Cette première version de HDFS permet l'écriture, la lecture et la suppression d'un fichier sous forme de chunks sur des machines (potentiellement) distantes.

La gestion du système de fichiers se fait depuis HdfsClient.
Les commandes se lancent de la manière suivante :

- Lecture : `java hdfs.HdfsClient -r <filename> [ <destfile> ]`
  La valeur par défaut de `<destfile>` est `r_<filename>`
- Ecriture : `java hdfs.HdfsClient -w <filename> -f ln|kv`
  Les formats supportés sont LINE (ln) et KV (kv).
  L'argument optionnel `--chunk-size=<mode>` permet de spécifier le choix de taille des chunks : `distributed` indique que pour si N serveurs sont disponibles, on aura k <= N chunks, c'est équivalent au passage d'un entier négatif. Si `<mode>` est un entier (long) strictement positif, alors il représente la taille en octet des chunks.
- Suppression : `java hdfs.HdfsClient -d <filename>`

On peut lister sommairement les fichiers enregistrés avec `java hdfs.HdfsClient -l`.

Spécifications techniques :

Les fichiers développés dans cette partie sont dans le package `hdfs`.
HDFS utilise le port 3000 dans cette version et les ip des serveurs distants sont hard codées dans HdfsClient. Un fichier de configuration texte contenant les ip / ports est prévu pour les versions ultérieures.

- Metadata & Filedata
  Ces deux classes sérialisables définissent les métadonnées des fichiers contenus dans Hdfs. Elles permettent entre autres de récupérer la liste des chunks pour un fichier et localiser facilement les serveurs où récupérer un chunk.
  Le nom du fichier de métadonnées pour HDFS est contenu dans la constante `HdfsClient.DATAFILE_NAME`.

- HdfsClient
  C'est la classe gestionnaire du système de fichier. Elle permet sa gestion en ligne de commande comme indiqué plus haut.
  Les 3 méthodes centrales sont celles associées à ses commandes `HdfsRead`, `HdfsWrite` et `HdfsDelete`. Toutes trois ayant à communiquer avec des serveurs multiples, elles font appel à un pool de threads. Chaque tâche (ie. interaction avec un serveur distant, et/ou copie d'un bout de fichier local <-> distant) est représentée par une classe implémentant un Callable (respectivement `Read`, `Write` et `Delete`). Cette dernière renvoie un objet de type `OperationResult<T>` encapsulant le résultat de la transaction avec un `HdfsServer`. Ce résultat peut être un objet quelconque de type T.

Pour cette première version, il s'agit d'un booléen dans le cas de Write et Delete (vrai si tout s'est passé correctement, faux sinon) et d'un fichier local temporaire (copie du chunk) dans le cas de Read (qui est null s'il y a eu une erreur). On peut donc seulement à ce stade savoir si la transaction s'est faite sans avoir plus de détails. De plus, une erreur indiquée par une valeur false ou null n'est pas encore traitée dans cette version. Ce sera bien sûr le cas dans les versions à venir.
Enfin pour la fiabilité du service, il est prévu d'ajouter des échanges de messages de confirmation de transaction entre client et serveur.

- HdfsServer
  Cette classe est le daemon actif sur les machines distantes.
  Il s'agit d'une classe qui englobe une classe `Slave` (qui hérite de la classe `Thread`) qui aura pour but d'initialiser la connexion TCP par socket avec un ou plusieurs `HdfsClient`.
  Une connexion avec un client suit ce schéma :
  - Le daemon `HdfsServer` tourne sur une machine et accepte toute connexion par socket sur un port prédéfini.
  - Pour chaque `HdfsClient` se connectant, un nouveau slave se lance permettant un dialogue entre le serveur et chaque client, grâce à la méthode `run()`.
  - Cette méthode `run()` permet d'initier les échanges via un buffer d'octets. On va recevoir dans un premier temps la première commande du client (un `READ`, `WRITE` ou `DELETE`), et de vérifier si elle est correcte (nombre de paramètres conforme).
  - Après réception de la commande elle est donc exécutée :
    `WRITE` : le fichier tronqué est transmis par `HdfsClient` est stocké localement dans un répertoire `data`.
    `READ` : le fichier local sera retransmis au client puis recomposé pour le lire
    `DELETE` : le fichier local sera supprimé
  - Après traitement, la communication sera `close()`.

  Certains traitements d'erreurs ont déjà été implantés : le traitement de la commande et ses paramètres ou encore la gestion d'exception par socket. Pour avoir un meilleur aperçu du bon échange d'informations entre client et serveur, ou encore la signalisation d'erreurs (suppression d'un fichier inexistant par exemple), des échanges de messages (qui sont pour le moment affichés sur console) seront implantés.