3I002 : Programmation par objet
Licence de Sciences et Technologies, Sorbonne Université
Année 2018–2019

# Partiel : Micro-OS

**11 mars 2019**

**durée : 2h30**

## Déroulement de l'épreuve

- L'examen est individuel et se déroule sur machine.
- Pour chaque question, vous devez fournir comme réponse des sources Java qui compilent et donnent les résultats demandés.
- À la fin de l'épreuve, sauvegardez vos fichiers, quittez les applications (Eclipse, etc.) et fermez votre session. Les fichiers Java laissés sur votre compte seront automatiquement ramassés.
- Vos réponses seront notées par une banque de tests JUnit 4. La compilation sans erreur puis chaque test passé avec succès vous rapportera un certain nombre de points.
- Le barème est donné à titre indicatif et pourra être révisé lors de la correction.
- Durant l'épreuve, vous n'avez pas accès à internet. Néanmoins, vous pouvez consulter le site de l'UE et une copie locale de la documentation de l'API Java 8.
- Tous les documents papier (transparents du cours, notes personnelles, livres, etc.) sont autorisés. L'utilisation d'appareils électroniques, de moyens de communication ou d'accès à internet (téléphone, tablette, ordinateur personnel, etc.) est interdite. L'utilisation d'une clé ou d'un disque USB personnel, en lecture seule et étiqueté à votre nom, est autorisée.

## Mise en place : projet Eclipse

L'archive **`MicrOS.zip`** à télécharger contient un projet Eclipse nommé **`MicrOS`**.

- Téléchargez le fichier **`MicrOS.zip`** sur le site de l'UE et décompressez-le dans la racine de votre `$HOME`. Il crée un répertoire `workspace/MicrOS`.
- Lancez Eclipse pour Java en tapant `./eclipse` dans un terminal.
  Spécifiez `workspace` comme répertoire de travail d'Eclipse.
  Il est possible qu'Eclipse se bloque en essayant d'installer un connecteur SVN. Dans ce cas, tapez dans un terminal : `killall eclipse; killall java` et relancez Eclipse.
- Importez le projet Java contenu dans `workspace/MicrOS` : allez dans « File / Open Projects From Filesystem… », cliquez sur « Directory… », choisissez le répertoire `workspace/MicrOS`, puis « Finish ».
- Nous travaillerons dans le projet **`MicrOS`**, dans le package **`pobj.micros`** et ses sous-packages. Les fichiers à fournir seront à placer dans des sous-répertoires de **`workspace/MicrOS/src/pobj/micros/`**.

Assurez-vous de :

- respecter les noms de classe, de package et de chemin de fichier indiqués dans l'énoncé ;
- fournir des sources qui compilent, même si elles ne répondent pas complètement à la question (les méthodes qui fonctionnent pourront vous rapporter une partie des points) ;
- implanter les interfaces et hériter des classes spécifiées dans l'énoncé ;
- respecter les consignes de visibilité des méthodes ; les attributs seront toujours privés ;
- ne pas modifier les interfaces fournies, sauf quand demandé explicitement dans l'énoncé ; sinon, les tests de notation, qui sont programmés vis-à-vis de ces interfaces, risquent de ne plus compiler.

Un source qui n'est pas trouvé par le correcteur ou qui ne compile pas ne rapporte aucun point.

## Plan de l'examen

Durant cette épreuve, nous allons programmer quelques éléments d'un système d'exploitation miniature, que nous appellerons MicrOS.

L'examen comporte **deux parties indépendantes** :

1. un système de fichiers, implanté dans `pobj.micros.fs` ;
2. un gestionnaire de tâches, implanté dans `pobj.micros.scheduler`.

Le projet `MicrOS` fourni contient les interfaces mentionnées dans l'énoncé. Le package `pobj.micros.test` contient des tests JUnit pour évaluer vos réponses. Les tests utilisés pour la notation, bien que dans le même esprit, seront différents et plus exhaustifs. Enfin, le package `pobj.micros.errors` contient une classe d'exception dédiée au projet : `OSError`.

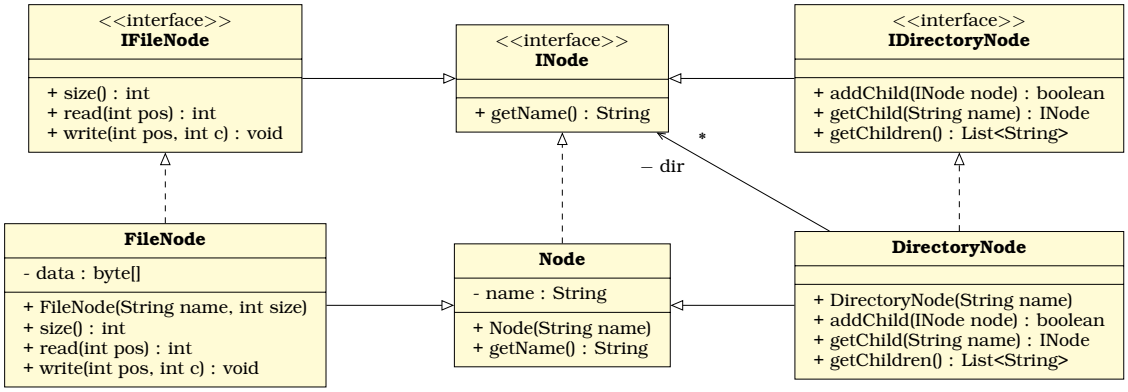
# 1 Système de fichiers (12 pt)

## 1.1 Arborescence

Un système de fichiers est une arborescence. Les nœuds de cet arbre obéissent à l'interface `INode` et représentent soit des fichiers, soit des répertoires (d'interface `IFileNode` et `IDirectoryNode`). Tout nœud possède un nom. Les fichiers contiennent en plus un tableau d'octets (`byte[]` en Java). Les répertoires contiennent une table d'association des noms vers des nœuds.

Nous allons programmer des classes `Node` (question a), `FileNode` (question b) et `DirectoryNode` (question c) qui implantent ces trois interfaces. `Node` implante uniquement la gestion du nom de fichier, commune à tous les nœuds. Les classes `FileNode` et `DirectoryNode` s'appuient sur `Node` par héritage et y ajoutent les comportements spécifiques des fichiers et des répertoires.

Voici un diagramme UML de la construction complète obtenue à l'issue de la question c :




### a) Nœud générique : classe `Node` (1 pt)

La classe `Node` factorise la gestion du nom. Elle implante la méthode `getName` de `INode`.

**À faire :** une classe `pobj.micros.fs.Node` qui implante l'interface `INode` et :

- a un attribut (privé, comme tous les attributs) chaîne de caractères (`name` sur le diagramme) ;
- un accesseur publique `String getName()` pour cet attribut ;
- un constructeur public `Node(String name)` permettant de fixer cet attribut.

**À fournir :** un fichier `workspace/MicrOS/src/pobj/micros/fs/Node.java`

**Fourni :** dans l'archive `MicrOS.zip`, l'interface `pobj.micros.fs.INode` et une classe de test JUnit `pobj.micros.test.NodeTest`

### b) Fichiers : classe `FileNode` (1.5 pt)

Un nœud fichier contient un tableau nu d'octets (`byte[]`), de taille fixée à la création, et des méthodes `read`, `write`, `size` pour y accéder.

**À faire :** une classe `pobj.micros.fs.FileNode` qui implante `IFileNode`, hérite de `Node` et a :

- un constructeur public `FileNode(String name, int size)` fixant le nom et la taille du fichier ;
- un attribut privé de type tableau nu d'octets (`data` sur le diagramme UML) ;
- une méthode publique `int size()` retournant la taille du fichier ;
- une méthode publique `void write(int pos, int c)` qui écrit `c` à la position `pos` du tableau, après avoir converti `c` en octet (la conversion s'écrit en Java : `(byte)c`) ; en cas d'accès en dehors des bornes du tableau, la méthode ne fait rien (en particulier, elle ne signale pas d'exception) ;

- une méthode publique `int read(int pos)` qui retourne l'octet à la position `pos` ; en cas d'accès en dehors des bornes, la méthode ne signale pas d'exception et retourne la valeur spéciale 255.

**À fournir :** un fichier `workspace/MicrOS/src/pobj/micros/fs/FileNode.java`

**Fourni :** l'interface `pobj.micros.fs.IFileNode` et une classe de test JUnit `pobj.micros.test.FileNodeTest`

#### c) Répertoires : classe `DirectoryNode` (2 pt)

Un répertoire est une collection de sous-nœuds de type `INode`. Afin de pouvoir retrouver efficacement un sous-nœud par son nom, nous utilisons une table associative, obéissant à l'interface `Map<String,INode>` de la bibliothèque de collections Java.

**À faire :** une classe `pobj.micros.fs.DirectoryNode` qui implante `IDirectoryNode`, hérite de `Node` et a :

- un attribut privé de type `Map<String,INode>` (`dir` sur le diagramme UML) ;
- un constructeur public `DirectoryNode(String name)` qui crée un répertoire vide de nom donné ;
- une méthode publique `boolean addChild(INode node)` pour ajouter un sous-nœud ; il s'agit donc d'ajouter une association de clé `node.getName()` et de valeur `node` dans la table `dir` ; si un nœud de même nom existe déjà, alors `addChild` ne fait rien et retourne `false` ; sinon, le nœud est ajouté et la méthode retourne `true` ;
- une méthode publique `List<String> getChildren()` retournant la liste des noms de sous-nœuds ;
- une méthode publique `INode getChild(String name)` qui retourne le sous-nœud du nom précisé, ou `null` s'il n'existe aucun nœud de ce nom.

**À fournir :** un fichier `workspace/MicrOS/src/pobj/micros/fs/DirectoryNode.java`

**Fourni :** l'interface `pobj.micros.fs.IDirectoryNode` et une classe de test JUnit `pobj.micros.test.DirectoryNodeTest`

### 1.2 Interface de flux : classe `FileStream` (1.5 pt)

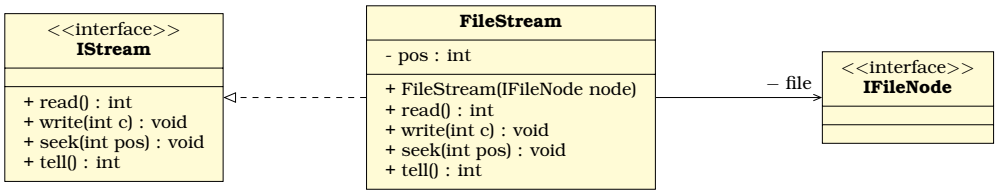
*Note : les questions 1.2, 1.3 et 1.4 sont indépendantes les unes des autres.*

Notre OS n'expose pas directement l'interface `IFileNode`, interne à notre implantation, mais l'interface `IStream`. Un objet de cette interface représente un fichier ouvert sous forme de flux d'octets. Il maintient un attribut entier indiquant la position courante et possède les méthodes publiques suivantes :

- `int read()` retourne l'octet à la position courante et incrémente la position ; après la fin du fichier, la position n'est pas changée et la valeur 255 est retournée ;
- `void write(int c)` modifie l'octet à la position courante et incrémente la position ; après la fin du fichier, l'opération n'a aucun effet (la position n'est pas changée) ;
- `void seek(int pos)` change la position courante, 0 étant le début du fichier ;
- `int tell()` retourne la position courante.

Nous allons programmer une classe `FileStream` permettant d'exposer un fichier `IFileNode` sous forme d'interface `IStream`. Le constructeur public `FileStream(IFileNode node)` précise le nœud fichier à ouvrir et se positionne au début du fichier (position 0). Les opérations de lecture et d'écriture sont déléguées à l'objet `node` spécifié à la construction.

Voici le schéma UML correspondant :




**À faire :** une classe `pobj.micros.fs.FileStream` qui implante `IStream` et a un constructeur public `FileStream(IFileNode)`.

**À fournir :** un fichier `workspace/MicrOS/src/pobj/micros/fs/FileStream.java`

**Fourni :** l'interface `pobj.micros.fs.IStream` et une classe de test JUnit `pobj.micros.test.FileStreamTest`

## 1.3 Copie récursive (2 pt)

Nous ajoutons aux nœuds de l'arborescence une méthode `INode copy()` qui effectue une copie en profondeur des fichiers et répertoires.

**À faire :** ajoutez dans l'interface `INode` une méthode :

- `INode copy()`

Implantez cette méthode publique dans toutes les classes obéissant à l'interface `INode` :

- dans `Node`, `copy()` retourne un nouveau nœud de nom identique ;
- dans `FileNode`, `copy()` retourne un nouveau nœud fichier de même nom et contenant une copie du tableau ; ainsi, toute modification par `write` de la copie ne modifie pas le tableau original ;
- dans `DirectoryNode`, `copy()` retourne une copie du répertoire ; la méthode doit copier récursivement la table et les sous-nœuds (copie en profondeur).

**À fournir :** des versions à jour des fichiers `workspace/MicrOS/src/pobj/micros/fs/INode.java`, `Node.java`, `FileNode.java` et `DirectoryNode.java`

**Fourni :** une classe de test JUnit `pobj.micros.test.CopyNodeTest`

## 1.4 Gestion de haut niveau des fichiers

Une application interagit avec le système de fichiers par une interface `IFileSystem` offrant une vue de haut niveau. Elle offre des méthodes `createDirectory`, `listDirectory` et `openFile` pour créer un répertoire, lister son contenu, ouvrir un fichier (question b). Ces méthodes identifient les répertoires par des chemins, sous forme de chaînes de caractères (question a).

#### a) Résolution des chemins : classe `NodeUtils` (2 pt)

Pour l'application, un répertoire est représenté par un chemin : une chaîne de caractères où les noms de répertoire sont séparés par le caractère `'/'`. Ainsi `"a/b/c"` représente le répertoire nommé `"c"` du sous-répertoire `"b"` du sous-répertoire `"a"` du répertoire racine, tandis que la chaîne vide `""` représente le répertoire racine lui-même. Une tâche commune aux méthodes de `IFileSystem` est de retrouver un répertoire par son chemin. Nous allons factoriser ce code dans une méthode statique `findDirectory` d'une classe utilitaire `NodeUtils`.

**À faire :** une classe `pobj.micros.fs.NodeUtils` contenant une méthode publique statique :

- `IDirectoryNode findDirectory(IDirectoryNode root, String path) throws OSError`

qui retourne le répertoire accessible depuis `root` en suivant le chemin `path`. Par exemple, `findDirectory(root,"")` retourne `root`, tandis que `findDirectory(root,"a")` retourne `root.getChild("a")` converti en `IDirectoryNode`, en supposant que `getChild` retourne bien un objet non `null` d'interface `IDirectoryNode` ; si c'est le cas, `findDirectory(root,"a/b")` retournera alors son entrée `b` convertie en `IDirectoryNode`, etc. Si le répertoire `path` n'existe pas, alors l'exception `OSError` est signalée avec pour message `"Invalid path"`.

Vous pourrez utiliser la méthode `split` de `String`. Ainsi, si `x` est une chaîne représentant un chemin, alors `x.split("/")` retourne un tableau de chaînes `String[]` représentant les éléments de chemin. Par exemple, si `x` vaut `"aa/bb/cc"`, `x.split("/")` retourne le tableau `["aa","bb","cc"]`. Attention, certains éléments du tableau sont des chaînes vides, qu'il faut ignorer. En particulier, si `x` est la chaîne vide, alors `x.split("/")` retourne `[""]`. Notez enfin que les chemins `""` et `"/"` doivent être synonymes, de même que `"a/b"` et `"/a/b/"`.

**À fournir :** un fichier `workspace/MicrOS/src/pobj/micros/fs/NodeUtils.java`.

**Fourni :** une classe de test JUnit `pobj.micros.test.NodeUtilsTest`

#### b) Gestion des fichiers : classe `FileSystem` (2 pt)

La classe `FileSystem` représente un système de fichiers. Elle possède :

- un attribut privé de type `IDirectoryNode` représentant la racine du système de fichiers ;
- un constructeur public `FileSystem()` sans argument qui initialise la racine à un nouveau répertoire, initialement vide, de nom `"root"` (ce nom importe peu).

Par ailleurs, elle implante les méthodes publiques suivantes de `IFileSystem` :

- `List<String> listDirectory(String path)` trouve le répertoire de chemin `path` dans le système de fichiers et retourne la liste de ses fichiers et sous-répertoires ;
  si `path` n'existe pas, l'exception `OSError` est signalée avec le message `"Invalid path"` ;
- `void createDirectory(String path, String name)` ajoute au répertoire de chemin `path` un répertoire vide de nom `name` ;
  si `path` n'existe pas, ou si un fichier ou répertoire `name` existe déjà dans `path`, l'exception `OSError` est signalée avec comme message `"Invalid path"` ;
- `IFileNode openFile(String path, String name, int size)` retourne le fichier de nom `name` dans le répertoire de chemin `path`, si un tel fichier existe ;
  si `path` n'existe pas, l'exception `OSError` est signalée avec comme message `"Invalid path"` ;
  si `name` n'existe pas dans ce chemin, un fichier est créé avec pour taille `size` et est retourné ;
  si `name` existe mais est un sous-répertoire au lieu d'un fichier, `OSError` est signalée avec comme message `"Invalid file"`.

**À faire :** une classe `pobj.micros.fs.FileSystem` implantant l'interface `IFileSystem` et respectant la spécification ci-dessus.

**À fournir :** un fichier `workspace/MicrOS/src/pobj/micros/fs/FileSystem.java`.

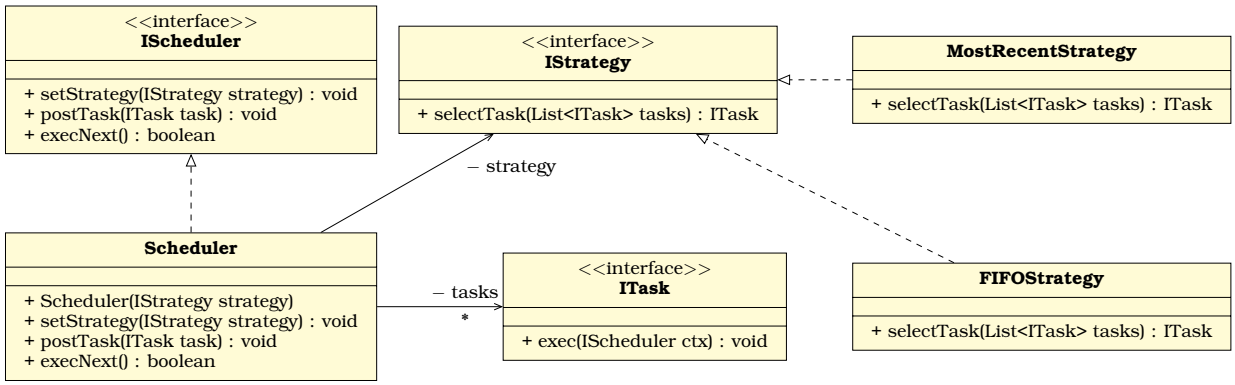
**Fourni :** l'interface `pobj.micros.fs.IFileSystem` et une classe de test JUnit `pobj.micros.test.FileSystemTest`

# 2 Gestion des tâches (8 pt)

## 2.1 Ordonnanceur

Un ordonnanceur (question b) orchestre l'exécution des tâches. Il obéit à l'interface `IScheduler` et maintient une liste ordonnée, `List<ITask>`, de tâches à exécuter. Cette liste est dynamique : une tâche est enlevée de la liste au moment de son exécution, mais l'exécution d'une tâche peut ajouter de nouvelles tâches. Le système appellera donc l'ordonnanceur en boucle jusqu'à épuisement des tâches (question c). Il existe plusieurs stratégies de sélection de la prochaine tâche à exécuter ; l'ordonnanceur déléguera ce choix à des classes séparées (question a).

Voici un diagramme UML décrivant l'implantation à la fin de la question 2.1.b :




### a) Stratégie de sélection des tâches : classes `MostRecentStrategy` et `FIFOStrategy` (1 pt)

L'interface `IStrategy` possède une seule méthode :

- `ITask selectTask(List<ITask> tasks)`

qui ôte un élément de la liste `tasks` et le retourne. Si la liste est vide, `null` est retourné.

Nous supposons que les tâches les plus récemment ajoutées se trouvent en tête de liste. Nous proposons alors deux implantations de `IStrategy` :

- `FIFOStrategy` (premier arrivé, premier servi) choisit la tâche la plus ancienne de la liste ;
- `MostRecentStrategy` (dernier arrivé, premier servi) choisit la tâche ajoutée la plus récemment.

**À faire :** des classes `pobj.micros.scheduler.MostRecentStrategy` et `pobj.micros.scheduler.FIFOStrategy` qui implantent `IStrategy` et obéissent à la spécification ci-dessus.

**À fournir :** des fichiers `workspace/MicrOS/src/pobj/micros/scheduler/MostRecentStrategy.java` et `FIFOStrategy.java`

**Fourni :** l'interface `pobj.micros.scheduler.IStrategy` et une classe de test JUnit `pobj.micros.test.StrategyTest`

#### b) Ordonnanceur et tâches : classe `Scheduler` (2 pt)

L'interface `ITask` des tâches a une seule méthode :

- `void exec(IScheduler ctx) throws OSError`

Exécuter une tache consiste à exécuter sa méthode `exec`. L'ordonnanceur est passé en argument à la tâche ; elle peut ainsi demander à l'ordonnanceur l'ajout de nouvelles tâches. L'exécution peut générer des erreurs `OSError`, ce que nous indiquons par `throws OSError`.

La classe `Scheduler` obéit à l'interface `IScheduler` et a :

- un constructeur public `Scheduler(IStrategy strategy)` spécifiant la stratégie initiale ;
- une méthode publique `void setStrategy(IStrategy strategy)` pour changer la stratégie ;
- un attribut privé `List<ITask>` maintenant la liste des tâches, initialement vide ;
- une méthode publique `void postTask(ITask task)` ajoutant une tâche en tête de la liste ;
- une méthode publique `boolean execNext() throws OSError` qui utilise la stratégie pour choisir une tâche à exécuter ; si la liste est vide, `false` est retourné ; sinon, la tâche choisie est supprimée de la liste, sa méthode `exec` est exécutée et `true` est retourné.

**À faire :** une classe `pobj.micros.scheduler.Scheduler` qui implante `IScheduler`.

**À fournir :** un fichier `workspace/MicrOS/src/pobj/micros/scheduler/Scheduler.java`

**Fourni :** l'interface `pobj.micros.scheduler.IScheduler` et une classe de test JUnit `pobj.micros.test.SchedulerTest`

#### c) Boucle d'exécution : classe `TaskRunner` (1 pt)

**À faire :** une classe `TaskRunner` ayant une unique méthode publique statique :

- `void run(IScheduler scheduler)`

qui exécute en boucle `scheduler.execNext()` tant qu'il reste des tâches à exécuter (c'est-à-dire, tant que `execNext` retourne `true`). Si l'exécution d'une tâche signale une exception `OSError`, le message de l'exception doit être affiché sur la console (méthode `getMessage()`) et l'exécution de la boucle se poursuit avec la tâche suivante.

**À fournir :** un fichier `workspace/MicrOS/src/pobj/micros/scheduler/TaskRunner.java`

**Fourni :** une classe de test JUnit `pobj.micros.test.TaskRunnerTest`

### 2.2 Exemple de tâche : classes `ExampleTask` et `ExampleMain` (2 pt)

Nous programmons ici un exemple d'utilisation de l'ordonnanceur et des tâches. Notre tâche, `ExampleTask`, a un attribut entier `n`, fixé lors de la construction. Lors de son exécution, elle affiche cet entier sur la console et, si $n > 0$, elle ajoute, avec `postTask`, deux nouvelles tâches avec pour attribut `n-1`. Ainsi, avec la stratégie FIFO et une valeur initiale de 2, l'exécution affichera `2110000` (la tâche affiche 2 et crée deux tâches qui affichent 1 et créent chacune deux tâches qui affichent 0).

**À faire :** une classe `pobj.micros.scheduler.ExampleTask` qui implante `ITask` et obéit à la spécification ci-dessus, et une classe `pobj.micros.scheduler.ExampleMain` qui a un point d'entrée :

- `public static void main(String[] args)`

dont l'exécution a pour effet de créer un ordonnanceur `Scheduler` (question 2.1.b) avec la stratégie FIFO (question 2.1.a), d'y ajouter une tâche `ExampleTask` avec `n` = 3, et de lancer l'exécution avec `TaskRunner` (question 2.1.c).

**À fournir :** des fichiers `workspace/MicrOS/src/pobj/micros/scheduler/ExampleTask.java` et `workspace/MicrOS/src/pobj/micros/scheduler/ExampleMain.java`

**Fourni :** une classe de test JUnit `pobj.micros.test.ExampleTaskTest`

## 2.3 Bibliothèque de tâches : classe `Service` (2 pt)

*Note : cette question est indépendante des précédentes.*

Nous supposons maintenant que le système maintient une bibliothèque de tâches `ITask` répertoriées par un nom de service (chaîne de caractères) et un numéro de version (un entier). Ce répertoire sera implanté par une table d'association `Map<IService,ITask>`, où les clés `IService` décrivent des paires nom / version. Nous réalisons ici une implantation `Service` de l'interface `IService` utilisable comme clés dans cette table. Ces clés sont immuables.

**À faire :** une classe `Service` qui implante `IService` et a :

- des attributs privés immuables `name` de type `String` et `version` de type `int` ;
- les getters publics `getName()` et `getVersion()` associés ;
- un constructeur public `Service(String,int)` qui fixe la valeur de ces attributs ;
- une méthode `toString` qui retourne une chaîne de la forme `"nom/version"` ;
- des méthodes `equals` et `hashCode` permettant d'utiliser les instances de `Service` comme clés dans des tables d'association.

**À fournir :** un fichier `workspace/MicrOS/src/pobj/micros/scheduler/Service.java`

**Fourni :** l'interface `pobj.micros.scheduler.IService` et une classe de test JUnit `pobj.micros.test.ServiceTest`