Les bases du langage Java

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-10-31

Table des matières

1 Objectifs d'apprentissage 2
2 Les Types en Java 3
2.1 Types Primitifs 3
2.2 Références 3
3 Les identificateurs en Java 3
4 Les Constantes en Java 4
5 Les Objets et les Références en Java 4
5.1 Les Objets 4
5.2 Les Références 4
6 La Classe Object en Java 5
6.1 Rôle de la classe Object 5
6.2 Méthodes principales de Object 5
7 Les Variables en Java (1/2) 5
7.1 Les différents types de variables 5
8 Variables finales en Java 6
9 Familles de méthodes en Java 6
10 Surcharge de méthodes 7
11 Exemple 7
12 Les paramètres en Java : Passage par valeur 8
12.1 Exemple 8
13 Méthodes et variables de classe 9
13.1 Bloc statique 10
13.2 Pourquoi utiliser les méthodes et variables de classe ? 10
13.3 Exemple 1 10
13.4 Exemple 2 10
14 Protection en Java : Niveaux d'accès 11
14.1 Exemple concret 12
15 Structure d'un programme Java 13

15.1 Exemple : un programme simple 13
16 L'instanciation et les constructeurs 13
16.1 Le constructeur par défaut 13
16.2 Le constructeur paramétré 14
17 Les références en Java : un lien vers l'objet 14
17.1 Exemple 15
18 Le ramasse-miettes en Java : un gestionnaire de mémoire automatique 15
19 Les tableaux en Java : un aperçu 16
19.1 Exemple: 16
20 Les chaînes de caractères en Java : un aperçu 17
20.1 Exemple 17
21 Enveloppement des types primitifs en Java 17
21.1 Exemple 18
22 Expressions, Instructions et Blocs en Java 18
22.1 Blocs 18
22.2 Exemple 18
23 Branchements : Instruction `switch` 18
23.1 Exemple 19
24 Boucles While, Do-While et For en Java 20
25 Boucles ForEach en Java 20
26 Les paquetages en Java 20
27 La structuration d'un programme : Les paquetages (2/3) - Exemple concret . 21
27.1 Exemple 21
27.2 La structuration d'un programme les paquetages (3/3) 22
28 Les paquetages standards en Java 22
29 Les notations 22
30 La Documentation 23
30.1 Exemple 23
31 L'organisation (3/3) 24
32 Les fichiers JAR : Packaging d'applications Java 25

1 Objectifs d'apprentissage

À l'issue de cette leçon, vous serez capable de :

- **OOC-BASES-01** — Reconnaître, dans un extrait de code Java, les types primitifs, les règles de nommage et la portée des variables.
- **OOC-BASES-02** — Analyser la distinction objet/référence, le passage par valeur des arguments, l'autoboxing des wrappers et la collecte par le ramasse-miettes.
- **OOC-BASES-03** — Analyser le comportement de la surcharge, des méthodes de classe et des niveaux d'accès à partir d'un extrait de code.

- **OOC-BASES-04** — Décrire les éléments constitutifs d'une classe Java simple (constructeurs, `toString`, `equals`) tels que mobilisés dans le TP compagnon.
- **OOC-BASES-05** — Analyser le comportement de tableaux, de chaînes et des structures de contrôle (`switch`, boucles, `for-each`) sur un extrait de code.
- **OOC-BASES-06** — Évaluer une organisation de programme en paquetages et en fichiers JAR, ainsi que sa modélisation UML et sa documentation Javadoc.

2 Les Types en Java

Java est un langage **fortement typé**. Chaque variable et chaque expression a un type défini à la compilation.

2.1 Types Primitifs

- **Booléen : boolean** (1 bit) : Vrai ou faux
- **Entiers :** byte (1 octet), short (2 octets), int (4 octets), long (8 octets)
- **Flottants :** float (4 octets), double (8 octets)
- **Caractère : char** (2 octets, Unicode)

2.2 Références

- **Pointent vers des objets**
- **Incluent :** classes, interfaces, tableaux

Remarque : * Les types primitifs sont stockés directement dans la variable. * Les références stockent l'adresse mémoire de l'objet.

```
int age = 30; // Type primitif
String nom = "Dupont"; // Référence à un objet de type String
```

3 Les identificateurs en Java

Les identificateurs sont les noms que nous attribuons aux différents éléments de notre code Java (classes, variables, méthodes, etc.).

- **Définition:** Un identificateur est une séquence de caractères utilisée pour nommer un élément dans un programme.
- **Règles de formation:**
  - **Début:** Doit commencer par une lettre (majuscule ou minuscule) ou un underscore (_).
  - **Suite:** Peut contenir n'importe quelle combinaison de lettres, de chiffres et d'underscores.
  - **Caractères:** Privilégier les caractères ASCII pour une meilleure portabilité. Les caractères Unicode sont autorisés.
- **Restrictions:**

- **Mots-clés:** Ne doit pas être un mot-clé réservé du langage (ex: `int`, `class`, `public`).
- **Constantes:** Ne doit pas être identique à `true`, `false` ou `null`.
- **Casse:** Java est sensible à la casse (exemple: `age` et `Age` sont différents).

4 Les Constantes en Java

Une constante est une valeur qui ne peut pas être modifiée une fois initialisée. Elles sont généralement déclarées avec le mot-clé `final`.

- **Types de constantes:**
  - **Numériques:**
    - **Entières:**
      - **long:** Suffixé par `L` (exemple: `long MAX_VALUE = Long.MAX_VALUE;`)
      - **int:** Sans suffixe (exemple: `int MAX_INT = 2147483647;`)
    - **Flottantes:**
      - **double:** Sans suffixe (exemple: `double PI = 3.14159;`)
      - **float:** Suffixé par `f` (exemple: `float EPSILON = 1e-6f;`)
  - **Caractères:** Un caractère Unicode entre simples quotes (exemple: `char SEPARATEUR = ',';`)
  - **Booléennes:** `true` et `false` (exemple: `boolean EST_VALIDE = true;`)
  - **Références:** `null` (indique une absence de référence)

```
final int AGE_MINIMAL = 18;
final double TAUX_TVA = 0.2;
```

5 Les Objets et les Références en Java

5.1 Les Objets

Un objet est une instance d'une classe. Il représente une entité réelle ou conceptuelle avec ses propres attributs (champs) et comportements (méthodes).

5.2 Les Références

Une référence est une valeur qui pointe vers un objet en mémoire. Elle est stockée dans une variable.

- **null:** Une référence peut avoir la valeur `null`, indiquant qu'elle ne pointe vers aucun objet.
- **Opérateurs sur les références:**
  - **Accès aux champs:** `objet.champ`
  - **Appel de méthodes:** `objet.methode()`
  - **Transtypage (cast):** `(Type)objet`
  - **Concaténation de chaînes:** `chaine + objet` (appelle `objet.toString()`)

- **instanceof:** Vérifie si un objet est d'un certain type (`objet instanceof Type`)
- **Comparaison:** `==` compare les références (si elles pointent vers le même objet), `!=` compare l'inverse.
- **Opérateur conditionnel:** `condition ? valeurSiVraie : valeurSi fausse`

6 La Classe Object en Java

La classe `Object` est la classe de base de toutes les classes en Java. Cela signifie que toutes les classes, même si vous ne les en faites pas explicitement hériter, héritent indirectement de `Object`.

6.1 Rôle de la classe Object

- **Classe racine:** Tous les objets en Java sont des instances de `Object` ou de ses sous-classes.
- **Méthodes de base:** Fournit un ensemble de méthodes de base que toutes les classes peuvent utiliser ou redéfinir.

6.2 Méthodes principales de Object

- **getClass():** Retourne un objet de type `Class` représentant la classe de l'objet courant. Cette méthode est utilisée pour l'introspection, c'est-à-dire pour obtenir des informations sur la classe d'un objet à l'exécution.
- **toString():** Retourne une représentation sous forme de chaîne de caractères de l'objet. Par défaut, elle retourne le nom de la classe suivi de son hashcode.
- **equals(Object obj):** Détermine si deux objets sont égaux. Par défaut, elle compare les références (c'est-à-dire si les deux objets sont le même objet).
- **hashCode():** Retourne un code hash, un entier qui sert à placer les objets dans des structures de données comme les `HashMap`.
- **clone():** Crée une copie de l'objet. Cette méthode doit être redéfinie dans les sous-classes pour permettre le clonage.
- **finalize():** Méthode appelée par le garbage collector juste avant de détruire un objet. Elle est rarement utilisée et peut entraîner des comportements imprévisibles.

7 Les Variables en Java (1/2)

En Java, les variables sont utilisées pour stocker des données. Leur portée et leur durée de vie varient en fonction de leur type.

7.1 Les différents types de variables

Type de variable	Portée	Durée de vie	Initialisation
Instance	Toute la classe	Par instance	Déclaration ou constructeur

Type de variable	Portée	Durée de vie	Initialisation
Classe	Toute la classe	Une fois par classe	Lors du chargement de la classe
Locale	Bloc	Pendant le bloc	Déclaration
Tableau	Tableau	Tant que le tableau existe	Lors de la création du tableau
Paramètre de méthode	Méthode	Pendant la méthode	Passage d'arguments
Paramètre de constructeur	Constructeur	Pendant le constructeur	Passage d'arguments
Paramètre d'exception	Bloc catch	Pendant le bloc catch	Capture d'exception

8 Variables finales en Java

- **Immutable après initialisation.**
- **Référence finale:** pointe toujours vers le même objet, mais l'état de l'objet peut changer.
- **Variable d'instance finale:** constante par instance.
- **Initialisation obligatoire:** avant toute utilisation.
- **Valeurs par défaut:** 0 pour les numériques, false pour les booléens, null pour les références.

Pourquoi ? * Fiabilité. * Lisibilité. * Optimisation.

Exemple:

```
final double PI = 3.14159;
```

9 Familles de méthodes en Java

Les méthodes en Java peuvent être classées en différentes familles selon leur rôle et leur visibilité :

- **Accesseurs et modificateurs :** Permettent respectivement de lire et de modifier les attributs d'un objet.
- **Méthodes privées :** Utilisées en interne à une classe, elles ne sont pas accessibles depuis l'extérieur.
- **Méthodes publiques :** Constituent l'interface de la classe, elles sont accessibles depuis n'importe où.

10 Surcharge de méthodes

Définition: La surcharge consiste à définir plusieurs méthodes dans une même classe portant le même nom mais ayant des signatures différentes.

Signature d'une méthode: La signature d'une méthode est composée de : * **Nom de la méthode:** Le nom qu'on utilise pour appeler la méthode. * **Liste des paramètres:** Le nombre et le type des paramètres que la méthode prend en entrée.

Important: Le type de retour ne fait pas partie de la signature. Deux méthodes ne peuvent donc pas être surchargées uniquement en changeant leur type de retour.

11 Exemple

```java
public class Voiture {
    // Instance variables
    private String immatriculation; // Vehicle registration number
    private String marque; // Brand
    private boolean enMarche; // Running status

    // Constructors
    public Voiture(String immatriculation, String marque) {
        this.immatriculation = immatriculation;
        this.marque = marque;
        this.enMarche = false; // Initialize running status to false
    }

    public Voiture(String immatriculation) {
        this(immatriculation, "");
    }

    // Modifiers
    public void setMarque(String marque) {
        this.marque = marque;
    }

    // Accessors
    public String getImmatriculation() {
        return immatriculation;
    }

    public String getMarque() {
        return marque;
    }
```

```
    public boolean isEnMarche() {
        return enMarche;
    }

    // Methods
    public void demarrer() {
        if (!enMarche) {
            enMarche = true;
            System.out.println("La voiture démarre.");
        } else {
            System.out.println("La voiture est déjà en marche.");
        }
    }

    public void arreter() {
        if (enMarche) {
            enMarche = false;
            System.out.println("La voiture s'arrête.");
        } else {
            System.out.println("La voiture est déjà arrêtée.");
        }
    }
}
```

12 Les paramètres en Java : Passage par valeur

- En Java, tous les paramètres sont passés par valeur.** Cela signifie qu'une copie de la valeur de chaque argument est transmise à la méthode appelée.
- Cas particuliers des références
  - **Modification de la référence:** Si on tente de modifier la valeur de la référence elle-même à l'intérieur d'une méthode, cette modification n'aura aucun effet en dehors de cette méthode.
  - **Modification de l'objet référencé:** Si on modifie les propriétés de l'objet référencé par la référence passée en paramètre, ces modifications seront visibles en dehors de la méthode, car on travaille sur l'objet d'origine.

12.1 Exemple

```
class Voiture {
    private String immatriculation;
    private String marque;
```

```
    // ... constructeurs et getters/setters ...
}

public class Parametres {
    public static void main(String[] args) {
        Voiture uneVoiture = new Voiture("1234 AB 83");
        uneVoiture.setMarque("Rolls-Royce");

        // La méthode changeMarque modifie l'objet uneVoiture
        changeMarque(uneVoiture);
        System.out.println(uneVoiture); // Affiche toujours "Rolls-Royce"

        // La méthode abandonner ne modifie pas la référence uneVoiture
        abandonner(uneVoiture);
        System.out.println(uneVoiture); // Affiche toujours l'objet, même si la
référence locale a été mise à null
    }

    private static void abandonner(Voiture v) {
        v = null; // Cette modification n'a pas d'effet sur uneVoiture
    }

    private static void changeMarque(Voiture v) {
        v.setMarque("Fiat"); // Modifie l'objet uneVoiture
    }
}
```

13 Méthodes et variables de classe

- Variables de classe (statiques)
  - **Déclaration:** Utiliser le mot-clé `static` pour déclarer une variable appartenant à la classe elle-même plutôt qu'à une instance particulière.
  - **Constantes de classe:** Lorsqu'une variable de classe est déclarée `final`, elle devient une constante pour l'ensemble du programme.
  - **Initialisation:** Les variables de classe sont initialisées lors du chargement de la classe.
  - **Accès:** Elles peuvent être accédées directement depuis la classe, sans avoir besoin d'instancier un objet.
- Méthodes de classe (statiques)
  - **Invocation:** S'appellent directement à partir de la classe (e.g., `NomDeLaClasse.nomDeLaMethode()`).
  - **Contexte:** Ne peuvent accéder qu'à des variables et méthodes statiques.

- **Signature:** Ne peuvent pas avoir la même signature qu'une méthode d'instance.

13.1 Bloc statique

- **Initialisation complexe:** Utilisé pour effectuer des initialisations complexes de variables de classe, qui ne peuvent pas être faites directement dans la déclaration.
- **Exécution:** S'exécute une seule fois, lors du chargement de la classe.

13.2 Pourquoi utiliser les méthodes et variables de classe ?

- **Partage:** Les variables de classe sont partagées par toutes les instances de la classe.
- **Utilitaires:** Les méthodes de classe peuvent être utilisées sans créer d'instance.
- **Constantes:** Les constantes de classe sont idéales pour des valeurs qui ne changent jamais.

13.3 Exemple 1

```
public class MathUtils {
    public static final double PI = 3.14159;

    static {
        System.out.println("La classe MathUtils est chargée");
    }

    public static int factorielle(int n) {
        // ...
    }
}
```

13.4 Exemple 2

```
public class Chien {
    // Attributs
    private String tatouage; // Tatouage unique du chien
    private String nom; // Nom du chien
    private static int nbChiens = 0; // Nombre total de chiens créés
    private static final int NB_REFUGES = 3; // Nombre de refuges
    private static int[] occupationRefuge = new int[NB_REFUGES]; // Occupation
des refuges (-1: libre, 0: occupé)

    // Constructeur
    public Chien(String tatouage, String nom) {
        this.tatouage = tatouage;
        this.nom = nom;
        nbChiens++; // Incrémente le nombre de chiens à chaque création
```

```
    }

    // Méthodes statiques pour gérer les informations globales
    public static int getNbChiens() {
        return nbChiens;
    }

    // Méthode pour ajouter un chien à un refuge (on incrémente l'occupation)
    public static boolean ajouterChienAuRefuge(int numeroRefuge) {
        if (numeroRefuge < 0 || numeroRefuge ⩾ NB_REFUGES) {
            System.err.println("Numéro de refuge invalide");
            return false;
        } else occupationRefuge[numeroRefuge]++;
        return true;
    }

    // Méthode pour enlever un chien d'un refuge (on décrémente l'occupation)
    public static boolean enleverChienDuRefuge(int numeroRefuge) {
        if (numeroRefuge < 0 || numeroRefuge ⩾ NB_REFUGES) {
            System.err.println("Numéro de refuge invalide");
            return false;
        } else if (occupationRefuge[numeroRefuge] > 0) {
            occupationRefuge[numeroRefuge]--;
            return true;
        } else {
            System.err.println("Le refuge est déjà vide");
            return false;
        }
    }

    // Bloc statique pour initialiser le tableau des refuges
    static {
        for (int i = 0; i < NB_REFUGES; i++) {
            occupationRefuge[i] = -1;
        }
    }
}
```

14 Protection en Java : Niveaux d’accès

- Le principe de base

En Java, la protection d’un élément (méthode ou attribut) définit l’ensemble des entités qui peuvent y accéder. **Cette protection s’applique au niveau de la classe et non de l’objet.**

- Les différents niveaux d'accès

Modificateur	Dans la même classe	Dans le même package	Dans un sous-package	Depuis une autre classe
private	Oui	Non	Non	Non
(aucun)	Oui	Oui	Non	Non
protected	Oui	Oui	Oui	Non (sauf si sous-classe)
public	Oui	Oui	Oui	Oui

14.1 Exemple concret

```
public class Personne {
    /**
     * Le nom de la personne.
     * Cet attribut est privé pour garantir l'encapsulation des données.
     */
    private String nom;

    /**
     * L'âge de la personne.
     * Cet attribut est protégé pour permettre l'accès aux sous-classes.
     */
    protected int age;

    /**
     * L'adresse de la personne.
     * Cet attribut est public pour être accessible de n'importe où.
     */
    public String adresse;

    /**
     * Le numéro de téléphone de la personne.
     * Cet attribut est accessible uniquement depuis le même package.
     */
    String telephone;

    // ... méthodes ...
}
```

15 Structure d'un programme Java

- Un programme Java est un ensemble de **classes**. Chaque classe définit un nouveau type d'objet. Pour qu'un programme soit exécutable, il doit contenir au moins une classe avec une méthode `main`.
- La méthode `main` : le point d'entrée
- **Signature:** `public static void main(String[] args)`
  - `public` : accessible de partout
  - `static` : appartient à la classe, pas à une instance
  - `void` : ne retourne rien
  - `String[] args` : tableau de chaînes de caractères pour les arguments en ligne de commande

15.1 Exemple : un programme simple

```java
public class BonjourATous {
    public static void main(String[] args) {
        System.out.println("Bonjour à tous !");
    }
}
```

16 L'instanciation et les constructeurs

- **L'instanciation**, c'est créer un objet (un instance) à partir d'une classe.
- **Le constructeur** est une méthode spéciale qui :
  - A le même nom que la classe.
  - Est appelée automatiquement lors de la création d'un objet.
  - Sert à initialiser les attributs de l'objet.
  - Ne retourne aucune valeur.
  - Peut être surchargé.
- **Syntaxe d'instanciation:**

```java
Classe nomDeLaVariable = new Classe(paramètres);
```

16.1 Le constructeur par défaut

- **Création automatique:**
  - Si vous ne définissez pas de constructeur, le compilateur en crée un par défaut.
  - Il est sans paramètres.
  - Il a la même visibilité que la classe.
  - Il initialise les attributs à leurs valeurs par défaut (null, 0, etc.).

```java
public class Personne {
    private String nom;
    private int age;
    // Constructeur par défaut créé implicitement
}

Persone personne = new Personne();
```

16.2 Le constructeur paramétré

- **Définition explicite:**
  - Vous pouvez définir un ou plusieurs constructeurs avec des paramètres.
  - Permet d'initialiser les attributs avec des valeurs spécifiques lors de la création de l'objet.
  - Si vous définissez un constructeur, le constructeur par défaut n'est plus créé automatiquement.
  - Vous pouvez toujours définir un constructeur sans paramètres si nécessaire.
  - Vous pouvez également définir plusieurs constructeurs avec des paramètres différents (surcharge).

```java
public class Personne {
    private String nom;
    private int age;

    // Constructeur paramétré
    public Personne(String nom, int age) {
        this.nom = nom;
        this.age = age;
    }

    // Constructeur surchargé
    public Personne(String nom) {
        this(nom, -1); // Appelle le constructeur principal avec age = -1
    }
}
```

17 Les références en Java : un lien vers l'objet

- **Qu'est-ce qu'une référence ?**
  - C'est une variable qui pointe vers un objet en mémoire.
  - Elle ne contient pas l'objet lui-même, mais son adresse.
- **Allocation en mémoire (heap):**
  - Lors de l'instanciation avec `new`, la mémoire est allouée dans le heap.

    - La référence obtenue pointe vers cette zone.
- **Destruction des objets (Garbage Collector):**
    - Pas de destructeur explicite en Java.
    - Les objets sont détruits automatiquement lorsqu'ils ne sont plus référencés.
- **Passage de références en paramètre:**
    - Une copie de la référence est passée.
    - Les deux références pointent vers le **même objet**.
- **Modifications de l'objet:**
    - Toute modification via une référence est visible par les autres.
- **Durée de vie d'un objet:**
    - Tant qu'au moins une référence pointe vers lui.
    - Le garbage collector libère la mémoire lorsqu'il n'y a plus de références.

17.1 Exemple

```
class RamasseMiette {
    public static void main(String[] args) {
        { // Bloc pour limiter la portée de chien1
            Chien chien1 = new Chien("X1", "Rex");
            // chien1 sera collecté à la fin de ce bloc
        }

        // Création d'un grand tableau pour consommer de la mémoire
        int[] grandTableau = new int[1000000]; // Simule une grosse consommation
mémoire

        // Boucle infinie (pour observer le comportement du GC)
        while (true) {
            // Rien à faire ici, juste attendre que le GC s'exécute
        }
    }
}
```

18 Le ramasse-miettes en Java : un gestionnaire de mémoire automatique

- **Qu'est-ce que c'est ?**
    - Mécanisme automatique de libération de la mémoire.
    - Identifie les objets inutilisés et les détruit.
- **Pourquoi l'utiliser ?**
    - **Sécurité:** Évite les fuites mémoire.
    - **Simplicité:** Délégation de la gestion mémoire à la JVM.
- **Comment ça marche ?**

- **Algorithmes:** Mark and Sweep, Copying, Generational.
- **Déclenchement:**
  - Lorsqu'il y a un manque de mémoire.
  - À intervalles réguliers.

- **La méthode `finalize()`**
  - **Invocation:** Juste avant la destruction d'un objet.
  - **Utilité:** Pour effectuer des opérations de nettoyage.
  - **Limitations:**
    - Ne garantit pas la destruction.
    - Appelée une seule fois.
    - À utiliser avec précaution.
- **Forcer le ramasse-miettes (mauvaise idée)** `java Runtime.getRuntime().gc();`

19 Les tableaux en Java : un aperçu

- **Un tableau est un objet**
  - Instancié avec `new`
  - Possède des propriétés (comme `length`)
- **Taille dynamique**
  - Déclarée sans taille précise
  - Taille définie à la création avec `new`
- **Indexation à partir de 0**
  - Premier élément : `tableau[0]`
- **Initialisation**
  - **Avec des valeurs par défaut:** Toutes les valeurs sont initialisées à zéro (pour les types numériques), à `null` (pour les références).
  - **Avec des valeurs explicites:** Lors de la déclaration.
- **Accès à la taille**
  - Propriété `length` : donne le nombre d'éléments du tableau.
- **Instanciation des éléments**
  - **Tableau de primitives:** Les éléments sont des valeurs primitives.
  - **Tableau de références:** Le tableau contient des références vers des objets. Instancier le tableau ne crée pas les objets eux-mêmes.

19.1 Exemple:

```
int[] mesEntiers = new int[3]; // Tableau de 3 entiers
String[] mesChaines = {"Bonjour", "monde"}; // Tableau de 2 chaînes
Voiture[] mesVoitures = new Voiture[3]; // Tableau de 3 références vers des
objets Voiture
```

20 Les chaînes de caractères en Java : un aperçu

- **Les chaînes sont des objets**
  - Instance de la classe `String`
  - Immuables : une fois créée, une chaîne ne peut pas être modifiée
- **Littéraux**
  - Écrits entre guillemets doubles (`"`)
  - Deux littéraux identiques référencent le même objet
- **Opérations de base**
  - **Concaténation:** Opérateur `+`
  - **Comparaison:** Méthodes `equals()` et `compareTo()`
- **Immuabilité**
  - Une chaîne ne peut pas être modifiée après sa création
  - Toute opération de modification crée une nouvelle chaîne
- **Chaînes modifiables**
  - **StringBuffer:** Synchronisée, adaptée à un environnement multi-thread
  - **StringBuilder:** Non synchronisée, plus performante en mono-thread

20.1 Exemple

```
String message = "Hello"; // Création d'une chaîne
message = message + " world"; // Création d'une nouvelle chaîne
StringBuilder sb = new StringBuilder("Hello");
sb.append(" world"); // Modification du StringBuilder
```

21 Enveloppement des types primitifs en Java

- **Pourquoi des classes d'enveloppement ?**
  - **Traiter les types primitifs comme des objets:** Permet d'utiliser des méthodes, de les placer dans des collections, etc.
  - **Utiliser des constantes prédéfinies:** Par exemple, `Integer.MAX_VALUE`.
  - **Effectuer des conversions de types:** Entre types primitifs et leurs équivalents objet.
- **Les classes d'enveloppement**
  - **Byte, Short, Integer, Long:** Pour les nombres entiers
  - **Float, Double:** Pour les nombres à virgule flottante
  - **Boolean:** Pour les booléens
  - **Character:** Pour les caractères
- **Utilisations courantes**
  - **Autoboxing et unboxing:** Conversion automatique entre types primitifs et leurs classes d'enveloppement.
  - **Méthodes statiques:** Pour accéder à des constantes ou effectuer des conversions.

- **Collections:** Les collections en Java ne peuvent contenir que des objets.

21.1 Exemple

```
int age = 30;
Integer ageObj = age; // Autoboxing
int agePrimitive = ageObj; // Unboxing
System.out.println(Integer.MAX_VALUE); // Utilisation d'une constante
```

22 Expressions, Instructions et Blocs en Java

- **Expression**
  - Combinaison de variables, opérateurs et appels de méthode
  - Évalue à une valeur unique
  - Exemples : `x + 5`, `Math.sqrt(9)`, `a == b`
- **Instruction**
  - Unité exécutable
  - Types :
    - Affectation : `x = 5;`
    - Appel de méthode : `System.out.println("Hello");`
    - Déclaration : `int age = 30;`
    - Structure de contrôle : `if`, `for`, `while`, etc.
  - Créée à partir d'une expression en ajoutant un `;`

22.1 Blocs

- **Bloc**
  - Groupe d'instructions délimité par `{}`
  - Utilisations :
    - Corps de méthodes
    - Corps de structures de contrôle
    - Délimiter la portée des variables

22.2 Exemple

```
int x = 10; // Déclaration et affectation
if (x > 5) { // Bloc if
    System.out.println("x est supérieur à 5"); // Instruction dans le bloc
}
```

23 Branchements : Instruction `switch`

- **Fonctionnalité:** Exécute différents blocs de code en fonction de la valeur d'une expression.
- **Types de valeurs:** Entiers, chaînes de caractères, énumérations

- Java 14: Le switch a été amélioré en Java 14, permettant une syntaxe plus concise et des fonctionnalités supplémentaires.

```
switch (expression) {
    case valeur1:
        // Instructions à exécuter si expression == valeur1
        break;
    case valeur2:
        // Instructions à exécuter si expression == valeur2
        break;
    default:
        // Instructions à exécuter si aucune correspondance
}
```

23.1 Exemple

```
// Utilisation avec une énumération
public enum Jour { LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE }
Jour j = Jour.SAMEDI;
switch (j) {
        case LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI:
            System.out.println("Jour de semaine.");
            break;
        case SAMEDI, DIMANCHE:
            System.out.println("Jour de fin de semaine.");
            break;
        default:
            System.out.println("Un nouveau jour ?");
    }
```

```
// Utilisation avec une chaîne de caractères
    String j = "Samedi";
    switch (j) {
        case "Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi":
            System.out.println("Jour de semaine.");
            break;
        case "Samedi", "Dimanche":
            System.out.println("Jour de fin de semaine.");
            break;
        default:
            System.out.println("Un nouveau jour ?");
    }
```

24 Boucles While, Do-While et For en Java

```
Chien[] mesChiens = {new Chien("C1", "Rex"), new Chien("C2", "Medor"), new
Chien("C3", "Pluto")};
int chienCourant = 0;

while (chienCourant < mesChiens.length) {
  System.out.println(mesChiens[chienCourant++]);
}
```

```
for (int i = 0; i < args.length; i++) {
  System.out.print(i == args.length - 1 ? args[i] + "\n" : args[i] + ", ");
}
```

- Le premier exemple (Listing 13) utilise une boucle While pour parcourir un tableau d'objets Chien et en afficher les informations.
- Le second exemple (Listing 14) utilise une boucle For pour itérer sur un tableau d'arguments de la ligne de commande et les afficher.

25 Boucles ForEach en Java

```
Chien[] mesChiens = {new Chien("C1", "Rex"), new Chien("C2", "Medor"), new
Chien("C3", "Pluto")};

for (Chien c : mesChiens) {
  System.out.print(c + " ");
}
```

- La syntaxe `for (Chien c : mesChiens)` permet d'itérer sur chaque élément du tableau et de l'assigner à la variable temporaire `c`.
- Plus concise et lisible que les boucles for traditionnelles pour parcourir des tableaux.
- Améliore la maintenabilité du code.

26 Les paquetages en Java

- **Organisation hiérarchique:**
  - Les classes Java sont regroupées dans des **paquetages** (packages).
  - Cette organisation crée une structure arborescente similaire à un système de fichiers.
- **Déclaration d'un paquetage:**
  - Le mot-clé `package` suivi du nom du paquetage est placé en début de fichier.

- **Exemple:**

```
package mon.paquet;
```

- **Nom complet d'une classe:**
  - Composé du nom du paquetage suivi du nom de la classe, séparés par des points.
  - **Exemple:** `mon.paquet.MaClasse`
- **Utilisation des classes:**
  - **Dans le même paquetage:** On utilise simplement le nom de la classe.
  - **Dans un autre paquetage:**
    - **Import complet:** `import mon.paquet.MaClasse;`
    - **Import de toutes les classes:** `import mon.paquet.*;` (par défaut, `java.lang.*` est importé)
- **Pourquoi utiliser des paquetages ?**
  - **Organisation du code:** Facilite la gestion de projets de grande taille.
  - **Évitement des conflits de noms:** Permet d'avoir des classes avec le même nom dans différents paquetages.
  - **Modularité:** Favorise la réutilisation de code et la création de bibliothèques.

27 La structuration d'un programme : Les paquetages (2/3) - Exemple concret

- **Paquetage `fr.univtln.bruno.java.animaux`:** Contient les classes représentant les différents types d'animaux.
- **Paquetage `fr.univtln.bruno.java.cages`:** Contient les classes représentant les différents types de cages.

27.1 Exemple

```
package fr.univtln.bruno.java.animaux;

public class Chat {
    // ... (constructeurs, méthodes)
    public void miauler() {
        System.out.println("Miaou");
    }
}
```

```
package fr.univtln.bruno.java.cages;
import fr.univtln.bruno.java.animaux.Chat;
public class BoiteAChat {
```

```
    private Chat pensionnaire;

    public void enfermer(Chat c) {
        pensionnaire = c;
    }
}
```

27.2 La structuration d'un programme les paquetages (3/3)

28 Les paquetages standards en Java

- **Nommage des paquetages:**
  - Les noms de paquetages sont généralement des noms de domaine inversés pour éviter les conflits de noms.
  - **Exemple:** `fr.univtln.bruno.samples.Voiture`
  - **Note:** Les tirets (-) ne sont pas autorisés dans les noms d'identificateurs.
- **Paquetages standards de Java:**
  - **java.lang:** Contient les classes de base du langage (String, Integer, etc.).
  - **java.util:** Fournit des utilitaires variés (collections, dates, etc.).
  - **java.io:** Permet de gérer les entrées/sorties (fichiers, flux, etc.).
  - **java.awt** et **javax.swing:** Utilisés pour créer des interfaces graphiques.
  - **java.applet:** Pour développer des applets (moins utilisé aujourd'hui).
  - **java.net:** Pour les opérations réseau (sockets, URL, etc.).

29 Les notations

- **Capitalisation des noms:**
  - **Classes:** Commencent par une majuscule (CamelCase). Exemple : `Chien`, `Voiture`.
  - **Constantes:** Entièrement en majuscules, mots séparés par des underscores. Exemple : `Chien.NB_REFUGES`.
  - **Autres identificateurs:** Commencent par une minuscule, chaque nouveau mot commençant par une majuscule (CamelCase). Exemples : `nbChiens`, `monChien`.
- **Pourquoi ces conventions ?**
  - **Lisibilité:** Facilite la lecture et la compréhension du code.
  - **Cohérence:** Rend le code plus uniforme et professionnel.
  - **Conformité:** Respecte les standards de l'industrie Java.

```
public class Chien {
    public static final int NB_REFUGES = 10; // Constante
    private String nom;
```

```
    public void aboyer() {
        System.out.println("Ouaf !");
    }
}
```

30 La Documentation

- **L'importance de la documentation:**
  - **Pourquoi documenter ?** Pour faciliter la compréhension et la maintenance du code.
  - **Comment ?** En utilisant des commentaires spécifiques (Javadoc) placés juste avant les éléments à documenter (classes, méthodes, attributs).
- **Javadoc : l'outil de documentation standard de Java:**
  - **Fonctionnement:** Javadoc analyse le code source et les commentaires spécifiques pour générer une documentation HTML.
  - **Utilisation:** Il est largement utilisé dans l'industrie, notamment par Oracle pour documenter le JDK.
  - **Lien vers la documentation officielle:** http://www.oracle.com/technetwork/java/javase/documentation/javadoc-137458.html
- **Commentaires Javadoc:**
  - **Syntaxe:** Ils commencent par `/**` et se terminent par `*/`.
  - **Placement:** Ils sont placés juste avant la déclaration de la classe, de la méthode ou de l'attribut à documenter.
  - **Tags:** Des tags spécifiques permettent de structurer la documentation (par exemple, `@author`, `@param`, `@return`).

30.1 Exemple

```
/**
 * Représente un livre.
 *
 * @author VotreNom
 */
public class Livre {
    /**
     * Titre du livre.
     */
    private String titre;

    /**
     * Constructeur de la classe Livre.
     *
```

```
     * @param titre Le titre du livre.
     */
    public Livre(String titre) {
        this.titre = titre;
    }

    /**
     * Retourne le titre du livre.
     *
     * @return Le titre du livre.
     */
    public String getTitre() {
        return titre;
    }
}
```

31 L'organisation (3/3)

- Proposition de structure typique (sans Maven) :

```
monProjet/
├── src/
│   ├── java/
│   │   ├── monPackage/
│   │   │   └── MaClasse.java
│   └── resources/
│       ├── config.properties
│       └── messages_fr.properties
├── bin/
├── lib/
│   ├── externalLib.jar
└── README.md
```

- L'organisation avec Maven : une structure standardisée
- **Avantages:**
  - Structure standardisée et reconnue
  - Gestion automatisée des dépendances
  - Cycle de vie du projet défini (compilation, tests, packaging)
  - Facilité de partage et de collaboration

```
monProjet/
├── pom.xml
```

```
├── src/
│   ├── main/
│   │   ├── java/
│   │   └── resources/
│   └── test/
│       ├── java/
│       └── resources/
└── target/
```

32 Les fichiers JAR : Packaging d'applications Java

- **Un programme Java : un ensemble de classes**
  - Organisées en paquetages pour une meilleure modularité.
- **Le format JAR : une archive**
  - Contient les classes compilées, les ressources (images, etc.) et un manifeste.
  - **Avantages:**
    - Facilite la distribution.
    - Améliore l'organisation.
    - Permet l'exécution directe.
- **Structure d'un JAR**
  - **Classes:** Le code compilé de l'application.
  - **Ressources:** Fichiers non-code utilisés par l'application.
  - **Manifeste (META-INF/MANIFEST.MF):** Contient des métadonnées, notamment le point d'entrée de l'application.
- **Utilisation d'un JAR**
  - **Ajout au classpath:** Indique au JVM où trouver les classes.
  - **Exécution directe:** Si un point d'entrée est spécifié dans le manifeste.
- **Pour aller plus loin:**
  - **Tutoriel officiel:** https://docs.oracle.com/javase/tutorial/deployment/jar/