Java – Classes et Objets

Wijin

v1.0 2024

Plan

- Introduction aux classes et objets
- JavaBeans et encapsulation
- Surcharge
- static et this
- final
- Constructeur
- Tableaux
- Enumérations

Java - Introduction aux classes et objets

Java – Introduction aux classes et objets

- Java met en œuvre les **concepts** suivants :
  - Le polymorphisme
  - L'héritage
  - L'encapsulation
  - L'abstraction
  - Les classes
  - Les objets
  - Les méthodes
  - Le passage de paramètre

Java – Introduction aux classes et objets

- Une **classe** est le **modèle** qui va recevoir essentiellement les *attributs* ou *propriétés* de l'objet, et les *méthodes*, ou fonctions

- Dans Java, il est possible de définir **plusieurs classes** dans un fichier suffixé par java

  - Mais, parmi ces classes, une seule sera définie comme publique (« public »)

Java – Introduction aux classes et objets

- Le fait qu'une classe soit déclarée « ***public*** » permet de la rendre accessible **depuis un autre package** que celui dans laquelle elle est déclarée

- Par contre, une classe **non** déclarée « ***public*** » ne sera visible que dans le **package** de la classe principale

Java – Introduction aux classes et objets

```java
package gestion;

public class Client {

    private String nomClient;
    private Integer statut;

    public Boolean creerClient(String nomClient, Integer statut) {
        this.nomClient = nomClient;
        this.statut = statut;
        return true;
    }

    public Boolean changerStatut(Integer statut) {
        if (this.statut == 0) {
            return false;
        }
        this.statut = statut;
        return true;
    }
}

class Adresse {
    public String adresse;
}
```

Java – Introduction aux classes et objets

- On trouve des mots-clés pour la **visibilité** des **attributs**
  - ***public***
  - ***private***
  - ***protected***
- **private** : impossible d'y accéder directement depuis l'extérieur
  - Utilisé généralement pour éviter que les valeurs soient écrasées depuis l'extérieur

Java – Introduction aux classes et objets

```java
public class Client {

    private String nomClient;
    private Integer statut;
```

```java
Client monClient = new Client();
monClient.nomClient = "unNom";
```

Erreur de compilation !!!

Java – Introduction aux classes et objets

- On trouve des mots-clés pour la **visibilité** des **<u>méthodes</u>**
  - ***public***
  - ***private***
  - ***protected***
- **private** : impossible d'y accéder directement depuis l'extérieur
  - Intérêt uniquement dans la classe

Java – Introduction aux classes et objets

```java
public Boolean creerClient(String nomClient, Integer statut) {
    this.nomClient = nomClient;
    this.statut = statut;
    return true;
}
```

```java
Client monClient = new Client();
monClient.creerClient("Mon Client", 1);
```

valide

Java – Introduction aux classes et objets

- **Signature** d'une méthode :
  - *Visibilité*
  - *Type de retour* (primitif, Classe, rien)
  - *Nom*
  - *Paramètre(s)*
    - Type
    - Nom

Java – Introduction aux classes et objets

```java
public Boolean creerClient(String nomClient, Integer statut) {
    this.nomClient = nomClient;
    this.statut = statut;
    return true;
}
```

```java
public void sansRetourEtSansParametre() {
    // Ici, pas besoin de "return" --> "void" en type de retour
}
```

Pas de type de retour !

Java – Introduction aux classes et objets

- **Signature** d'une méthode :
- **Contenu** d'une méthode :
  - Utilisation du mot-clé ***this***
    - Se réfère à **l'instance** de l'objet **en cours**

Java – Introduction aux classes et objets

```
Client monClient = new Client();
monClient.creerClient("Mon Client", 1);

Client deuxiemeClient = new Client();
deuxiemeClient.creerClient("Mon 2ème Client", 1);
```

La valeur de **this** pour chaque objet sera différente

Java – Introduction aux classes et objets

```java
package gestion;

public class Client {
    private String nomClient;
    private Integer statut;
    private Adresse adresse;

    public Boolean creerClient(String nomClient, Integer statut, String adresse) {
        this.nomClient = nomClient;
        this.statut = statut;
        this.adresse = new Adresse();
        this.adresse.setAdresse(adresse);
        return true;
    }

    public String getAdresse() {
        if (this.adresse != null) {
            return this.adresse.getAdresse();
        } else {
            return "Ce client n'a pas encore d'adresse";
        }
    }

    public Boolean changerStatut(Integer statut) {
        if (this.statut == 0) {
            return false;
        }
        this.statut = statut;
        return true;
    }
}
```

Java – Introduction aux classes et objets

```
32
33 class Adresse {
34     private String adresse;
35
36     public String getAdresse() {
37         return adresse;
38     }
39
40     public void setAdresse(String adresse) {
41         if (null != adresse && !"".equals(adresse)) {
42             this.adresse = adresse;
43         }
44     }
45 }
```

Java – Introduction aux classes et objets

```java
package gestion;

public class Gestion {

    public static void main(String[] args) {
        Client monClient = new Client();
        monClient.creerClient("Mon Client", 1, "Rue des tests");
        System.out.println(monClient.getAdresse());
    }

}
```

Création

Appel de creerClient sur l'objet monClient

Java – Introduction aux classes et objets

- Exécution en mode « **debug** » sous Eclipse / Visual Studio Code / IntelliJ
  - Mode **pas à pas**

Java - JavaBeans et encapsulation

Java - Encapsulation

- L'**encapsulation** en Java est un terme qui recouvre une chose simple :
  - **Protéger** les données d'un objet
- Une classe peut contenir des données de type **public** et **private**
  - Si on déclare toutes les propriétés en « **public** », **pas d'encapsulation !**

Java - Encapsulation

```java
package nonencapsule;

public class Client {

    public void crediterCompte(double credit) {
        CompteBancaire cb = new CompteBancaire();
        cb.crediter(100.12);
        cb.valeur = 200;
    }

}

class CompteBancaire {
    public double valeur;

    public double crediter(double credit) {
        this.valeur+=credit;
        return this.valeur;
    }

}
```

Accessible directement !

Non protégée !

Java - Encapsulation

```java
package nonencapsule;

public class Client {
    public void crediterCompte(double credit) {
        CompteBancaire cb = new CompteBancaire();
        cb.crediter(100.12);
    }
}

class CompteBancaire {
    private double valeur;

    public double crediter(double credit) {
        this.valeur+=credit;
        return this.valeur;
    }

    public double getValeur() {
        return this.valeur;
    }

    public void setValeur(double valeur) {
        this.valeur = valeur;
    }
}
```

Protégée !

Accesseur pour accéder à la valeur → **Encapsulation**

Mutateur pour modifier la valeur → **Encapsulation**

Java - JavaBeans

- **JavaBean** : représentation d'une entité **fonctionnelle**
- Exemples :
  - Un client
  - Une commande
  - Un article
  - …
- Représentation **unique**
- Utilisation possible à **divers endroits** de l'application

Java - JavaBeans

- **Acronymes :**

  - POJO (**P**lain **O**ld **J**ava **O**bject)
  - DTO (**D**ata **T**ransfer **O**bject)
  - DAO (**D**ata **A**ccess **O**bject)
  - …

Java - JavaBeans

- **JavaBean** : des conventions à respecter
  - Implémenter l'interface ***Serializable***
  - Proposer un **constructeur sans paramètre**
  - Disposer d'**accesseurs** et **mutateurs publiques** pour les attributs ***private*** (*getters* et *setters*)
  - Classe **non déclarée** « *final* »

Java - JavaBeans

```java
package javabean;

import java.io.Serializable;

public class Client implements Serializable {

    private String nom;

    public Client() {

    }

    public String getNom() {
        return nom;
    }

    public void setNom(String nom) {
        this.nom = nom;
    }

}
```

Java - JavaBeans

- **Exemple** d'utilisation

  - Classe métier ***ClientService*** : réalisation d'opérations sur un client (création, modification, suppression, …)

  - Les méthodes de cette classe vont travailler sur le *JavaBean* **Client**

Java - JavaBeans

- **Exercice**
  - Création d'un JavaBean
  - **Article**
    - Attribut : ***numero***, de type **Integer**
    - Attribut : ***libelle***, de type **String**
    - Getters et Setter
    - Constructeur par défaut

Java - JavaBeans

- **Exercice**
  - Création d'un JavaBean
  - **ArticleService**
    - Méthode ***creerArticle***

Java - JavaBeans

- **Exercice**
  - Création d'un JavaBean
  - **Programme principal**
    - Méthode ***main***
      - Création d'un objet de type *Article*
      - Création d'un objet de type *ArticleService*
      - Appel de la méthode *creerArticle*

Java - Surcharge

Java – Surcharge

- **Surcharge** en Java = capacité d'une classe à accepter d'avoir **plusieurs fois** une méthode avec **le même nom**

  - **Contrainte** : différenciation sur le ***nombre*** et/ou la ***nature*** des types qui sont déclarés pour cette méthode

  - **Attention** : le type de retour ne peut pas servir de discriminant !

Java – Surcharge

```java
package surcharge;

public class Imprimante {

    public void imprimer(String document1, String document2) {
        System.out.println("imprimer1");
        System.out.println(document1 + document2);
    }

    public void imprimer(String... document1) {
        System.out.println("imprimer2");
        for (String doc : document1) {
            System.out.println(doc);
        }
    }

    public void imprimer(String document, Integer nombre) {
        System.out.println("imprimer3");
        for(int indice = 0; indice < nombre; indice++) {
            System.out.println(document);
        }
    }

    public void imprimer() {
        System.out.println("Page de test");
    }
}
```

4 méthodes imprimer !!!

Java – Surcharge

```java
package surcharge;

public class Programme {

    public static void main(String[] args) {
        Imprimante imprimante = new Imprimante();
        imprimante.imprimer();
        imprimante.imprimer("document1");
        imprimante.imprimer("doc1", "doc2");
        imprimante.imprimer("document1", 3);
        imprimante.imprimer("doc5","doc6","doc7");
    }

}
```

Java – Surcharge

- Possibilité d'utiliser le mot-clé ***this*** pour appeler une méthode *surchargée* dans une autre

```java
package surcharge;

public class Imprimante {

    public void imprimer(String document1, String document2) {
        this.imprimer(document1, 10);
        this.imprimer(document2, 10);
    }

    public void imprimer(String document, Integer nombre) {
        for(int indice=0;indice<nombre;indice++) {
            System.out.println(document);
        }
    }

}
```

Java – Surcharge

- Exemple de **surcharge** dans les méthodes de la JDK :
  - Méthode ***valueOf*** de la classe **String**

JAVA - STATIC ET THIS

Java – static et this

- **this** réfère à l'instance *courante* (objet en cours d'exécution)

- **static** lie un élément (attribut ou méthode) à la classe

  - Un élément déclaré **static** est donc **unique** !

Java – static et this

- **static** lie un élément (attribut ou méthode) à la classe
  - Utilité
    - Définir des propriétés de niveau **global**
    - Définir des objets **uniques**

Java – static et this

- **static** lie un élément (attribut ou méthode) à la classe
  - Accès via le nom de la classe (convention)

```java
package staticthis;

public class Voiture {

    private String type;
    private static Integer nombreDeVoitures;

    public void creerVoiture(String type) {
        this.type = type;
        Voiture.nombreDeVoitures++;
    }

    public static Integer getNombreDeVoitures() {
        return Voiture.nombreDeVoitures;
    }

}
```

Java – static et this

- **Synthèse :**
  - **static** est intéressant lorsqu'il n'y a pas besoin d'instancier un objet pour accéder à une fonctionnalité

Java – static et this

- **Exemple avec this et static**

```java
package staticthis;

public class EssaiStatic {

    private String nom;

    public static void main(String[] args) {
        EssaiStatic.afficheStatic("test d'affichage");
        EssaiStatic objet = new EssaiStatic();
        objet.nom = "essai";
        objet.affiche();
    }

    private static void afficheStatic(String valeur) {
        System.out.println(valeur);
    }

    private void affiche() {
        System.out.println(this.nom);
    }

}
```

JAVA - FINAL

Java – final

- **final** = un mot clé **modificateur**
- Utilisable sur une **variable**
  - Après initialisation, sa valeur ne peut plus être changée !
  - Erreur de compilation si tentative de modification !
- Permet de déclarer une **constante**

Java – final

- **final** = un mot clé **modificateur**
- Association avec **static**
  - La variable devient une **constante unique** !

Java – final

```java
package testfinal;

public class Aire {

    public static final double pi = 3.141592653;
    public final double e = 2.718281828;

    public void modification() {
        System.out.println("Valeur de pi : " + pi);
        System.out.println("Valeur de e : " + e);
        Aire.pi = 3.145;
        this.e = 2.71;
    }
}
```

Modifications non autorisées car « final » !

Java – final

- **final** = un mot clé **modificateur**
- Utilisable sur une **méthode**
  - Pour **interdire la redéfinition** de cette méthode dans le cas de l'héritage !

Sera vu plus tard ;)

Java – final

- **final** = un mot clé **modificateur**
- Utilisable sur une **classe**
  - Pour **interdire l'héritage** depuis cette classe !

Sera vu plus tard ;)

Java - Constructeurs

Java – Constructeurs

- Un **constructeur** = une méthode particulière permettant de **créer** un *objet*
  - Notion essentielle !
  - Porte le **nom de la classe** dans laquelle il est décrit
  - N'a **pas de type de retour**
  - Est **appelé automatiquement** en Java lorsqu'on demande la création d'un objet pour la classe de ce constructeur

Java – Constructeurs

- Un **constructeur** = une méthode particulière permettant de **créer** un *objet*
  - C'est la **conséquence** de la demande de création d'un objet (« *new* »)
  - Il n'est pas obligatoire, mais très **utile** !
  - Il **peut être surchargé**, comme une méthode classique
  - Il peut être **appelé depuis un autre constructeur** de façon explicite (avec « *this* »)

Java – Constructeurs

```java
package constructeur;

public class Maison {

    private String materiau;
    private Integer surface;

    public Maison(String materiau, Integer surface) {
        this.materiau = materiau;
        this.surface = surface;
    }

    public Maison(Integer surface) {
        this("brique", surface);
    }

    public Maison(String materiau) {
        this(materiau, 100);
    }

    public Maison() {
        this("brique", 100);
    }

}
```

Pas de type de retour

Pas de « return »

Utilisation de « this » pour appeler un autre constructeur

Java – Constructeurs

- Exemples d'appel

```
1 package constructeur;
2
3 public class Programme {
4
5     public static void main(String[] args) {
6
7         Maison petite = new Maison("Brique", 60);
8
9         Maison moyenne = new Maison("Pierre");
10    }
11 }
```

Java – Constructeurs

- Un **constructeur** = une méthode particulière permettant de **créer** un *objet*
  - On peut également **initialiser** des objets dans un constructeur !

Java – Constructeurs

```java
package constructeur;

public class Maison {
    private String materiau;
    private Integer surface;
    private BlocPorte blocPorte;

    public Maison(String materiau, Integer surface) {
        this.materiau = materiau;
        this.surface = surface;
    }

    public Maison (String materiau, Integer surface, Integer hauteur, Integer largeur) {
        this(materiau, surface);
        this.blocPorte = new BlocPorte(hauteur, largeur);
    }
}

class BlocPorte {
    private Integer hauteur;
    private Integer largeur;

    public BlocPorte(Integer hauteur, Integer largeur) {
        this.hauteur = hauteur;
        this.largeur = largeur;
    }
}
```

Attribut de type BlocPorte

Pour appel du constructeur de BlocPorte

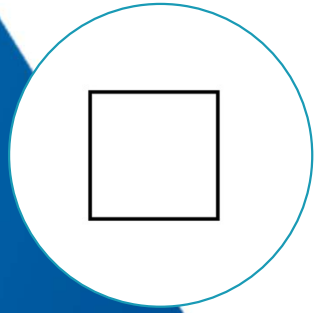
Remarque sur la gestion mémoire : dans l'objet de type ***Maison*** créé, c'est une **référence** vers l'objet de type ***BlocPorte*** qui sera stocké, et non l'objet lui-même

Java – Constructeurs

- Un **constructeur** = une méthode particulière permettant de **créer** un *objet*
  - Il existe un autre **mot-clé** pour les constructeurs : **super**
    - Pour déclencher l'exécution du constructeur de la **classe héritée** pour la **signature demandée**
    - Exemple : **super()** déclenche l'exécution du constructeur sans paramètre de la classe héritée

Est-ce qu'on va finir par voir ce que c'est l'héritage ???!!!

Java - Tableaux



Java – Tableaux

- **Tableau** de base = une **suite** ordonnée ou non d'éléments du **même type**

  - On peut avoir des tableaux de ce qu'on veut (*int*, *String*, *Maison*, …)

  - **Particularité** : déclaration + allocation mémoire

```
private String[] tableau = new String[4];
```

On peut stocker 4 objets de type ***String*** dans ce tableau

Java – Tableaux

- **Tableau** de base = une **suite** ordonnée ou non d'éléments du **même type**

  - Pour **alimenter** un tableau

```
 9●     public void chargement() {
10          System.out.println("chargement...");
11          this.tableau[0] = "chaine1";
12          this.tableau[1] = "chaine2";
13          this.tableau[2] = "chaine3";
14          this.tableau[3] = "chaine4";
15      }
```

Le premier indice commence à **0** !

Java – Tableaux

- **Tableau** de base = une **suite** ordonnée ou non d'éléments du **même type**

  - Pour **lire** un tableau

```
17•    public void afficheTableau() {
18         System.out.println("afficheTableau...");
19         for (int i = 0 ; i < this.tableau.length ; i++) {
20             System.out.println(this.tableau[i]);
21         }
22     }
```

Récupération de la taille du tableau (propriété « ***length*** »)

Accès à l'élément pour l'indice « i »

Java – Tableaux

- **Tableau** de base = une **suite** ordonnée ou non d'éléments du **même type**
  - Pour **lire** un tableau
    - Le nombre d'éléments chargés peut être **différent** de la taille
      - → l'accès à un indice (dans les bornes du tableau) pour lequel il n'y a pas d'objet chargé retourne **null**
    - La tentative d'accès **en dehors des bornes** du tableau lève une **exception** !

Java – Tableaux

- **Tableau** de base = une **suite** ordonnée ou non d'éléments du **même type**

```java
package tableaux;

public class Tableau {

    private String[] tableau = new String[4];

    public void chargement() {
        System.out.println("chargement...");
        this.tableau[0] = "chaine1";
        this.tableau[1] = "chaine2";
    }

    public void accesTableau() {
        System.out.println("accesTableau...");
        System.out.println(this.tableau[3]); // NULL
        System.out.println(this.tableau[4]); // Exception
    }

}
```

Retourne null car il n'y a pas d'élément chargé pour l'indice 3 !

Lève une exception (erreur) car en dehors de bornes du tableau !

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 4 out of bounds for length 4
        at tableaux.Tableau.accesTableau(Tableau.java:16)
        at tableaux.Programme.main(Programme.java:16)
```

Java – Tableaux

- **Tableau** de base = une **suite** ordonnée ou non d'éléments du **même type**

  - Possibilité de **déclarer** et **alimenter** un tableau **sans préciser la taille**

```
 5      private String[] tableau =  {"chaine1", "chaine2", "chaine3"};
 6
 7•     public void afficheTableau() {
 8          System.out.println("afficheTableau...");
 9          for (int i = 0 ; i < this.tableau.length ; i++) {
10              System.out.println(this.tableau[i]);
11          }
12      }
```

Crée directement un tableau de taille 3 !

Java – Tableaux

- **Tableau** de base = une **suite** ordonnée ou non d'éléments du **même type**

  - Il est possible de créer un tableau avec nos propres **types** (classes)

```java
package tableaux;

public class Commande {

    private Integer refCommande;
    private String nomClient;

    public Commande(Integer refCommande, String nomClient) {
        this.refCommande = refCommande;
        this.nomClient = nomClient;
    }

    public void affiche() {
        System.out.println("Commande N° " + this.refCommande + " pour le client " + this.nomClient);
    }
}
```

Classe Commande

Java – Tableaux

- **Tableau** de base = une **suite** ordonnée ou non d'éléments du **même type**

  - Il est possible de créer un tableau avec nos propres **types** (classes)

```java
package tableaux;

public class Tableaux {

    private Commande[] commandes = new Commande[4];

    public void afficheTableau() {
        for (int indice = 0; indice < commandes.length; indice++) {
            commandes[indice].affiche();
        }
    }

    public void chargement() {
        commandes[0] = new Commande(1, "Client 1");
        commandes[1] = new Commande(2, "Client 2");
        commandes[2] = new Commande(3, "Client 3");
        commandes[3] = new Commande(4, "Client 4");
    }

}
```

Classe Tableaux avec utilisation d'un tableau de « Commande »

Utilisation du tableau

Java – Tableaux

- **Tableau** de base = une **suite** ordonnée ou non d'éléments du **même type**

  - Il est possible de créer un tableau avec nos propres **types** (classes)

```
private Commande[] commandes = { new Commande(10, "Client 1"),
                                 new Commande(20, "Client 2"),
                                 new Commande(30, "Client 3"),
                                 new Commande(40, "Client 4")};
```

Autre exemple

Java – Tableaux

- **Tableau** de base = une **suite** ordonnée ou non d'éléments du **même type**
  - Il est possible de créer un tableau à plusieurs **dimensions**
    - Pas forcément très pratique et lisible
    - Exemple de tableau à 2 dimensions

```
private String[][] tableau2d = { { "Oyonnax", "Gex", "Belley", "Bourg en Bresse" },
        { "Laon", "Saint-Quentin", "Château-Thierry", "Hirson", "Guise" } };
```

Java – Tableaux

- **Tableau** de base = une **suite** ordonnée ou non d'éléments du **même type**
  - Exemple de tableau à 2 dimensions

```java
package tableaux;

public class TableauMultiDimension {

    private String[][] tableau2d = { { "Oyonnax", "Gex", "Belley", "Bourg en Bresse" },
            { "Laon", "Saint-Quentin", "Château-Thierry", "Hirson", "Guise" } };

    public void listeTableau() {
        int colonne = 0;
        int ligne = 0;
        for (String[] sousTab : tableau2d) {
            colonne = 0;
            for (String str : sousTab) {
                System.out.println("Valeur de l'élément : " + str);
                System.out.println("Valeur du tableau à l'indice [" + ligne + "][" + colonne + "] est : "
                        + tableau2d[ligne][colonne]);
                colonne++;
            }
            ligne ++;
        }
    }
}
```

Récupération de chaque sous-tableau

Extraction des éléments du sous-tableau courant

Java – Tableaux

- **Tableau** de base = une **suite** ordonnée ou non d'éléments du **même type**

  - Il est possible de passer un tableau en paramètre d'une méthode

```java
package tableaux;

public class TableauPassage {

    private String[] tableau = { "Oyonnax", "Gex", "Belley", "Bourg en Bresse" };

    public void envoiTableau() {
        receptionTableau(this.tableau);
    }

    public void receptionTableau(String[] tableau) {
        for (String valeur : tableau) {
            System.out.println(valeur);
        }
    }
}
```

Java – Tableaux

- **Tableau** de base = une **suite** ordonnée ou non d'éléments du **même type**

  - Il existe des méthodes utilitaires de Java qui **retournent des tableaux** (pas besoin d'initialisation) :

```
String uneChaine = "une chaine de caractères";
String[] sousChaines = uneChaine.split(" ");
for (String valeur : sousChaines) {
    System.out.println(valeur);
}
System.out.println(sousChaines.length);
```

Retourne un tableau de « String » comportant le découpage de la variable « unechaine »

Java - Enumérations

Java – Enumérations

- Une **énumération** = un **ensemble fini** de valeurs
  - Utilisable avec le type ***enum*** en Java
  - **Particularité** : on ne peut utiliser que les valeurs définies dans l'énumération
    - Comme une classe, **mais** :
      - Les objets sont déjà existant
      - On ne peut pas ajouter dynamiquement un nouvel objet

Java – Enumérations

- Une **énumération** = un **ensemble fini** de valeurs

  - Exemple sans énumération

```java
package enumeration;

public class SansEnum {

    public static final int LUNDI = 1;
    public static final int MARDI = 2;

    public void methodeTest (int maDonnee) {
        if (maDonnee == LUNDI ) {
            System.out.println("Traitement 1...");
        } else if (maDonnee == MARDI) {
            System.out.println("Traitement 2...");
        }
    }

}
```

Utilisation de constantes

Java – Enumérations

- Une **énumération** = un **ensemble fini** de valeurs

  - Exemple simple **avec énumération**

```java
package enumeration;

enum Jours {
    LUNDI,
    MARDI;
}

public class TestEnumSimple {
    public static void main(String[] args) {
        TestEnumSimple.test(Jours.MARDI);
    }

    public static void test(Jours quelJour) {
        switch(quelJour) {
            case LUNDI:
                System.out.println("Nous sommes lundi");
                break;
            case MARDI:
                System.out.println("Nous sommes mardi");
                break;
            default :
                System.out.println("Nous sommes un autre jour");
        }
    }
}
```

Utilisation d'une « enum »

Java – Enumérations

- Une **énumération** = un **ensemble fini** de valeurs

  - Autre exemple, plus complexe

```
public enum Atom_const {
    HYDROGEN(.1f),
    CARBON(.28f);

    private float radius;

    private Atom_const(float radius) {
        this.radius = radius;
    }

    public float getRadius() {
        return radius;
    }
}
```

Définition des valeurs de l'énumération

Attribut de l'énumération

Constructeur

Accesseur

Java - Quiz

Java – Quiz

- Quelle est la **conséquence** de l'utilisation du mot clé **static** sur une méthode ?

  - On n'a pas besoin d'instancier la classe (création d'un objet) pour accéder à la méthode
  - On ne peut plus hériter de cette méthode
  - On ne peut plus appeler la méthode à partir d'un objet de la classe concernée
  - La méthode ne peut pas traiter de variables

Java – Quiz

- Quelles sont les règles que doit suivre un **JavaBean** ?
  - Les propriétés ***private*** doivent être accessibles via des accesseurs publics (getter et setter)
  - Il doit y avoir une méthode nommée ***toString()*** pour retourner l'objet sous forme de chaîne de caractères
  - Il doit y avoir un constructeur sans paramètre
  - La classe doit implémenter l'interface ***Serializable***

Java – Quiz

- Que référence **this** ?
  - L'objet courant
  - L'adresse de la méthode courante
  - La classe courante

Java – Quiz

- Comment déclare t-on un tableau de 10 éléments de type String en Java ?
  - private String[10] tableau = new String[10];
  - private String[] tableau = new String[10];
  - private String[] tableau = new String(10);
  - private String[10] tableau = new String(10);

Java – Quiz

- Quelle est la conséquence de l'utilisation du mot clé ***final*** sur une propriété en Java ?

  - Cette propriété devient l'équivalent d'une constante
  - Cette propriété ne peut plus changer de type
  - Cette propriété est partagée par l'ensemble des objets

Java – Quiz

- Comment doit être déclaré le point d'entrée d'une **classe** ?
  - public static void main()
  - public static void main(String args[])
  - public static void main(String args)
  - public void main(String args[])

Java – Quiz

- Qu'est-ce qu'un **constructeur** en Java ?
  - Une méthode qui permet de construire l'objet
  - Une méthode qui est la conséquence de la création d'un objet
  - Une méthode qui permet d'associer les méthodes d'un objet
  - Une méthode qui permet d'écrire une ligne dans une base de données

Java – Quiz

- Comment déclarer une **énumération** nommée « Journee » en Java ?
  - enum Journee {JOUR, NUIT}
  - enum (JOUR, NUIT) as Journee;
  - enum Journee (JOUR, NUIT);
  - enum Journee = JOUR, NUIT;

Java – Quiz

- Qu’est-ce que la **surcharge** en Java ?

- La possibilité de déclarer plusieurs fois la même méthode avec le même nombre d’arguments, mais des types différents

- La possibilité de déclarer plusieurs fois la même méthode avec un nombre d’arguments différent

- La possibilité de déclarer plusieurs fois la même méthode avec des types de retour différents

MERCI POUR VOTRE ATTENTION

Faites-moi part de vos remarques concernant le cours afin qu'il soit amélioré pour les prochaines sessions : nicolas.sanou@wijin.tech