L3 MIASHS

INF F5 — Objets et classes

UGA UFR SHS Université Grenoble Alpes

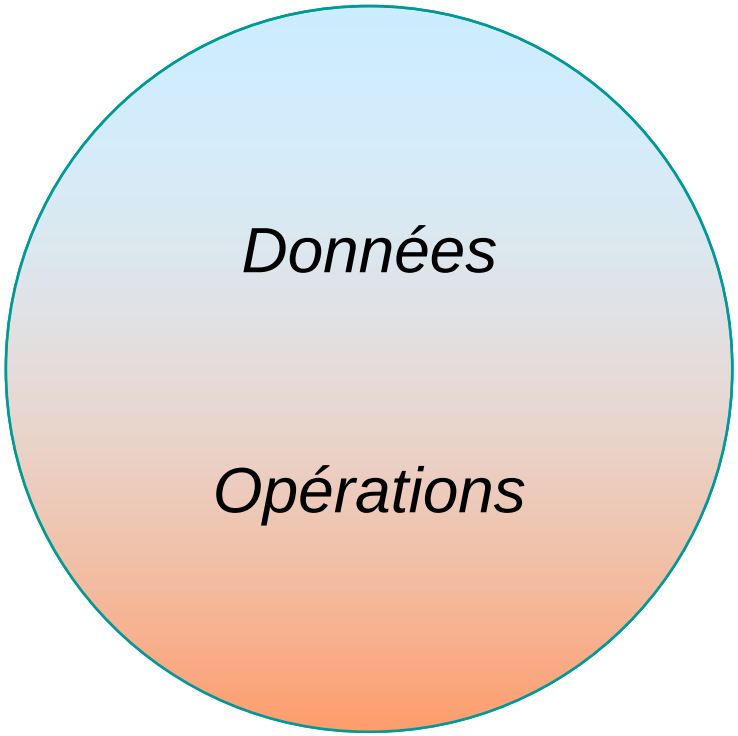
Daniel.Bardou@univ-grenoble-alpes.fr

Sommaire

- Introduction
- Classe
- Attributs et méthodes (d'instance)
- Constructeurs et création d'instances
- Attributs et méthodes de classes
- Exemples de classe : chaînes de caractères

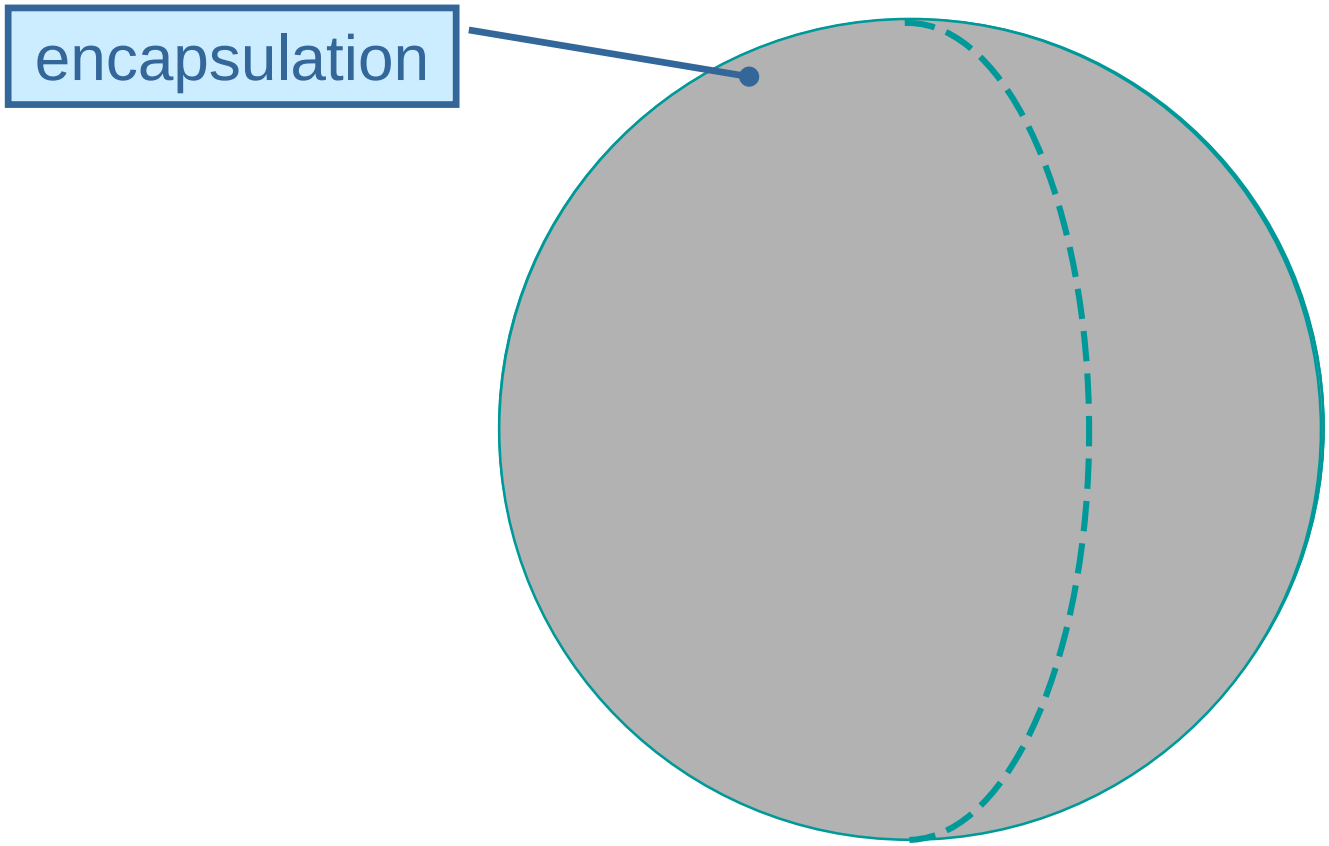
Introduction

Notion d'objet



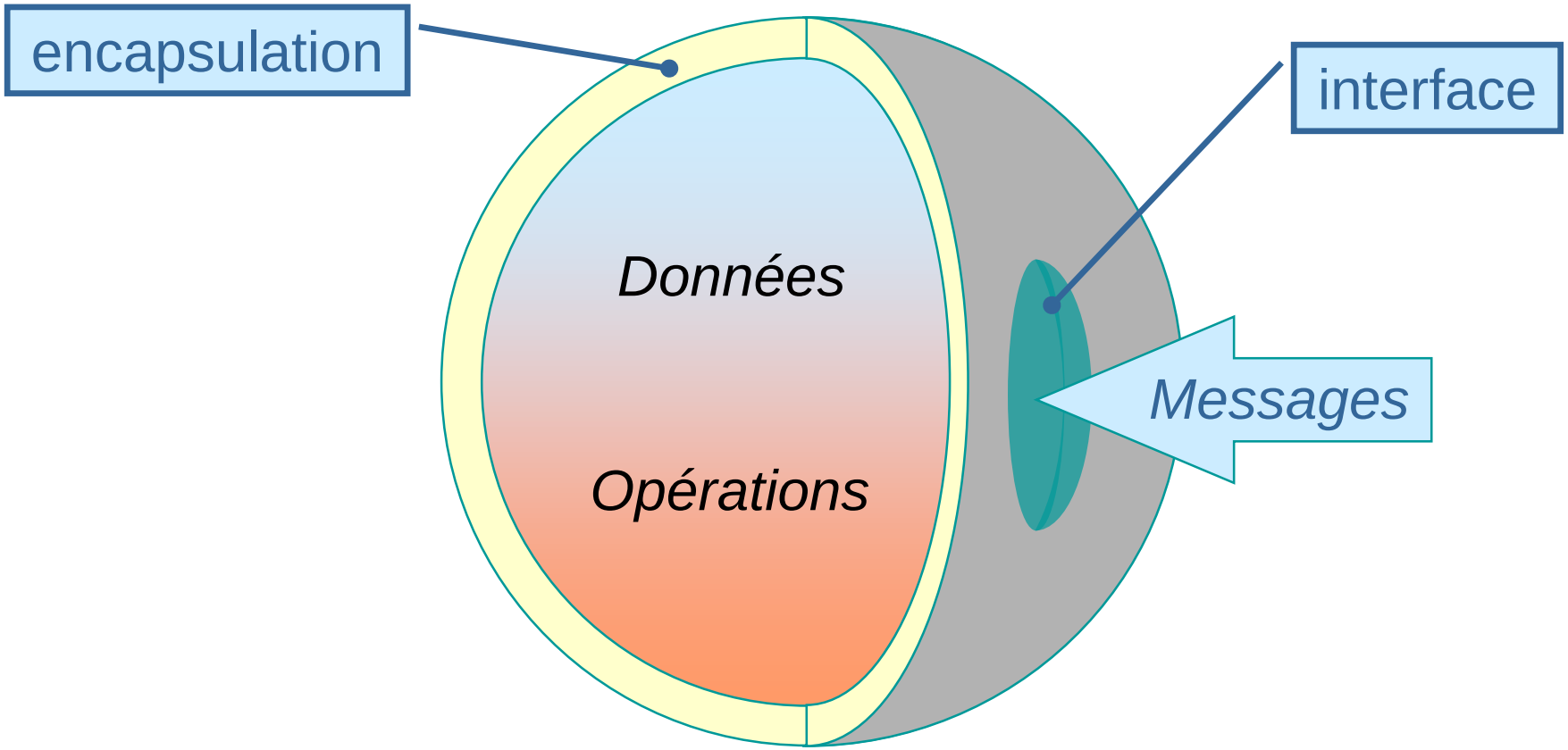
Regrouper les données et les opérations qui les manipulent

Notion d'objet



Se protéger des effets des bords, cerner les responsabilités

Notion d'objet



Proposer un service et réagir aux messages

Notion d'objet

- Un objet est caractérisé par :
  - ses données (valeurs d'attribut),
  - ses comportements (méthodes),
  - son identité (qui le distingue pleinement des autres objets).
- En Java, les objets ont un type de référence. L'identité des objets peut être vérifiée au travers de l'égalité de références.

Caractéristiques de la programmation par objets

- Encapsulation : les attributs et méthodes d'un objet sont définis conjointement dans le même environnement (généralement une classe).
- Accès contrôlé aux informations : niveau de visibilité des définitions.
- Héritage : mécanisme permettant la réutilisation de définitions dans la définition de nouveaux objets.
- Polymorphisme : une méthode peut avoir un comportement différent selon l'objet sur lequel elle est activée.

Classe

Classe

- Une classe est la description d'un objet, en termes d'**attributs** (ses données) et de **méthodes** (ses opérations).
- Cette description peut s'appliquer à plusieurs objets ayant même structure (mêmes attributs) et même comportement (mêmes méthodes).
- Ces objets sont appelés **instances** de la classe et la création d'objets qui se fait en tenant compte de la classe **instanciation**.

Définition d'une classe

- Une classe est définie en utilisant le mot réservé **class**, suivi du nom de la classe et d'un bloc contenant les déclarations/définitions propres à la classe.
- Par convention, un nom de classe commence par une majuscule (exemples : MaClasse, System, Math, String).

Définition d'une classe

- Exemple :

```
public class Point {
    // définitions propres à la classe Point
}
```

- 3 niveaux de visibilité pour une classe :
  - accès public (modificateur **public**),
  - accès paquetage (aucun modificateur),
  - accès privé (modificateur **private**) : la classe n'est visible que depuis le fichier où elle est définie.

Attributs et méthodes (d'instance)

Attributs

- Les attributs d'instance permettent de décrire les valeurs qui seront propres à un objet instance de la classe.
- Une déclaration d'attribut s'apparente à une déclaration de variable dans un programme Java, mais apparaît directement dans le bloc de définition d'une classe.
- Exemple :

```
public class Point{
        double x, y; // deux attributs pour les
    coordonnées
    }
```

Méthodes

- Les méthodes d'instance permettent de décrire les comportements des instances de la classe.
- Une définition de méthode comprend :
  - le type de retour (résultat) de la méthode (ou void si la méthode n'admet pas de résultat),
  - le nom de la méthode,
  - les paramètres de la méthode entre parenthèses, séparés par des virgules (chaque paramètre étant défini par son type et un nom de paramètre formel),
  - le « corps » (les instructions) de la méthode dans un bloc.

Méthodes

- Exemple :

```
public class Point{
        double x, y;

        void afficheX() {
            System.out.println(x);
        }

        double distance(double x2, double y2) {
            double diffX = x2 – x, diffY = y2 – y;
            return Math.sqrt(diffX * diffX + diffY * diffY);
        }
    }
```

Signature de méthode

- La signature d'une méthode est constituée par son nom et la liste dans l'ordre des types de ses paramètres.
- Dans une classe, deux méthodes peuvent avoir le même nom, mais pas la même signature.
- Exemple :

```
public class Point{
    double x, y;

    double distance(double x2, double y2) {...}
    double distance(Point p) {...}
}
```

Instruction return

- L'instruction **return** dans le corps d'une méthode permet d'en arrêter l'exécution et d'en retourner le résultat.
- **return** peut aussi être utilisé dans les méthodes **void** pour simplement en arrêter l'exécution.
- Exemple :

```
public class Point{
        double x, y;

        void afficheXSiPositif() {
            if (x < 0)
                return;
            System.out.println(x);
        }
    }
```

Visibilité

- 4 niveaux de visibilité pour un attribut ou une méthode :
  - accès public (modificateur **public**),
  - accès protégé (modificateur **protected**) : depuis les sous-classes et les classes du même paquetage,
  - accès paquetage (aucun modificateur) : depuis les classes du même paquetage,
  - accès privé (modificateur **private**) : depuis la classe uniquement.

Visibilité

- La bonne pratique (sauf cas particuliers, comme celui de « méthodes outils ») est de déclarer les attributs **private** et les méthodes **public**.
- Exemple :

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

        public void afficheX() {…}
        public double distance(Point p) {…}
    }
```

Accesseurs

- Pour contrôler l'accès aux attributs, il est courant de définir des accesseurs en lecture (*"getters"*) ou en écriture (*"setters"*).
- Exemple :

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

    public double getX() { // getter
        return x;
    }
    public void setX(double val) { // setter
        x = val;
    }
}
```

Opérateur .

- L'opérateur `.` permet d'accéder à certaines propriétés de la partie référencée par une référence. Par exemple, si `t` référence un tableau, `t.length` donne le nombre d'éléments de `t`.
- Pour une référence à un objet, cet opérateur permet d'accéder aux attributs et aux méthodes visibles depuis le programme où l'opérateur est utilisé. Par exemple, si `p` référence un point, `p.x` donne accès à son abscisse et `p.distance(1, 2.0)` active la méthode `distance` sur `p`.
- L'utilisation de l'opérateur `.` sur la référence indéterminée **`null`** lève une exception `NullPointerException`.

Référence this

- Dans le corps d'une méthode, **this** est une référence vers l'instance sur laquelle est activée la méthode.
- Exemples :

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

        public void surLAxeDesX(double x) {
            double y = 0.0;
            this.x = x;
            this.y = y;
        }
        public double distance(Point p) {
            return this.distance(p.x, p.y);
        } // this aurait pu être omis ci-dessus
    }
```

Passage de paramètres

- Les paramètres d'une méthode sont toujours passés « par valeur ».
- **Attention** : la non modification des paramètres effectifs n'est pas garantie pour les paramètres de type référence.
- Exemples :

```
public void m(int x, int [] t, Point p) {
    x = 0; // modification non effective
           // pour le programme appelant
    t[x] = x; // modification d'1 élément du tableau !
    p.setX(x); // modification du point !
}
```

Constructeurs et création d'instances

Constructeurs

- Les constructeurs sont comparables à des méthodes à l'exception des remarques suivantes.
  - Ils n'ont pas de type de retour.
  - Ils ont le même nom que celui de la classe (plusieurs constructeurs si plusieurs signatures).
  - Ils ne peuvent être activés que lors de la création d'une instance.
- Contrairement à ce que semble indiquer leur appellation, ils ne servent pas à construire des instances, mais à **initialiser leurs attributs**.

Constructeurs

- Exemple

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

        public Point() {
            x = y = 0;
        }
        public Point(double x, double y) {
            this.x = x;
            this.y = y ;
        }
    }
```

Constructeurs multiples

- Lorsque plusieurs constructeurs sont donnés dans une classe, il est possible d'en appeler un dans le corps d'un autre grâce à une syntaxe spéciale : **this**(*paramètres*);
- Cet appel n'est permis que si c'est la **première instruction** du constructeur.
- Cette syntaxe permet de décrire le processus (parfois complexe) d'un objet à un seul endroit, tout en laissant à l'utilisateur de la classe de nombreuses possibilités.

Constructeurs multiples

- Exemple

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

        public Point() {
            this(0, 0);
        }
        public Point(double x, double y) {
            this.x = x;
            this.y = y ;
        }
        public Point(Point p) {
            this(p.x, p.y);
        }
    }
```

Constructeur par défaut

- Si **aucune** définition de constructeur n'est donnée dans une classe. Java considère qu'il existe un constructeur sans paramètres qui ne fait aucune initialisation particulière. Par exemple, les 2 définitions de classe ci-dessous sont équivalentes.

```
public class X {
        private int i;



}
```

```
public class X {
        private int i;

        public X() {
        }
}
```

Création d'instances

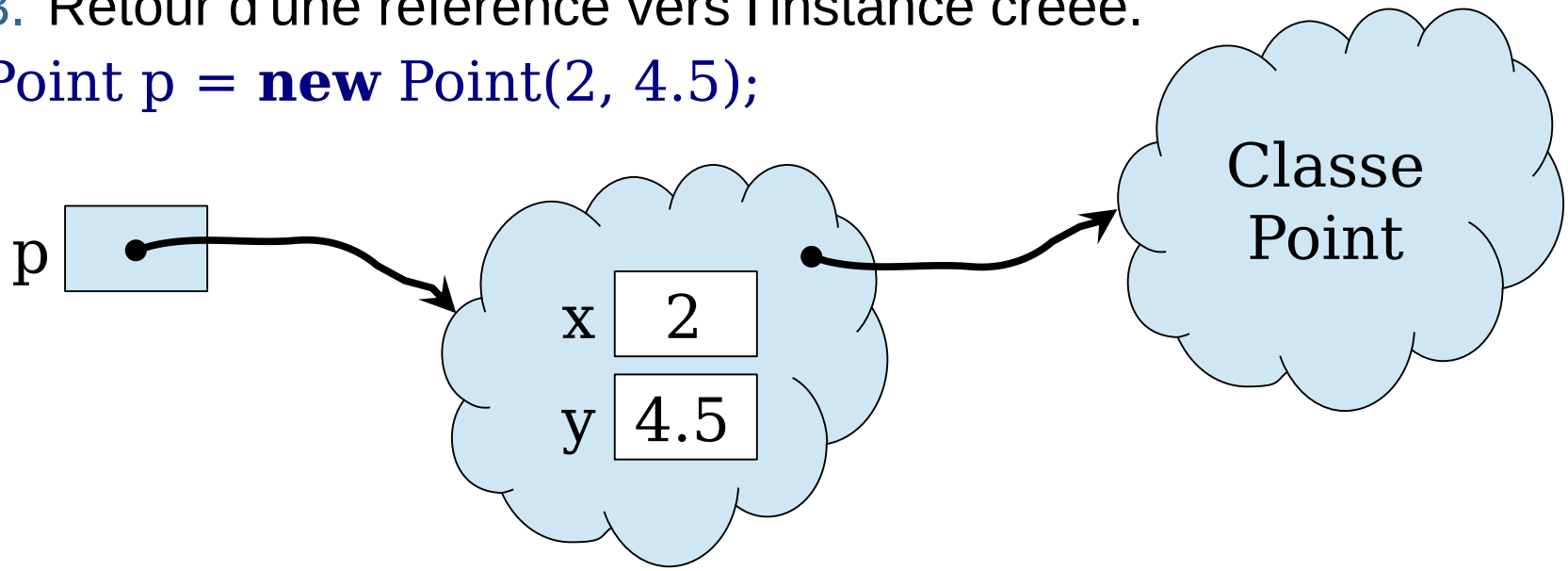
- L'instanciation d'une classe est réalisée grâce à l'usage conjoint de l'opérateur **new** et d'un constructeur de la classe (éventuellement le constructeur par défaut).
- Exemples :

```
Point p1; // déclaration d'une variable référence
   p1 = new Point(2, 4.5); // instanciation
   Point p2 = new Point(); // déclaration et instanciation
   Scanner sc = new Scanner(System.in);
```

3 Étapes pour l'instanciation avec new

1. Réservation de l'espace mémoire nécessaire.
2. Exécution du constructeur (initialisation des attributs) et liaison à la représentation interne de la classe.
3. Retour d'une référence vers l'instance créée.

```
Point p = new Point(2, 4.5);
```



Destruction d'instances ?

- Il n'y a pas de moyens sûrs de provoquer la destruction d'une instance en Java. La mémoire occupée par les objets qui ne sont plus référencés est récupéré par le « ramasse miettes » (*"garbage collector"*) qui est exécuté dans un processus en tâche de fond.
- Il est possible de définir une méthode `finalize()` de type **`void`** qui s'exécute automatiquement sur l'objet lorsque sa mémoire est récupérée.

Attributs et méthodes de classe

Modificateur static

- Le modificateur **static** peut être utilisé dans la déclaration d'un attribut ou la définition d'une méthode : il désigne alors un attribut de classe ou une méthode de classe.
- Exemples :

```
public class Point {
    private static Point origine;

    public static Point getOrigine() {
        return origine;
    }
}
```

Attribut de classe

- Un attribut de classe est directement stocké dans la représentation interne de la classe. Il est accessible dans le contexte de la classe et dans celui des instances (c'est donc une seule valeur partagée par les instances).
- Exemples :
  - La classe `Math` définit les attributs constants de classe `PI` et `E`.
  - La classe `Color` contient des couleurs prédéfinies définies dans des constantes de classe : `WHITE`, `BLACK`, `RED`, `BLUE` ...

Méthode de classe

- Une méthode de classe est une méthode exécutable dans le contexte général de la classe et non celui plus particulier d'une de ses instances.
- En conséquence, on ne peut pas accéder directement à des attributs ou des méthodes d'instance dans une méthode de classe.
- Exemples :
  - La classe `Math` définit de nombreuses méthodes de classe : `sqrt`, `random`, `sin`, `cos`, ...
  - La classe `Integer` définit la méthode de classe `String parseInt(int i)`, ...

Bloc static

- Un bloc **static** est un bloc qui contient des instructions qui sont exécutés lors du chargement d'une classe (de sa première utilisation dans un programme).
- Un bloc **static** peut permettre d'initialiser des attributs de classe lorsque leur initialisation est complexe.
-
```
public class X {
    // ...

    static {
        // instructions exécutées au chargement
    }
}
```

Exemples de classes : chaînes des caractères

Chaînes de caractères en Java

- Il existe plusieurs classes en Java permettant de créer et manipuler des chaînes de caractères.
- Les deux classes principalement utilisées sont :
  - la classe `String` admet des instances immuables,
  - la classe `StringBuffer` admet des instances dont on peut changer les valeurs.
- La classe `String` est particulière en Java (facilités d'écriture).

Classe StringBuffer

- Constructeurs
  - `StringBuffer(String s)`
    permet d'obtenir une chaîne avec les mêmes caractères que `s`.
  - `StringBuffer(int capacite)`
    permet d'obtenir une chaîne vide de capacité `capacite`.
  - `StringBuffer()`
    permet d'obtenir une chaîne vide de capacité `16`.
- La capacité d'une `StringBuffer` correspond au nombre de caractères qu'elle peut contenir sans ré-allocation de mémoire.

Classe StringBuffer

- Méthodes générales
  - **int** length()
    retourne la longueur (nombre de caractères) de la chaîne.
  - **int** capacity()
    retourne la capacité de **this**.

Classe StringBuffer

- Méthodes de concaténation
  - `StringBuffer append(String s)`
    ajoute en fin de chaîne les caractères de `s`. Modifie **`this`** et le retourne en résultat (permet l'activation de méthodes en cascade).
  - `StringBuffer append(int i)`
    ajoute en fin de chaîne `i` converti en chaîne. Modifie **`this`** et le retourne en résultat.
  - Il existe de nombreuses autres signatures pour `append`.

Classe StringBuffer

- Caractère et sous-chaîne
  - **char** charAt(**int** i)
    retourne le caractère d'indice i dans la chaîne.
  - **String** substring(**int** i)
    retourne le suffixe de la chaîne à partir de l'indice i.
  - **String** substring(**int** i, **int** j)
    retourne la sous-chaîne de longueur j – i
    commençant à l'indice i.
  - Ces méthodes sont susceptibles de lever une StringIndexOutOfBoundsException.

Classe StringBuffer

- Méthodes de modification (suppression)
  - StringBuffer delete(**int** i, **int** j)
    supprime les caractères d'indices i à j – 1. Modifie **this** et le retourne.
  - StringBuffer deleteCharAt(**int** i)
    supprime le caractère d'indice i. Modifie **this** et le retourne en résultat.

Classe StringBuffer

- Méthodes de modification (insertion)
  - StringBuffer insert(**int** i, String s)
    insère s à l'indice i. Modifie **this** et le retourne.
  - StringBuffer insert(**int** i, **char** c)
    insère c à l'indice i. Modifie **this** et le retourne en résultat.
  - Il existe de nombreuses autres signatures pour insert.

Classe StringBuffer

- Méthodes de modification (remplacement)
  - StringBuffer replace(**int** i, **int** j, String s)
    remplace les caractères d'indices i (inclus) à j (exclus) par tous ceux de s. Modifie **this** et le retourne.
  - void setCharAt(**int** i, **char** c)
    remplace le caractère d'indice i par c.

Classe StringBuffer

- Conversions
  - `String toString()`
    retourne une `String` contenant les mêmes caractères que **this**.
  - La conversion inverse se fait par le constructeur `StringBuffer(String s)`.