Exemple :

```
do a++
while (a != b);
```



Instructions itératives (Do While)

- Faire...TantQue... L'exécution de cette instruction suit les étapes suivantes :

1. le bloc est exécuté ;

2. la condition (qui doit renvoyer une valeur booléenne) est évaluée. Si celle-ci est vraie on retourne à l'étape 1, sinon on passe à l'étape 3;

3. la boucle est terminée et le programme continue son exécution en interprétant les instruction suivant le bloc.

- Syntaxe :

do <bloc> while (<condition>);

- Exemple :

```
do a++
while (a != b);
```