

Bases de la programmation orientée objet

Encapsulation

Denis Poitrenaud

IUT de l'Université Paris Descartes
Denis.Poitrenaud@ParisDescartes.fr

DUT Informatique – 2ème Semestre

Grandes lignes de ce cours

Classes

Encapsulation

Paquetages

Surcharge

Principe

Surcharge et construction

Agrégation et utilisation

Compléments Java

Variables et méthodes de classe

Protocoles standards

Copie d'objets

Classes

Encapsulation

Paquetages

Surcharge

Principe

Surcharge et construction

Agrégation et utilisation

Compléments Java

Variables et méthodes de classe

Protocoles standards

Copie d'objets

Objets

La programmation objet est basée sur l'analogie avec les objets de la vie réelle

- ▶ Un objet met en œuvre des mécanismes complexes pour offrir des fonctionnalités
- ▶ Un objet propose une interface (i.e. un ensemble de commandes) dont le but est de faciliter son usage
- ▶ Un usager emploie uniquement les commandes proposées par l'objet pour accéder aux fonctionnalités
- ▶ Le concepteur de l'objet conçoit et réalise toute une machinerie interne pour que les commandes correspondent aux fonctionnalités attendues



Récepteur radio

Fonctionnalité

- ▶ Écouter la station de son choix

Commandes

- ▶ Allumer/Éteindre
- ▶ Augmenter/Baisser le son
- ▶ Augmenter/Baisser la bande de réception
- ▶ Augmenter/Baisser la fréquence, etc



Mécanismes internes

- ▶ Filtre d'antenne
- ▶ Amplificateur radio fréquence
- ▶ Mélangeur, démodulateur et amplificateur audio
- ▶ Alimentation, etc



Programmation orientée objet

En programmation,

- ▶ Un objet est une donnée sur laquelle peuvent s'appliquer des *méthodes* (i.e. des commandes)
- ▶ Tout objet appartient à une *classe* (i.e. un type)
- ▶ Une classe définit quelles sont les méthodes proposées par les objets de cette classe
- ▶ Une classe définit quels sont les *attributs* (i.e. les données caractéristiques) de chaque objet de cette classe
- ▶ Pour chaque méthode, une classe précise les modifications à apporter aux données caractéristiques lorsque la méthode est appliquée à un objet de cette classe

Le récepteur radio modernisé

Faisons l'hypothèse que le type d'objet **Radio** ait été défini

```
public class Appli {
    public static long TEMPS_ECOUTE = 3600 * 1000;
    public static void main(String [] args) {
        Radio r; // déclaration d'une radio
        r = new Radio(); // construction d'une radio
        // invocations de commandes
        r.allumer();
        r.monterVolume(3); // commande paramétrée
        r.chaineSuivante();

        Thread.sleep(TEMPS_ECOUTE);

        r.eteindre();
    }
}
```

Un objet Java est aussi facile d'emploi qu'un objet de la vie réelle

Méthodes (i.e. commandes)

Les commandes pouvant être appliquées à un objet sont des sous-programmes appelés des **méthodes**

Une méthode est toujours appliquée à un objet (`r.allumer()` indique que la commande `allumer` doit être appliquée à l'objet `r`)

Comme dans la vie réelle, la qualité d'un objet dépend de ses mécanismes internes mais aussi des commandes qui sont proposées

- ▶ Les commandes proposées sont-elles suffisantes ?
- ▶ Les commandes proposées forment-elles un tout cohérent ?
- ▶ Sont-elles faciles d'usage ?
- ▶ Dispose-t-on d'un mode d'emploi ?

Classes (i.e. types d'objet)

Un type d'objet est appelé une **classe**

Concevoir un nouvelle classe consiste à

1. choisir les méthodes qui seront offertes¹
2. définir les attributs (données caractéristiques) des objets
3. programmer, tester et documenter les méthodes

```
public class Radio {  
    // données caractéristiques  
    private boolean enMarche = false ;  
    private int volume = 10 ;  
    private String station = "http://franceinter.com/direct" ;  
    ...  
    // méthodes  
    public void allumer() {  
        enMarche = true ; ...  
    }  
}
```

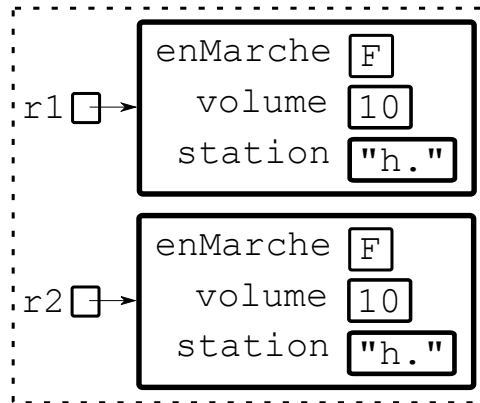
¹**Rappel** : écrire *a priori* un test peut faciliter le choix des méthodes à offrir

Attributs

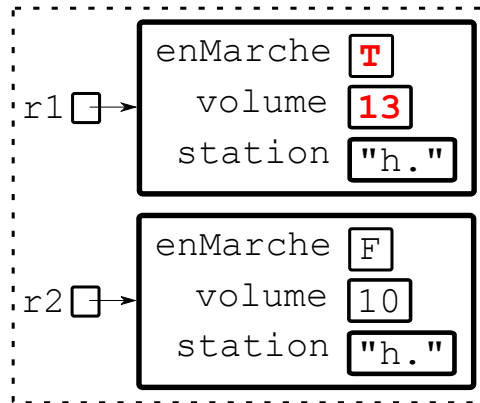
- ▶ Les attributs représentent les données caractéristiques de l'objet
- ▶ Chaque objet de la classe dispose de son propre jeu d'attributs (i.e. on connaît le volume courant de chaque radio)
- ▶ Toutefois, les attributs sont déclarés privés (**private**)
- ▶ Seules les méthodes peuvent y accéder et seules les méthodes publiques peuvent être invoquées par l'utilisateur

Remarque : Une classe peut être vue comme une structure

```
Radio r1 = new Radio ();  
Radio r2 = new Radio ();
```



```
r1.allumer ();  
r1.monterVolume ();
```



Encapsulation

Classes

Encapsulation

Paquetages

Surcharge

Principe

Surcharge et construction

Agrégation et utilisation

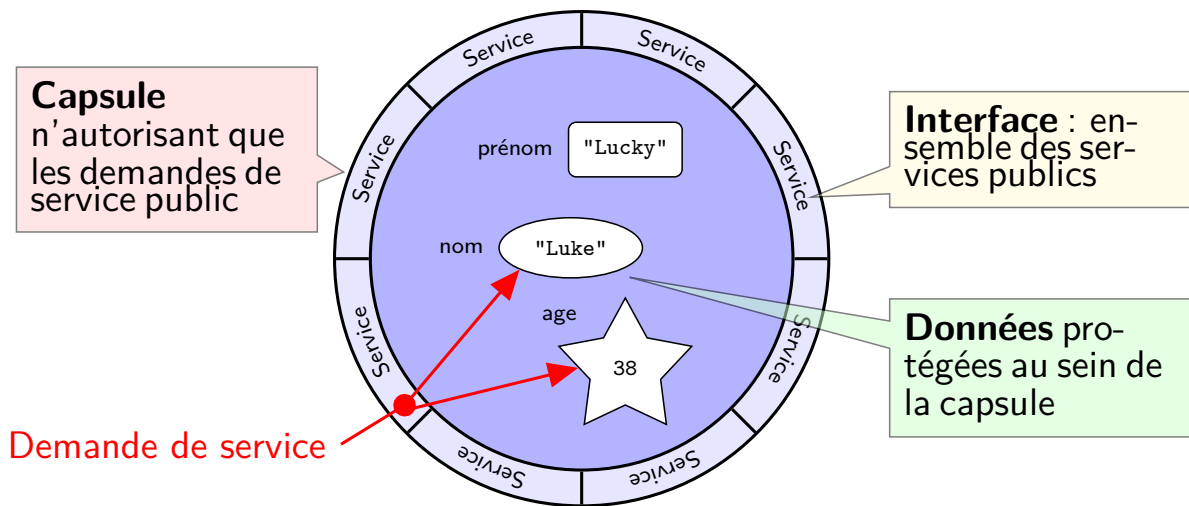
Compléments Java

Variables et méthodes de classe

Protocoles standards

Copie d'objets

Principe POO : Encapsulation et protection des données

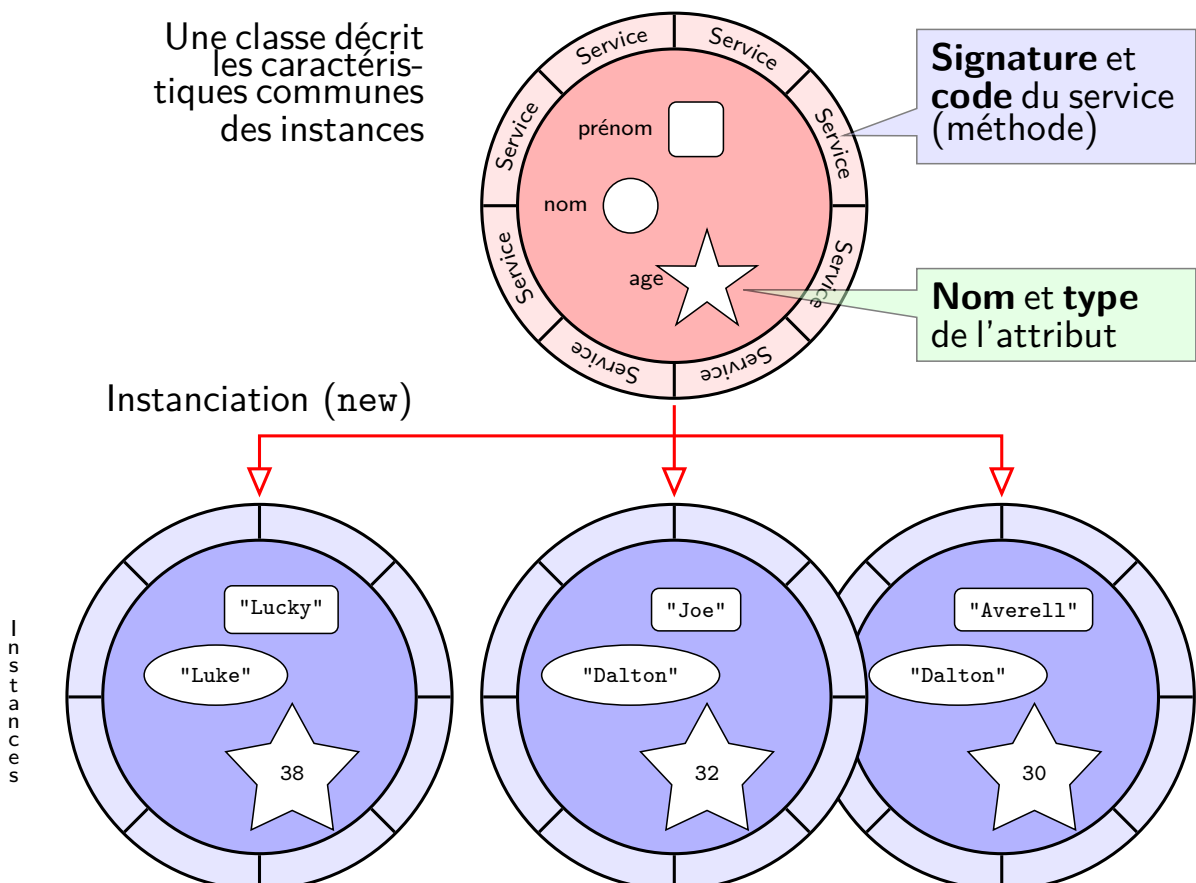


Règle

Les données d'un objet (**son état**) peuvent être lues ou modifiées **uniquement** par les services proposés par l'objet lui-même (**ses méthodes**)

Classe = Modèle d'objets

Une classe décrit les caractéristiques communes des instances



Démarche de conception

La démarche répond aux besoins de maintenance et de réutilisabilité

- ▶ Déterminer en premier lieu les classes d'objet qui seront utiles, le QUI ?
- ▶ Spécifier ensuite les interfaces publiques, le QUOI ?
- ▶ Rejeter le plus possible la réflexion sur la structure interne (implémentation), le COMMENT ?

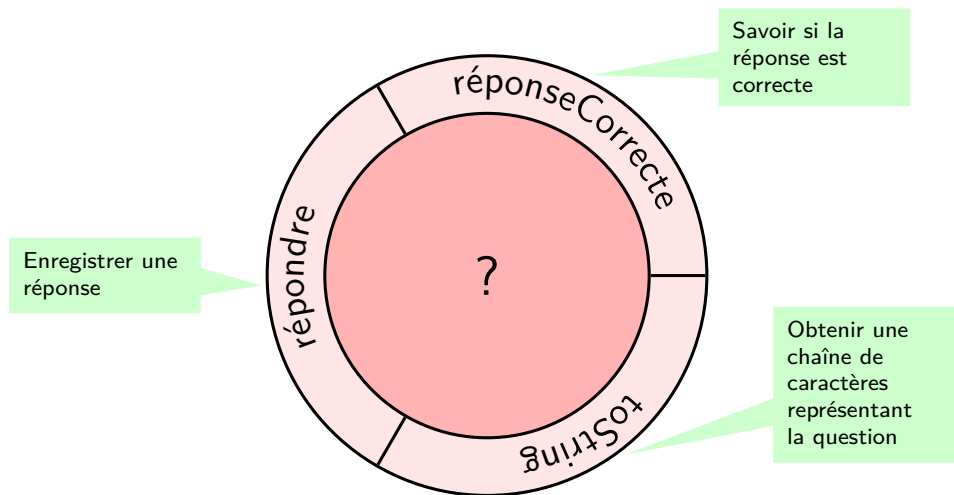
Exemple

On veut produire une application permettant à des étudiants de répondre à des QCM. Comme un QCM est formé d'une suite de questions, on s'intéresse à définir la classe `Question`. Elle doit permettre de

1. Poser une question (l'affichage ou, mieux, la production d'une chaîne de caractères représentant la question sera employée à cet effet)
2. Enregistrer la réponse fournie par un étudiant
3. Déterminer si la réponse fournie est la bonne

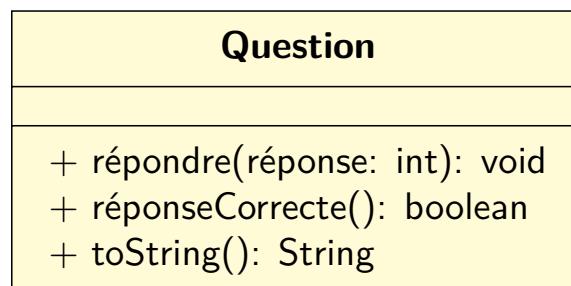
Conception de la classe Question

Les services offerts par la classe :



Aucune hypothèse sur la structure interne

Diagramme UML correspondant



Du point de vue de l'utilisateur de la classe, seuls les services publics (+) sont pertinents

Définition Java du squelette de la classe Question

```
public class Question {  
    ...  
    public void répondre(int réponse) {  
        ...  
    }  
  
    public boolean réponseCorrecte() {  
        ...  
    }  
  
    public String toString() {  
        ...  
    }  
}
```

- ▶ le mot clé **public** permet d'indiquer les services qui sont accessibles à l'utilisateur

Utilisation – Envoi de message

```
public class AppliQuestion {  
    public static void main(String[] args) {  
        ...  
        Question q = ...  
        System.out.println(q.toString());  
        System.out.print("Votre réponse: ");  
        Scanner s = new Scanner(System.in);  
        int i = s.nextInt();  
        q.répondre(i);  
        if (q.réponseCorrecte())  
            System.out.println("Bravo");  
        else  
            System.out.println("Encore raté");  
    }  
}
```

- ▶ `q.répondre(i)` \implies envoi du message `répondre` à l'objet `q`

Implémentation – Choix des données et de leur type

```
public class Question {  
    // intitulé de la question  
    private String question;  
    // tableau des choix possibles  
    private String[] choix;  
    // numéro de la réponse de l'étudiant  
    private int réponse;  
    // numéro de la bonne réponse  
    private int bonneRéponse;  
  
    public void répondre(int réponse) {  
        ...  
    }  
    ...  
}
```

- ▶ le mot clé **private** assure l'encapsulation

Implémentation – Corps des méthodes

```
public class Question {  
    ...  
    public void répondre(int réponse) {  
        this.réponse = réponse;  
    }  
  
    public boolean réponseCorrecte() {  
        return réponse == bonneRéponse;  
    }  
    ...  
}
```

- ▶ **this** référence l'objet pour lequel la méthode est invoquée
- ▶ Il permet d'accéder aux attributs de l'objet (lorsque ceux-ci sont masqués par un paramètre) mais aussi servir à un objet à se passer lui-même en paramètre par exemple

```
public class Question {
    ...
    public String toString() {
        String s = question + "␣:␣";
        int i = 1;
        boolean first = true;
        for (String r : choix) {
            if (!first)
                s = s + ",␣";
            s = s + i + "␣-␣" + r;
            ++i;
            first = false;
        }
        return s;
    }
}
```

Construction

Il est nécessaire de donner le moyen à l'utilisateur de la classe d'initialiser les données de l'objet

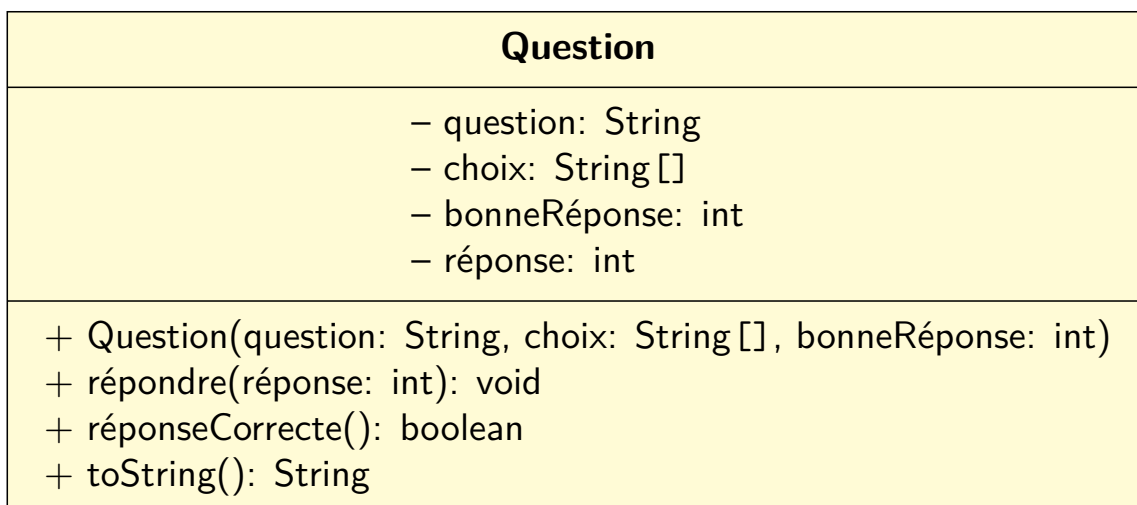
Cela peut être réalisé en ajoutant des méthodes ou (bien mieux) en introduisant un **constructeur**

```
public class Question {
    ...
    public Question (String question ,
                    String [] choix , int bonneRéponse) {
        this.question = question;
        this.choix = choix;
        this.bonneRéponse = bonneRéponse;
        this.réponse = 0;
    }
    ...
}
```

```
public class AppliQuestion {
    public static void main(String[] args) {
        String[] rep = { "Blanc", "Noir" };
        Question q =
            new Question ("Le cheval blanc est", rep, 1);
        System.out.println(q.toString());
        ...
    }
}
```

- ▶ q est une référence vers une instance de Question
- ▶ La valeur particulière **null** peut lui être affectée

Diagramme UML complet



Respect de l'encapsulation

- ▶ Les attributs doivent toujours être déclarés privés
- ▶ Lorsque c'est nécessaire, des méthodes d'accès (accesseur et/ou modificateur – méthode set et/ou get) peuvent être ajoutées

```
public class Question {  
    ...  
    public int getBonneRéponse() {  
        return bonneRéponse;  
    }  
  
    public void setBonneRéponse(int bonneRéponse) {  
        this.bonneRéponse = bonneRéponse;  
    }  
  
    public int getNbChoix() {  
        return choix.length;  
    }  
}
```

Rappel

Une classe implémente un concept associant

- ▶ des données manipulées exclusivement par les instances de la classe
 - ▶ les **attributs** : mode d'accès **private** pour garantir la protection de l'information
- ▶ les traitements que l'on souhaite effectuer
 - ▶ les **méthodes publiques** : mode d'accès **public** visibles depuis l'extérieur de la classe \implies *vision client*
 - ▶ les **méthodes privées** : mode d'accès **private** visibles uniquement depuis l'intérieur de la classe (autres méthodes) \implies *vision fournisseur*

- ▶ **Mauvaise pratique** La création des méthodes publiques ou privées doit se faire en fonction des besoins et non à-priori. On ne définit pas des accesseurs ou (pire) des modificateurs publics systématiquement
- ▶ **Bonne pratique** Si on a besoin d'un accesseur ou d'un modificateur, alors on le crée
 - ▶ si le besoin est à l'intérieur de la classe (fournisseur) \implies accesseur ou modificateur privé
 - ▶ si le besoin est à l'extérieur de la classe (client) \implies accesseur ou modificateur public mais attention, **danger** ! Il faut assurer l'indépendance de la représentation

Paquetages

Classes

Encapsulation

Paquetages

Surcharge

Principe

Surcharge et construction

Agrégation et utilisation

Compléments Java

Variables et méthodes de classe

Protocoles standards

Copie d'objets

Paquetages

- ▶ L'unité de compilation est le fichier
- ▶ Chaque fichier contient une classe (ou une interface ou une énumération)
- ▶ Le nom du fichier doit être le nom de la classe (ou de l'interface ou de l'énumération) terminé par `.java`

- ▶ Des classes sémantiquement liées peuvent être regroupées dans un paquetage
- ▶ Les fichiers contenant les classes doivent être dans un répertoire portant le même nom que le paquetage
- ▶ Les classes n'étant pas dans un paquetage sont dites être dans la paquetage par défaut (anonyme)
- ▶ Il est *fortement* conseillé de répartir les classes d'un même projet au sein de différents paquetages

Les paquetages standards

Les paquetages standards de la version initiale de Java (JDK1.0.2 1995) sont

- ▶ `java.io` : entrees/sorties fichiers
- ▶ `java.net` : programmation reseau
- ▶ `java.awt` : programmation graphique
- ▶ `java.util` : quelques utilitaires
- ▶ `java.lang` : classes du noyau de base

Utiliser un package :

```
java.util.Date monAnniversaire = new java.util.Date();  
import java.util.Date;  
import java.net.*;
```

Le package `java.lang` est importé implicitement (c'est le seul)

Organisation des fichiers sources

Déclaration d'une classe au sein d'un paquetage

```
package qcm;  
  
public class Question {  
    ...  
}
```

Utilisation à partir d'un autre paquetage

```
import qcm.Question;  
  
public class AppliQuestion {  
    public static void main(String[] args) {  
        String[] rep = { "Blanc", "Noir" };  
        Question q = new Question("Le cheval est", rep, 1);  
        ...  
    }  
}
```

Visibilité

Le modificateur qui précède chaque méthode (et attribut) détermine sa visibilité :

Modificateur	Visibilité
private	uniquement au sein la classe
pas de modificateur	uniquement au sein du même paquetage
protected	<i>lié à l'héritage</i>
public	au sein de toute classe

Surcharge

Classes

Encapsulation

Paquetages

Surcharge

Principe

Surcharge et construction

Agrégation et utilisation

Compléments Java

Variables et méthodes de classe

Protocoles standards

Copie d'objets

Principe

Classes

Encapsulation

Paquetages

Surcharge

Principe

Surcharge et construction

Agrégation et utilisation

Compléments Java

Variables et méthodes de classe

Protocoles standards

Copie d'objets

Distinction des signatures de méthodes

- ▶ Les signatures des méthodes doivent être différentes ou ces méthodes doivent appartenir à des classes différentes
- ▶ Les signatures de deux méthodes d'une même classe diffèrent si
 - ▶ Les méthodes n'ont pas le même nom
 - ▶ Les méthodes ont le même nom mais n'ont pas le même nombre de paramètres
 - ▶ Les méthodes ont le même nom et le même nombre de paramètres mais n'ont pas les mêmes types de paramètres
- ▶ **Attention** : Le type de la valeur de retour n'intervient pas pour distinguer deux méthodes :
 - ▶ **void** répondre(**int** réponse) {...}
 - ▶ **boolean** répondre(**int** réponse) {...} // *même signature*

Cas particulier : surcharge

- ▶ Surcharge : deux méthodes de mêmes noms et
 - ▶ soit de classes différentes
 - ▶ soit ayant des signatures différentes
- ▶ Exemple: deux méthodes d'une même classe ont le même nom mais le nombre de leurs paramètres diffèrent :
 - ▶ **void** répondre(**int** réponse) {...}
 - ▶ **void** répondre() { /* *répondre au hasard* */ }
- ▶ Pour le programmeur, ces méthodes ont l'air proches mais pour le compilateur, elles sont radicalement différentes, exactement comme si elles n'avaient pas le même nom.
- ▶ Ce n'est pas un principe objet mais plutôt une caractéristique de certains langages typés (Java, C++, CAML etc.).

Contrôle des types

- ▶ Java est un langage fortement typé (le contrôle des types est strict)
- ▶ Le compilateur détermine la méthode invoquée en étudiant le type de l'instance de l'appel ainsi que le nombre et le type des paramètres
- ▶ Exemple :

```
System.out.print (3);  $\implies$  java.io.PrintStream.print (int i)
System.out.print ("ok");  $\implies$  java.io.PrintStream.print (String s)

Question q = new Question(...);
System.out.print (q);  $\implies$  ...
                        java.io.PrintStream.print (Object o) ... mystère
```

Surcharge et construction

Classes

Encapsulation

Paquetages

Surcharge

Principe

Surcharge et construction

Agrégation et utilisation

Compléments Java

Variables et méthodes de classe

Protocoles standards

Copie d'objets

Cas d'utilisation courante de la surcharge – les constructeurs

Il est souvent nécessaire de donner plusieurs façons à l'utilisateur de construire des instances d'une même classe

Exemple (un court extrait de la documentation) :

- ▶ `String()`
Initializes a newly created String object so that it represents an empty character sequence
- ▶ `String(char[] value)`
Allocates a new String so that it represents the sequence of characters currently contained in the character array argument
- ▶ `String(String original)`
Initializes a newly created String object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string

Rappel du constructeur de la classe Question

```
public class Question {
    private String question;
    private String[] choix;
    private int bonneRéponse, réponse;
    private static final String[] choixDéfaut
        = {"vrai", "faux"};

    public Question(String question,
        String[] choix, int bonneRéponse) {
        assert (choix.length >= 2);
        assert (1 <= bonneRéponse);
        assert (bonneRéponse <= choix.length);
        this.question = question;
        this.choix = choix;
        this.bonneRéponse = bonneRéponse;
        this.réponse = 0;
    }
    ...
}
```

Ajout d'un nouveau constructeur

```
public class Question {
    ...
    private static final String [] choixDéfaut
        = {"vrai", "faux"};

    public Question(String question ,
        String [] choix, int bonneRéponse) {...}

    public Question (String question , int bonneRéponse) {
        assert (1 <= bonneRéponse);
        assert (bonneRéponse <= choixDéfaut .length);
        this .question = question;
        this .choix = choixDéfaut ;
        this .bonneRéponse = bonneRéponse;
        this .réponse = 0;
    }
    ...
}
```

Ajout d'un nouveau constructeur – Correction

Le constructeur précédent a été mal programmé car il faut réemployer le plus possible le code existant

```
public class Question {
    ...
    private static final String [] choixDéfaut
        = {"vrai", "faux"};

    public Question(String question ,
        String [] choix, int bonneRéponse) {...}

    public Question (String question , int bonneRéponse) {
        // appel au constructeur ci-dessus
        this (question , choixDéfaut , bonneRéponse);
        // Attention, cela doit être la première
        // instruction du constructeur
    }
    ...
}
```

```
// avec le constructeur original
String[] rep = { "Blanc", "Noir" };
Question q =
    new Question("Le cheval blanc est", rep, 1);
System.out.println(q);

// avec le nouveau constructeur
q = new Question("Vous aimez programmer?", 1);
System.out.println(q);
```

Agrégation et utilisation

Classes

Encapsulation

Paquetages

Surcharge

Principe

Surcharge et construction

Agrégation et utilisation

Compléments Java

Variables et méthodes de classe

Protocoles standards

Copie d'objets

Principe POO : Agrégation

Attributs = variables (visibilité au sein de la classe)

- ▶ types de base : **boolean**, **int**, **long**, **short**, **float**, **double**, **char**, **byte**
- ▶ types objet prédéfinis : `String`, `ArrayList<Integer>`, etc.
- ▶ types objet utilisateurs : `Question`, etc.

Agrégation (ou composition) = attributs objets

- ▶ Relation "**A-UN**" entre deux classes ("**A-DEUX**", "**A-DES**", "**EST-COMPOSÉ-DE**", etc.)
- ▶ Exemples :
 - ▶ Voiture "**A-UN**" Moteur
 - ▶ QCM "**A-DES**" Question

Agrégation : exemple

```
package qcm;
import java.util.ArrayList;

public class QCM {
    private ArrayList<Question> questions =
        new ArrayList<Question>();

    public void ajouter(Question q) {
        questions.add(q);
    }

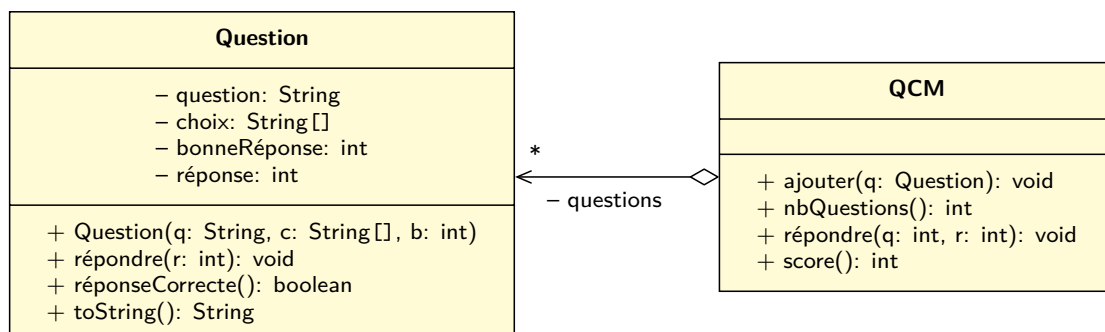
    public int nbQuestions() {
        return questions.size();
    }

    ...
}
```

Agrégation : exemple

```
public class QCM {  
    ...  
  
    public void répondre(int q, int r) {  
        questions.get(q).répondre(r);  
    }  
  
    public void afficher(int q) {  
        System.out.println(questions.get(q).toString());  
    }  
  
    public int score() {  
        int s = 0;  
        for (Question q : questions)  
            if (q.réponseCorrecte())  
                ++s;  
        return s;  
    }  
}
```

Diagramme UML - Agrégation



Agrégation : bonnes et mauvaises pratiques

Mauvaise pratique

- ▶ Ce n'est pas parce que des attributs sont présents dans une classe qui s'agit d'une agrégation

Bonne pratique

- ▶ Il y a agrégation de B "dans" A **si et seulement si** le test :

A "A-UN" B

est valide

Remarque

Cette règle doit être appliquée à tous les attributs quelque soit leur type (objet ou non)

Principe POO : Utilisation

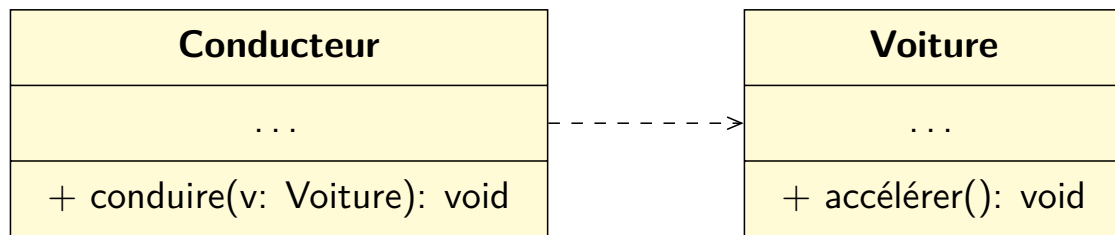
Utiliser des objets

- ▶ Créer des instances avec **new**
- ▶ Paramètres de méthodes
- ▶ Retours de méthodes

Lien d'utilisation

- ▶ Relation "UTILISE" entre deux classes ("EST-ASSOCIÉ-À", etc.)
- ▶ Exemples :
 - ▶ Conducteur "UTILISE" Voiture

Diagramme UML - Utilisation



Remarque

- ▶ agrégation = cas particulier (et plus précis) d'utilisation
- ▶ ajout de commentaire sur la flèche si nécessaire (ex. "conduit")

Compléments Java

Classes

Encapsulation

Paquetages

Surcharge

Principe

Surcharge et construction

Agrégation et utilisation

Compléments Java

Variables et méthodes de classe

Protocoles standards

Copie d'objets

Variables et méthodes de classe

Classes

Encapsulation

Paquetages

Surcharge

Principe

Surcharge et construction

Agrégation et utilisation

Compléments Java

Variables et méthodes de classe

Protocoles standards

Copie d'objets

Variable de classe

On veut introduire des choix par défaut

```
public class Question {  
    ...  
    private String [] choixDéfaut = {"vrai", "faux"};  
    ...  
}
```

Problème : chaque instance dispose de son propre tableau choixDéfaut alors qu'il devrait être commun à tous les objets

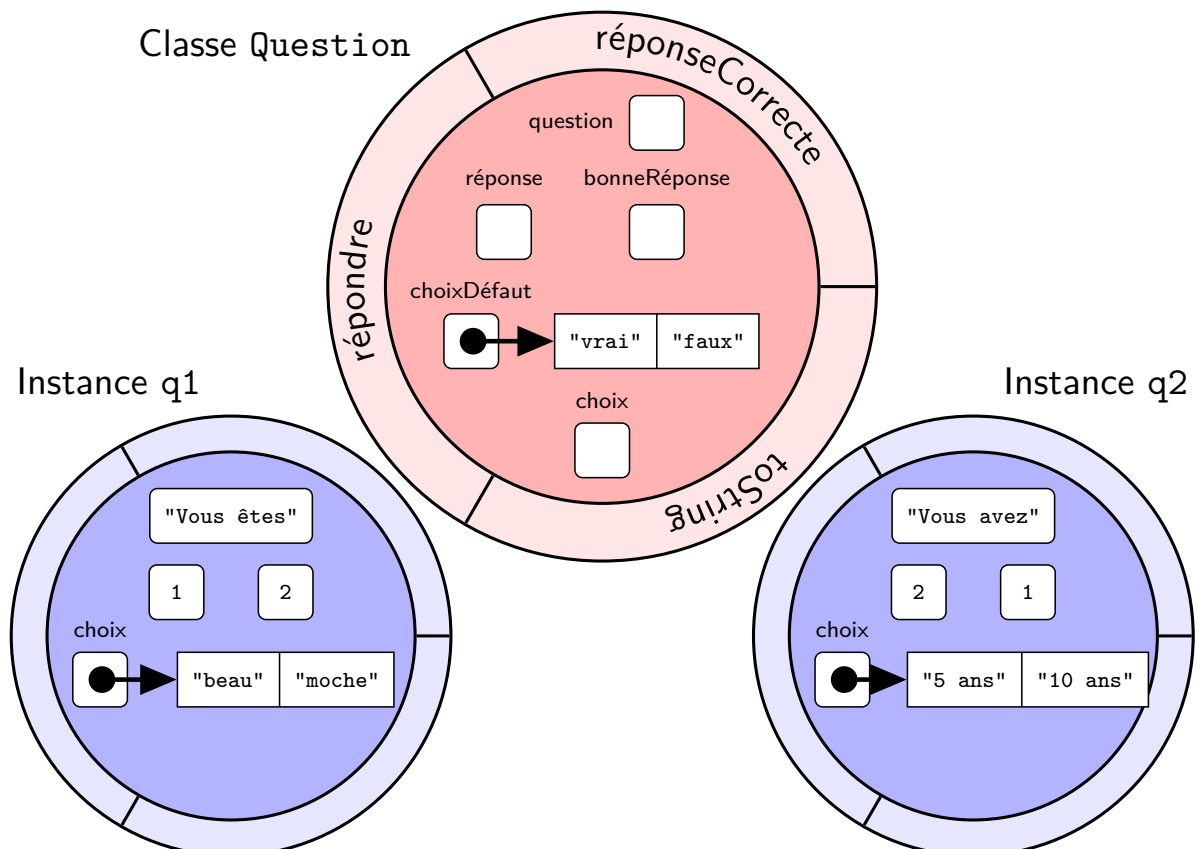
```
public class Question {  
    ...  
    private static String [] choixDéfaut  
        = {"vrai", "faux"};  
    ...  
}
```

Variable de classe et constante

On peut rendre ce tableau constant (non modifiable)

```
public class Question {  
    ...  
    private static final String [] choixDéfaut  
        = {"vrai", "faux"};  
    ...  
}
```

Variable de classe



Méthode de classe

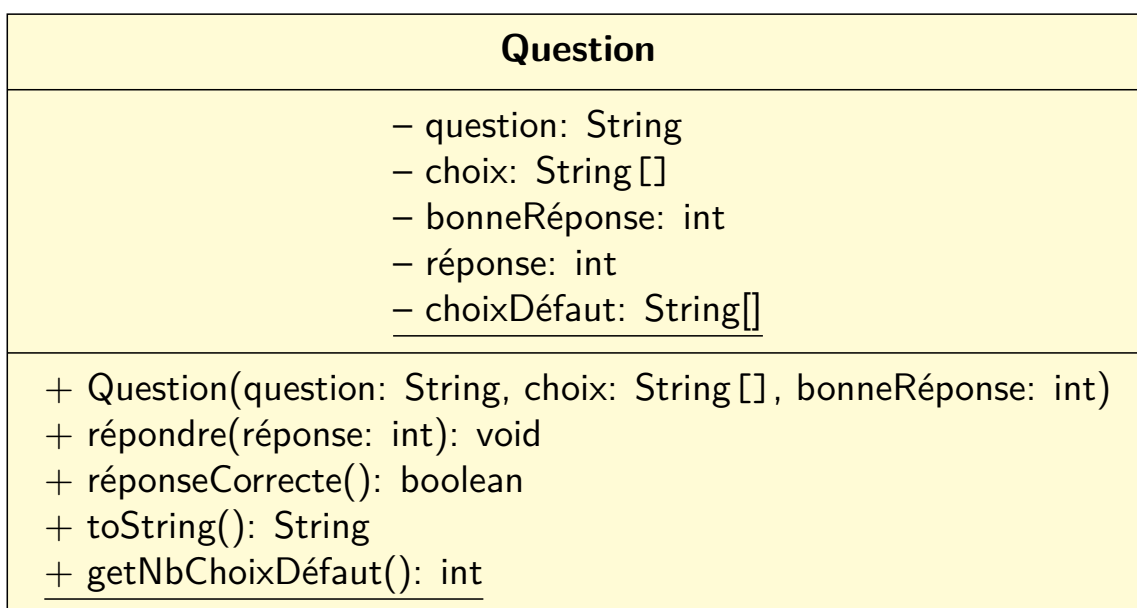
On veut pouvoir connaître le nombre de choix par défaut. Cette méthode n'accède à aucune donnée d'une instance. Elle peut donc être déclarée statique

```
public class Question {  
    ...  
    public static int getNbChoixDéfaut() {  
        return choixDéfaut.length;  
    }  
    ...  
}
```

Utilisation : `System.out.println (Question.getNbChoixDéfaut());`

Un autre exemple classique est la méthode `main` du protocole de lancement d'une application

Diagramme UML avec méthodes et variables de classe



Protocoles standards

Classes

Encapsulation

Paquetages

Surcharge

Principe

Surcharge et construction

Agrégation et utilisation

Compléments Java

Variables et méthodes de classe

Protocoles standards

Copie d'objets

Protocoles standards

Des méthodes standards peuvent être invoquées sur tous les objets (i.e. toutes les instances de classe)

- ▶ Représentation textuelle : `String toString()`
- ▶ Égalité structurelle : **boolean** `equals(Object)`
- ▶ Finalisation : **void** `finalize()`
- ▶ Clonage : `Object clone()` (cf. cours sur la copie d'objet)
- ▶ Hachage : **int** `hashCode()` (cf. les collections)
- ▶ + méthodes spécifiques au multi-threading

Méthodes prédéfinies ayant une implémentation par défaut dans `java.lang.Object`

Finalisation

Méthode : **public void** finalize ()

- ▶ Invocation automatique avant la destruction de l'objet
- ▶ Implémentation par défaut: ne fait rien
- ▶ Son rôle est de libérer les ressources (fichiers ouverts, socket, etc.) occupées par l'objet
- ▶ Très rarement "redéfinie" dans les classes (exemple dans la deuxième partie du module)
- ▶ Mais on peut, par exemple, afficher quelque chose pour comprendre la gestion mémoire de Java

Exemple

```
public void finalize () {  
    System.out.println ("l'objet_" +  
                        this + "_est_désalloué");  
}
```

Copie d'objets

Classes

Encapsulation

Paquetages

Surcharge

Principe

Surcharge et construction

Agrégation et utilisation

Compléments Java

Variables et méthodes de classe

Protocoles standards

Copie d'objets

Affectation d'objets

L'opérateur d'affectation (=) appliqué à des objets réalise toujours une copie de référence

```
public class AppliQuestion {
    public static void main(String[] args) {
        String[] rep = { "Blanc", "Noir" };
        Question q1 =
            new Question("Le cheval blanc est", rep, 1);
        Question q2 = q1;
        q1.setBonneRéponse(2);
        System.out.println(q2.getBonneRéponse());
    }
}
```

q1 et q2 désignent la même question

L'appel q1.setBonneRéponse(2) a un effet de bord sur q2

Affectation de tableaux (rappel)

Les tableaux fonctionnent sur le même schéma

Déclaration d'un tableau (ou plus exactement d'une référence vers un tableau) :

```
Question[] tab; // tab est initialisé à null
```

Instanciation d'un tableau :

```
tab = new Question[10];
// les 10 références sont initialisées à null
```

Cette instance restera en mémoire tant qu'au moins une référence la désignera. Si vous voulez la rendre "libérable" au plus tôt :

```
tab = null;
```

Affectation de tableaux (rappel)

L'opérateur d'affectation (=) appliqué à des tableaux réalise toujours une copie de référence

La méthode `clone` peut être employée pour dupliquer un tableau. Toutefois, cette méthode réalise une copie superficielle.

```
import java.util.Arrays;
public class ArrayCopyDemo {
    public static void main(String [] args) {
        int [] org1 = { 1, 2, 3, 4, 5 };
        int [] copie1 = org1.clone();
        System.out.println(Arrays.toString(copie1));

        int [][] org2 = { {1, 2}, {3, 4}};
        int [][] copie2 = org2.clone();
        org2[0][0] = 10;
        System.out.println(copie2[0][0]); // affiche 10
    }
}
```