Ift232

Devoir #1

Paradigme orienté objet

*N.B. : Ce devoir peut être fait seul(e) ou en équipe de 2. Les équipes de plus de 2 sont **interdites**.*

AVERTISSEMENT :

Lisez les deux annexes à la fin de ce document avant de coder par vous-même. L'un d'eux est un manuel d'instructions simplifié, l'autre est une liste de problèmes classiques qui peuvent vous arriver, si vous avez des bugs bizarres, c'est là qu'il faut aller lire d'abord.

La première partie de ce document est un tutoriel sur le langage Jarvis. FAITES-LE!!!!

Si vous préférez voir l'étendue du travail avant, lisez la partie « travail à réaliser » qui se trouve à la fin.

Mise en contexte

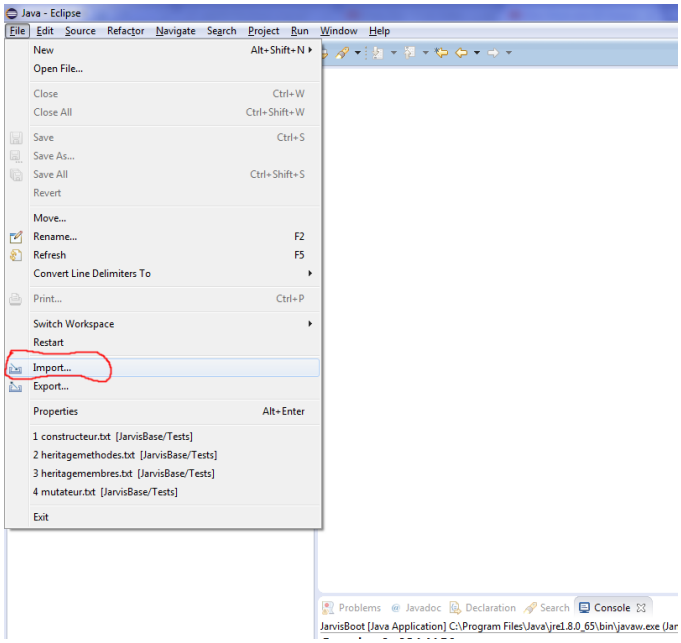
L'objectif de ce devoir est de vous amener à bien comprendre les concepts fondamentaux des langages orientés-objet.

À cette fin, un langage incomplet doublé de son interpréteur vous serviront de point de départ. Le matériel de base est codé en Java, dans l'environnement de développement Eclipse.

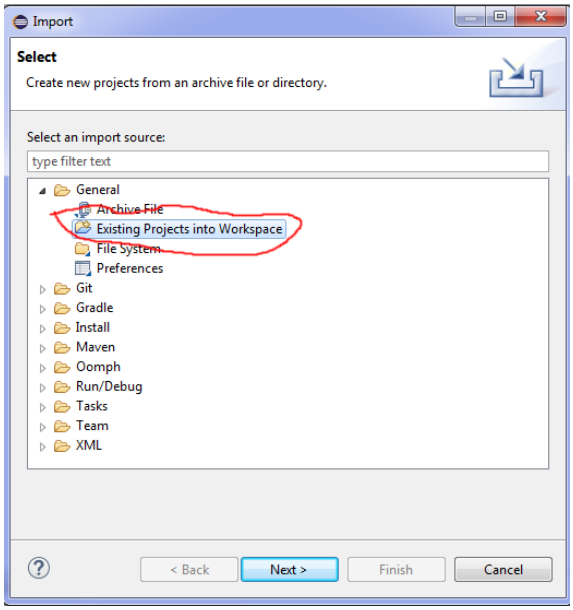
Afin de faire un tour d'horizon du code rapidement, vous pouvez importer l'archive qui se trouve sur le répertoire public (jarvisBase.zip) en tant que projet dans votre espace de travail. Un regard rapide des commentaires d'en-tête de la classe JarvisBoot va vous orienter vers les emplacements d'intérêt dans le code pour ce devoir. Vous pouvez sauter directement à la section qui explique les travaux à faire dans Jarvis pour avoir une idée de la taille du travail à réaliser, ou lire l'introduction (un peu longue) au langage Jarvis ci-après pour bien acquérir le vocabulaire et les notions avant de lire les questions.

Installation de Jarvis

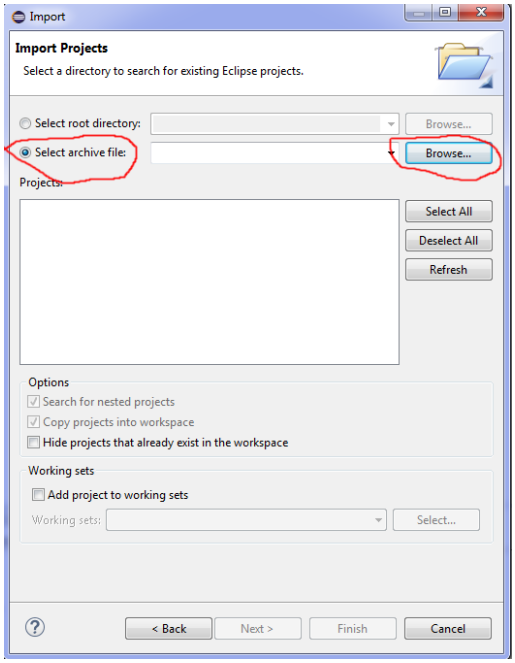
Pour arriver à démarrer l'interpréteur Jarvis, vous devez d'abord créer un projet Eclipse. Pour ce faire, démarrez Eclipse, puis suivez les étapes suivantes :



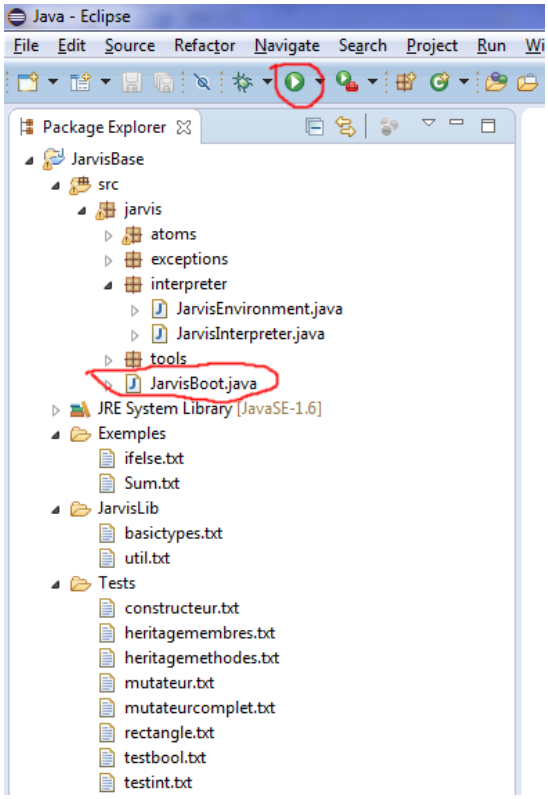
1-Utilisez l'option « import » du menu « file ».



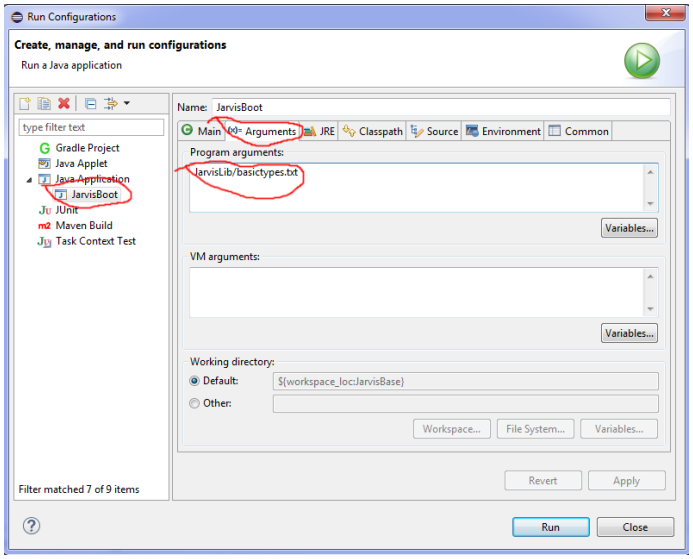
2-Choisissez d'importer un projet existant



3-Assurez-vous d'importer une archive, puis naviguez et trouvez le fichier « JarvisBase4.zip ». Le projet contenu dans le fichier devrait déjà être sélectionné pour l'importation. Appuyez sur « finish ».



4-La classe JarvisBoot contient le programme principal.



5-Si vous voulez que l'interpréteur charge et exécute le contenu d'un fichier en partant, spécifiez-le dans le en ouvrant le menu « run » et l'option « run configurations ». Sous l'onglet « arguments », vous pouvez spécifier ce que le programme recevra comme argument à son démarrage. Le nom du fichier avec son chemin relatif devrait fonctionner. C'est très pratique lorsque vous voulez éviter de retaper les mêmes tests à la main, ou pour toujours démarrer avec certaines définitions de base déjà chargées (basictypes.txt, par exemple).

Le langage Jarvis

Le code qui vous est fourni est un interpréteur qui comprend un langage nommé Jarvis. L'interpréteur lit au clavier une suite de commandes. Ces commandes, une fois interprétées, peuvent donner lieu à la création de variables, d'objets, l'évaluation de fonctions, etc. La liste des commandes comprises par l'interpréteur est fournie en annexe de cet énoncé, avec une description détaillée. Un tour dans la classe JarvisCommand, quoi qu'étourdissant, peut aussi vous renseigner à leur sujet.

Cependant, il est plus simple de lire quelques exemples et les explications qui suivent pour se mettre en selle.

Les atomes

À la base, l'interpréteur du langage Jarvis comprend quelques constructions syntaxiques et sait manipuler quelques types de données. Ces constructions de base sont des ***atomes***, les types minimaux que le langage doit supporter. Le code servant à gérer les atomes se trouve dans le package atoms. L'atome qui vous intéressera le plus est **ObjectAtom**, la représentation de base d'un objet en Jarvis.

Les types de données atomiques sont tous supportés sans aucune opération. Un atome est simplement une chose que l'interpréteur peut stocker, manipuler, lire au clavier, afficher à l'écran. Ce n'est pas un type abstrait de données (ou même l'équivalent d'un type de base en C++ ou en Java). Pour avoir un

type abstrait de données, et donc faire des opérations sur celles-ci, vous avez besoin de définir des **classes**. Ceci alourdit considérablement l'écriture de code en Jarvis, mais vous amènera à penser de façon presqu'exclusivement orientée objet.

Jarvis n'affiche pas les résultats engendrés par les commandes et les constructions du langage. Pour voir les résultats, il faut demander un affichage de ce qui se trouve dans la **file d'arguments** de l'interpréteur. Cette file est votre premier arrêt lorsque vous voulez comprendre ce qui s'est passé après avoir entré des commandes dans l'interpréteur. Vous pouvez en voir le contenu à l'aide de la commande **!args**, ou forcer l'affichage de la valeur en tête de file avec la commande **!p** . Ces commandes sont extrêmement utiles pour déverminer du code Jarvis. La file d'arguments porte également bien son nom, elle est utilisée pour préparer les valeurs à transmettre à une fonction ou à un objet lors d'un envoi de message.

Les types atomiques sont les suivants:

IntAtom représente les nombres entiers. Toute valeur entière **positive** entrée au clavier sera interprétée et transformée en IntAtom. **Attention!!** on ne peut pas entrer des nombres **négatifs** dans l'interpréteur (mais des résultats de calculs peuvent être négatifs et seront affichés correctement).

Soient les commandes suivantes:

```
>5
>!args
Args: (jarvis.atoms.IntAtom@60aeb0,)
>!p
5
```

Entrer **5** dans l'interpréteur place un atome de type IntAtom dans la file d'arguments. La commande **!args** force l'affichage du contenu de la file. La commande **!p** force l'affichage de la valeur en tête de file. L'affichage d'un IntAtom est tout simplement sa valeur. Il en est de même avec la plupart des atomes.

Vous pouvez fabriquer des nouvelles variables (ou plutôt, références à des atomes) avec la commande **!ref**, suivie d'un nom pour la variable et de la valeur qu'elle prendra. Les variables n'ont pas de type, ce ne sont que des symboles qui sont reliés à des atomes. Par exemple:

```
>!ref x 5
>x
>!args
Args: (jarvis.atoms.IntAtom@1270b73,)
>!p
5
>
```

Ici, on fabrique un nouveau symbole, **x**. Ce symbole est relié à un atome de type IntAtom. On demande ensuite son interprétation. Celle-ci place l'atome dans la file d'arguments. On peut ensuite le faire afficher avec la commande !p. À noter, vous pouvez également faire des nouvelles variables qui réfèrent à la valeur qui se trouvait en tête de la file d'arguments, de la façon suivante:

```
>5
5
>!args
Args: (jarvis.atoms.IntAtom@89ae9e,)
>!ref x
>!args
Args: ()
>x
>!args
Args: (jarvis.atoms.IntAtom@1270b73,)
>!p
5
```

Ici, on place l'atome 5 dans la file d'arguments, on fabrique ensuite une nouvelle variable x qui réfère à cet atome (et le sort ainsi de la file), puis on le replace en file et on l'affiche. Cette façon de faire est plus pratique parce que les résultats de calculs se trouvent toujours dans la file d'arguments. Si vous voulez affecter un résultat de calcul à une variable, c'est toujours de cette façon qu'il faut procéder.

Vous pouvez, en tout temps, regarder ce que contient l'environnement d'interprétation en entrant la commande **!e** . Cette commande affiche tous les symboles définis précédemment, ainsi que leur valeur.

Si on entre cette commande à la suite du code ci-haut, on obtient:

```
[Class     |      attributes:(attributes,methods,)methods:...]
[x     |     5]
```

Ce que vous voyez ici est la liste de tout ce que le langage connaît. À la base, seule la variable **Class** est définie. C'est l'instance de la classe de toutes les classes. Regarder le contenu de l'environnement est assez pratique pour déverminer. Pour inspecter une variable de plus près, vous pouvez entrer son nom dans l'interpréteur et forcer son affichage avec !p. N'oubliez pas, elle doit être en tête de file! Si la file n'est pas vide, vous pouvez toujours en forcer la vidange avec la commande **!c** .

Finalement, une classe est déjà écrite en Jarvis pour les entiers. Celle-ci s'appelle int, et elle est définie dans le fichier **basictypes.txt**. Quelques opérations sont supportées sur les objets de cette classe, vous allez devoir en ajouter.

BoolAtom représente les valeurs de vérité. Il peut prendre les valeurs **true** ou **false**. Ces valeurs entrées dans l'interpréteur placeront un atome BoolAtom dans la file d'arguments. Sa classe est également définie dans basictypes.txt, mais aucune opération n'est supportée pour l'instant.

StringAtom représente les chaînes de caractères. Il faut les entourer de guillements (" ") lorsqu'on les entre dans l'interpréteur. **Attention!!** on ne peut pas entrer des chaînes contenant des **espaces** dans l'interpréteur!

ListAtom représente les listes. Vous pouvez spécifier des listes d'atomes en ouvrant des parenthèses. Celles-ci peuvent contenir des valeurs atomiques simples, ou de symboles connus.

Voici un exemple de création d'une liste:

```
>(5 true "allo")
(5,true,allo,)
>!ref laliste
>!e
[[laliste     |     (5,true,allo,)]
[Class     |      attributes: ... methods:...]
]
```

Aucune opération de manipulation de liste n'est supportée dans le langage Jarvis (pas de classe liste définie dans le langage!) Les listes sont utilisées principalement pour énumérer les **attributs**, ou **membres**, d'une classe. Lorsqu'on veut construire une nouvelle classe, il faut d'abord construire la liste de ses attributs.

DictionnaryAtom représente les tables de hashage. Elles servent à faire des associations entre des atomes et d'autres atomes. Habituellement, on associe des valeurs à des symboles, tout comme dans l'environnement d'interprétation (il n'est en fait qu'un DictionnaryAtom!). On peut cependant associer n'importe quel atome à un autre. Le premier sera la clé, le second, la valeur. La spécification d'un dictionnaire se fait de la façon suivante:

```
>[ "nombre" 5 "valverite" false "chaine" "allo"]
[[chaine     |     allo]
[nombre     |     5]
[valverite     |     false]
]
>!ref ledict
>!e
[[ledict     |     jarvis.atoms.DictionnaryAtom@1270b73]
```

```
[Class      |      attributes: ... methods: ...]
]
>
```

Vous remarquerez, ici, que l'affichage d'un dictionnaire (**ledict**) se trouvant dans un autre (l'environnement), ne se fait pas récursivement, par souci de rendre lisible l'affichage. Pour voir le contenu d'un dictionnaire, placez-le dans la file d'arguments et utilisez la commande **!p**.

Tout comme la liste, le dictionnaire n'a pas sa classe correspondante en Jarvis, aucune opération n'est donc supportée pour manipuler les dictionnaires à travers l'interpréteur.

NullAtom représente la valeur **null**. L'entrée de cette valeur dans l'interpréteur fabrique un atome de ce type. Ce type d'atome est nécessaire lorsqu'on a besoin d'une valeur spéciale, indéfinie ou pour toutes sortes de cas spéciaux ou limites.

ObjectAtom représente un **objet**. Un objet est un ensemble de valeurs (pour ses variables d'instance) plus une référence à un autre objet, sa classe. Sa classe devrait décrire quel sont ses **attributs** (variables d'instance) et quelles sont ses **méthodes**. Ce type d'atome peut faire quelque-chose de spécial: on peut lui envoyer un message. Les messages ont la forme suivante:

```
!(obj sel args...)
```

Où **obj** est l'objet destinataire du message, **sel** est le sélecteur (le nom de l'attribut ou de la méthode qu'on désire avoir), et **args** sont les arguments (valeurs) qu'on veut passer avec le message, si le sélecteur est une méthode. Les arguments peuvent se trouver dans la file d'arguments avant l'envoi du message. Le destinataire et le sélecteur peuvent aussi avoir été placés dans la file d'arguments avant qu'on demande à l'interpréteur d'envoyer un message! Lisez attentivement l'exemple suivant:

```
!load JarvisLib/basictypes.txt
>!(int new 7)
>!args
Args: (jarvis.atoms.ObjectAtom@89ae9e,)
>!ref x
>!(x value)
>!p
7

>!c
>x
>!p
```

```
value:7
```

Ici, on charge d'abord la bibliothèque de types de base. Ensuite, on fait un premier envoi de message en commençant un énoncé par **!(** .

Le message demande à l'objet **int** (qui est une classe) de créer une nouvelle instance (la méthode est **new**), avec comme argument, **7**. Après un bref passage dans la file d'arguments, notre nouvel entier se trouve référencé par la variable **x**. Cette variable n'est pas un atome IntAtom! C'est un objet qui contient un champ **value**, qui a un IntAtom valant 7 comme valeur.

Le second message qu'on envoie est à l'objet **x**, pour obtenir son attribut **value**. Celui-ci est retourné via la file d'arguments. On affiche ensuite sa valeur. Finalement, on place l'objet **x** dans la file d'arguments et on force son affichage. L'affichage d'un objet est la liste de ses attributs avec leurs valeurs.

L'environnement de l'interpréteur à ce point-là ressemble à ceci:

```
[int     |      attributes:(value,) methods: ...]
[ifelse     |      attributes:(true,false,) methods: ...]
[Class     |      attributes:(attributes,methods,) ...]
[bool     |      attributes:(value,) methods: ...]
[x     |      value:7]
```

Vous remarquerez que les objets int, bool et ifelse se sont ajoutés à notre environnement, ils sont définis dans basictypes.txt, et représentent trois classes particulièrement utiles: les entiers, les booléens, et une construction de base pour faire des conditions.

Si on veut voir plus en détails de quoi est faite la classe int, on peut la mettre en file et forcer son affichage avec !p:

```
attributes:(value,)
methods:
[==     |     IntegerEqualsFunction]
[*     |     IntegerMultiplyFunction]
[+     |     IntegerAddFunction]
[-     |     IntegerSubtractFunction]
```

On obtient ici un portrait un peu plus clair de la situation! La classe int telle que définie dans basictypes.txt supporte donc quelques méthodes. On peut les essayer:

```
!c
>!(x + 5)
>!ref total
```

```
>!e
[total      |       value:12]
...
```

Ici, on a additionné un atome IntAtom à un objet int (l'objet int sait s'additionner à des atomes ou à d'autres ints). On a simplement envoyé le message "**+**" à notre objet, avec l'atome **5** comme argument.

On peut faire tout cela à l'aide de la file d'arguments également. Par exemple, on peut y placer les arguments d'un message avant de faire l'envoi, comme ceci:

```
8
>!args
Args: (jarvis.atoms.IntAtom@8813f2,)
>!(x +)
>!args
Args: (jarvis.atoms.ObjectAtom@1d58aae,)
>!p
 value:15
>
```

Ici, on a placé l'atome 8 dans la file. Ensuite, on envoie le message + à x. Le résultat est un nouvel objet int, avec le champ value à 15. Celui-ci a besoin d'un argument, s'il ne se trouve pas dans la liste entre parenthèses, il complètera avec ce qui se trouve dans la file. Attention! Au total, ce qui se trouve dans la parenthèse, PLUS ce qui se trouve dans la file constitue les arguments de votre méthode. Si le compte n'est pas bon, l'interpréteur va lancer une erreur!

Vous pouvez aussi utiliser la file d'arguments pour le destinataire et le sélecteur, comme ceci:

```
x
>"+"
>total
>!args
Args: (ObjectAtom@89ae9e, StringAtom@83cc67, ObjectAtom@e09713,)
>!(!a !a)
>!p
 value:19
```

Ici, c'est un peu plus tordu. On a placé tout ce qui servait à faire le message dans la file d'arguments. l'objet x, (le destinataire), le nom de la méthode ("+"), ainsi que l'objet total (l'argument). Ensuite, on fait un envoi de message en spécifiant que le destinataire et le sélecteur se trouvent dans la liste d'arguments (avec **!a**). Attention, ils doivent s'y trouver dans le bon ordre! On utilise !a uniquement pour remplacer le destinataire et/ou le sélecteur avec des atomes se trouvant dans la file d'arguments.

JarvisClosure représente une fonction qui s'évalue dans un environnement qui lui est propre (une fermeture). Une fermeture est constituée d'une liste de symboles qui seront ses paramètres, terminée par un point, ainsi que d'une suite de commandes pour l'interpréteur, terminée par une accolade fermante. L'interprétation d'une fermeture associe des arguments (qui doivent se trouver dans la file d'arguments au préalable!) aux paramètres, puis lance l'exécution des commandes spécifiées dans le corps de la fermeture. Voici un exemple simple:

```
>!{ param .
!e
!ref x 5
!e
param
!p
}
(param){[!e, !ref, x, 5, !e, param, !p]}
>!ref func
>!args
Args: ()
>"bonjour"
>func
[[param     |     bonjour]
]
[[param     |     bonjour]
[x     |     5]
]
bonjour
>!args
Args: (jarvis.atoms.StringAtom@1270b73,)
>!p
bonjour
>!e
[Class     |     attributes: ... methods: ...]
[func     |     jarvis.atoms.JarvisClosure@16caf43]
]
```

Bon, pas si simple que ça! On définit d'abord un atome JarvisClosure en commençant par écrire **!{** . On spécifie ensuite la liste des paramètres, se terminant par un point (**param .**). Le seul paramètre se nomme param. On spécifie le corps de notre fonction: afficher l'environnement, définir une variable x qui vaut 5, afficher encore l'environnement, puis, interpréter le paramètre reçu.

On place ensuite cette fermeture dans une variable qui s'appelle **func**. Pour interpréter func, il faut d'abord enfiler son argument. Une fermeture n'est pas un objet! Il ne faut pas lui envoyer de message! Lorsque l'interpréteur évalue un atome JarvisClosure, il ne peut que l'exécuter, c'est tout ce qu'il sait faire! Il suffit donc d'entrer **func** dans l'interpréteur pour évaluer la fonction, avec l'argument "bonjour", qu'on a placé dans la file au préalable. Les deux affichages d'environnement donnent deux résultats différents:

```
[[param     |     bonjour]
]
```

Cet environnement est local à notre fermeture. Il ne contient que la variable **param**. Après la définition de **x**, il contient deux variables:

```
[[param     |     bonjour]
[x     |     5]
]
```

Finalement, on demande à l'interpréteur d'évaluer param. Ceci le place dans la file d'arguments. On l'affiche avec la commande !p. La fermeture se termine ensuite et l'environnement local est détruit.
Cependant, "bonjour" **se trouve toujours dans la file d'arguments!!** Ainsi, il est possible de retourner des valeurs par effet de bord, en les plaçant tout simplement dans la file d'arguments.
Un examen de l'environnement de base est révélateur: ni **x** ni **param** n'existent en dehors de l'environnement propre à un appel de notre fonction.

IMPORTANT

Bien qu'une fermeture s'évalue dans un environnement local, elle a accès à toutes les variables de l'environnement à partir duquel elle est évaluée, et ce, récursivement jusqu'à l'environnement global.
Ceci implique que vous pouvez faire ce qui suit:

```
>!ref x 5
>!{.
!e
x
!p
}
>!ref func
>func
[]
5
>
```

Ici, **func** est une fermeture qui ne prend aucun paramètre. Elle affiche son environnement local, puis évalue **x** (qui est défini dans l'environnement global) et l'affiche avec **!p**. Son environnement local est bel et bien vide (**[]**) !

Lorsqu'un symbole est connu dans l'environnement local, on ne va pas chercher dans les environnements parents. **Les symboles de l'environnement local ont donc priorité sur les symboles se trouvant dans les environnements appelants**.
Cette faculté essentielle du langage permet de définir des fonctions dans des fonctions, celles-ci agissant librement sur les variables se trouvant dans leur environnement parent, ainsi que de faire des boucles par récursitivé (comme dans **Sum.txt**)

De plus, lorsqu'une méthode d'un objet est appelée (par passation de message), une variable spéciale est créée et acessible de l'environnement de la méthode: la variable **self**, qui est une référence à l'objet qui a reçu le message.

Travail à réaliser

Le travail à réaliser est séparé en plusieurs parties. Celles-ci **doivent** être réalisées dans l'ordre. Chaque partie se trouve annotée dans le code de Jarvis afin de vous guider vers les endroits importants à comprendre et à modifier. **Ouvrez la classe JarvisBoot et lisez les commentaires au début pour retrouver votre chemin.** Chaque étape vaut sa proportion de points, y compris la dernière, qui est une section bonus. Les tests qui doivent fonctionner pour chaque partie sont mentionnés dans le texte qui suit, et se trouvent dans le répertoire /Tests du projet. Ce sont des bouts de code écrits en Jarvis, qui ne fonctionnent pas encore correctement, parce qu'il manque quelques facultés à votre langage. Seules quelques modifications mineures sont acceptables dans les fichiers de tests. Ils seront vérifiés pour s'assurer que vous ne changez pas l'intention des tests.

Les primitives du langage (20%)

Requiert une bonne reconnaissance dans le code, une compréhension de la construction des classes en Jarvis et de l'implémentation des fonctions opérant sur des objets encapsulant des atomes, plus un beau brin de copier-coller.

Pour opérer sur des types de base comme les entiers, on n'a pas le choix, il faut éventuellement exécuter du code Java qui fera le travail. Ce code est encapsulé dans les objets du package **primitives.integers**. Les objets s'y trouvant représentent des fonctions qui peuvent être appelées en Jarvis, mais qui ne s'exécutent pas dans un environnement Jarvis. Elles font un travail derrière les coulisses! Ces fonctions représentent ce que le langage Jarvis n'est pas capable de coder par lui-même. Ce sont donc des fonctions tricheuses! Vous devez implanter les fonctions tricheuses permettant de faire fonctionner les fichiers de code **testint.txt et testbool.txt**. Ce sont les opérations de base sur les entiers et les booléens qui ne sont pas encore codées. Il faut aussi modifier la définition des classes **int** et **bool**, se trouvant dans basictypes.txt, pour ajouter les nouvelles méthodes dans leur dictionnaire respectif.

La classe Rectangle (40%)

Demande d'arriver à comprendre du code Jarvis plus complexe, de coder une fonction avec une condition, de bien saisir comment fonctionne la file d'arguments et les environnements.

Le fichier **rectangle.txt** contient la définition d'une classe simple, un rectangle défini par quatre valeurs entières. Cette classe possède des méthodes qui calculent la hauteur, la largeur et le périmètre. Cependant, lorsque les points ne sont pas ordonnancés correctement, on se retrouve avec un périmètre incorrect! Ceci est dû au fait que la hauteur et la largeur ne sont pas calculées comme des valeurs absolues, avant qu'on calcule le périmètre.

Pour corriger ce problème, il faut écrire en Jarvis, dans le fichier Rectangle.txt, une fonction qui fait la valeur absolue d'un entier. Cette fonction doit faire une condition, que vous pouvez réaliser avec la classe **ifelse**, et en vous appuyant sur l'exemple se trouvant dans **ifelse.txt**. Une fois cette fonction définie, vous pouvez modifier les méthodes de la classe Rectangle pour que la hauteur et la largeur soient calculées correctement. ATTENTION!!! N'écrivez pas la fonction valeur absolue dans la fermeture qui sert à définir la classe Rectangle! Définissez-la plutôt dans util.txt comme une fonction indépendante du Rectangle.

L'héritage, première partie (65%)

Demande une compréhension plus poussée des mécanismes de Jarvis, ainsi que de la théorie sur le paradigme orienté-objet. Demande une bonne compréhension de la classe Class. Quelques coups de bistouri bien précis!

Jarvis ne supporte présentement pas l'héritage. Pour arriver à implanter cette capacité essentielle pour un langage orienté-objet, vous devez d'abord ajouter un membre de plus à une classe, sa **super-classe**. Ensuite, vous devez penser à l'algorithme qui est utilisé pour interpréter un message (ObjectAtom.message()). Présentement, un objet ne cherche que dans sa propre classe. Il faut maintenant chercher dans les classes parentes, jusqu'à la racine de l'arbre d'héritage. Au fait, pas d'héritage implique pas encore de racine à votre arbre... il faudrait donc implanter la classe **Object**, qui est la classe (en Jarvis) de tous les ObjectAtom!
Celle-ci peut être écrite en Jarvis, parce que le langage supporte déjà d'instancier des nouvelles classes!
Dans cette partie, vous ne devez supporter que l'héritage de **méthodes**, donc chercher seulement dans les méthodes des classes parentes, pas dans les attributs.
Le fichier **heritagemethodes.txt** contient les tests à faire fonctionner pour cette partie. Les classes codées avant cette modifications ne fonctionneront plus, il faudra donc les modifier en conséquence.

Le mutateur universel (75%)

Requiert une compréhension plus poussée de la notion d'environnement, des mécanismes d'appel de méthode, ainsi que de la représentation interne des ObjectAtoms et de la classe Class.

Présentement, il est impossible d'affecter une nouvelle valeur à un attribut d'un objet une fois qu'il est créé. Pour arriver à ajouter cette fonctionnalité très utile, il faut coder une fonction tricheuse, **OperatorSetPrimitive**. Cette fonction va permettre la modification d'un attribut via un appel de méthode en Jarvis, comme ceux qui se trouvent dans le fichier de test, **mutateur.txt**. Elle doit fonctionner rétroactivement pour toutes les classes.

L'héritage, deuxième partie (90%)

Demande une compréhension accrue des mécanismes d'instanciation et d'interprétation de message.

Finalement, l'héritage ne fonctionne correctement que si l'on hérite des attributs des classes parentes. Ceci a l'effet malencontreux de complexifier la création de nouveaux objets (**OperatorNewPrimitive**) ainsi que la recherche d'attributs dans un objet, lorsqu'on répond à un message (ObjectAtom.message()). **Note**: Il n'est pas essentiel que le mutateur fonctionne en conjonction avec l'héritage d'attributs pour l'instant. Les classes qui ont pour parent Object doivent encore fonctionner correctement avec le mutateur, par contre.

Le mutateur complet (95%)

Requiert une implémentation plus propre de l'héritage d'attributs et du mutateur. Un bon coup de balai dans le code en préparation à l'étape finale!

Si vous n'aviez pas implanté le support pour l'héritage avec le mutateur, vous devez maintenant corriger cette lacune. Le fichier, **mutateurcomplet.txt** contient un test qui démontre cette capacité.

Le constructeur (100%)

Excellent test sur l'implémentation complète de l'héritage.

Une fois le support complet pour l'héritage et le mutateur en place, vous pouvez finalement séparer instanciation et construction.
Pour l'instant, **OperatorNewPrimitive** requiert autant d'arguments qu'il y a de membres dans votre classe. Idéalement, il faudrait lever cette contrainte et permettre à cette opération de recevoir un nombre quelconque de paramètres qui seront passés à une fermeture jarvis (ClosureAtom) qui servira de constructeur. Le constructeur sera une méthode membre possèdera toujours le nom __init__ . Si la classe ne possède pas de constructeur, alors **OperatorNewPrimitive** va faire le même traitement qu'avant, c'est-à-dire, demander qu'on lui passe assez d'arguments pour initialiser tous les membres de la classe. La méthode __init__ peut prendre autant d'arguments qu'on le désire et faire autant d'opérations qu'on le désire sur les variables d'instance. Les constructeurs sont également hérités, et peuvent être appelés par les constructeurs des classes dérivées. Le fichier **constructeur.txt** contient un test qui démontrera le bon fonctionnement de votre constructeur.

Les variables de classe (105%)

Demande un brin de motivation et une très bonne compréhension du paradigme orienté-objet.

En bonus, vous pouvez tenter d'implanter le support pour les variables de classe. Vous devez fournir du code Jarvis qui démontre leur utilisation dans /Tests/variablesclasse.txt.

Les implémentations correctes peuvent prendre plusieurs formes, vous devrez décider de celle qui vous convient.

Remise

Vous devez soumettre votre travail à l'aide de la commande turnin, sur le serveur tarin.
Exportez votre projet complet sous forme d'archive, nommée **jarvis.zip** . Prenez soin d'inclure les noms et matricules des membres de l'équipe dans un fichier à part, **rapport.txt** (peut être dans un autre format). Ce rapport doit contenir une brève description de votre solution ainsi que des modification que vous avez apportées au code fourni pour y arriver. L'objectif de ce rapport est de faciliter le travail de correction. **L'absence de ce rapport entraîne une pénalité de 10%** sur la note totale.
Vous devez réaliser votre soumission sur le turnin web, à l'adresse suivante :

http://opus.dinf.usherbrooke.ca:8080/turnin_off

Annexe 1 : Liste de commandes

Les commandes de l'interpréteur Jarvis

`!ui` : Démarre l'interface graphique. Présentement en développement!

`!e` : Environnement. Affichage de l'environnement. La liste des symboles connus ainsi que leurs valeurs apparaîtra. Pour avoir plus de détails sur la valeur associée à un symbole, vous devez en forcer l'affichage en l'interprétant d'abord (entrez son nom), puis en affichant la valeur en tête de file (**!p**).

`!ee` : All environnement. Affichage de tous les niveaux d'environnement. Utile si vous avez mis un point d'arrêt dans une closure, ou si une erreur est survenue. Vous pourrez voir tous les symboles définis dans tous les niveaux d'environnement. Également utile pour vérifier la présence de fonctions tricheuses.

`!p` : Print. Affichage de l'atome qui se trouve en tête de la file d'arguments.

`!c` : Clear. Vide la file d'arguments. Utile lorsque vous dialoguez directement avec l'interpréteur.

`!args` : Arguments. Affiche le contenu de la file d'arguments. Utile pour voir si la file est dans l'état prévu à un moment donné l'exécution de votre code. Outil de déverminage.

`!a` : Interpret argument. Interprète l'argument qui se trouve en tête de file. Si c'est un atome autre qu'une fermeture, il sera remis en arrière de la file. Utilisé principalement pour remplacer le destinateur et le sélecteur d'un message par des éléments de la file, ou pour exécuter une fermeture qu'on vient de définir sans lui associer de symbole.

`!eoc` : End of context. Détruit l'environnement actuel et remonte à l'environnement parent. Cette commande est faite automatiquement à la fin de l'évaluation d'une fermeture. Si vous la faites manuellement dans l'environnement d'interprétation, vous pourrez voir l'environnement racine, qui contient les définitions des fonctions tricheuses, telles qu'établies dans la méthode JarvisInterpreter.addCheaterCode(). Pratique pour déverminer les fonctions tricheuses.

`!load` : Load file. Interprète un fichier de code Jarvis au complet. Utile pour charger des bibliothèques. Attention, durant l'exécution de code via !load, la commande !debug ne fonctionne pas. Pour déverminer une bibliothèque, telle que basictypes.txt, vous feriez mieux de la charger au démarrage (argument à l'exécution de votre projet dans eclipse).

`!debug` : Debug. Cesse la lecture du fichier et prend les entrées du clavier. Attention! Cette commande ne fonctionne pas bien à l'intérieur d'une fermeture! Elle est utile pour mettre un point d'arrêt dans votre code Jarvis, et vous permettre de diagnostiquer l'état de l'interpréteur à ce moment-là à l'aide des commandes !e, !args, !p.

`!mute et !unmute :` Active et désactive la sourdine. Par défaut, lors de l,évaluation d'une fermeture, l'interpréteur affiche chaque commande qu'il interprète. Si vous trouvez que ça pollue l'affichage, activez la sourdine. Pour déverminer, il peut être utile de la désactiver et de voir si votre code s'est exécuté comme prévu.

`!ref :` Reference. Définit un nouveau symbole. Voir les exemples plus haut. Ce symbole est ajouté dans l'environnement courant.

`!( :` Mesage. Envoie un message à un objet. Les entrées suivantes doivent être: un objet (ou !a pour prendre l'objet de la file d'arguments), un sélecteur (nom d'attribut ou de méthode, ou !a pour le prendre de la file d'arguments), puis une liste d'arguments terminée par une parenthèse fermante. Cette liste d'arguments, PLUS ceux qui se trouvent dans la file, seront TOUS envoyés à la méthode choisie, si c'est une méthode.

`!{ :` Closure. Définit une fermeture. Les prochaines entrées doivent être une suite de symboles qui représentent les noms des paramètres, terminée par un point (**.**). Après, il faut entrer toutes les commandes que la fermeture fera. Toute commande acceptable par l'interpréteur est valide dans une fermeture, sauf !debug, qui ne fonctionne pas correctement. La suite de commande se termine par une accolade fermante (**}**).

Annexe 2 : conseils pratiques et idiosyncraties du langage

1- N'essayez pas d'entrer des nombres négatifs dans l'interpréteur (ou un fichier qu'il lira). Un JarvisInt peut être négatif, mais l'interpréteur n'arrive pas à reconnaître les nombres qui commencent par le - .

2-N'écrivez pas de chaînes de caractères avec des espaces. L'interpréteur ne reconnaît pas correctement les guillemets, vous ne devriez donc JAMAIS mettre des caractères spéciaux dedans, comme : {, }, [,], (,). L'exception est le point (.), vous pouvez faire des chaînes de caractères contenant des points (utile pour les noms de fichiers).

3-Lorsque vous définissez une fermeture (fonction en Jarvis), le point qui termine la liste de paramètres doit être séparé du dernier paramètre par un espace, sauf s'il n'y a pas de paramètre.

Exemple:

```
!{ a .
 a
 !p
}
```

Correct. Le symbole et le point sont séparés d'un espace.

```
!{ a.
 a
 !p
}
```

Incorrect. Le symbole et le point son collés

```
!{.
 !e
}
```

Correct. Il n'y a pas de paramètre.

4-La commande !debug ne fonctionnera pas lorsqu'elle se trouve dans un fichier qui est exécuté avec la commande !load.

5-Utilisez le plus possible les guillemets lorsque vous écrivez le sélecteur lorsque vous envoyez un message à un objet. Si un symbole dans l'environnement a le même nom que le membre ou la méthode mis en sélecteur, la valeur associée au symbole sera envoyée comme sélecteur!!!
Exemple :

```
>!ref value "asdf"
>!(int new 6)
>!ref x
>!(x value)
```

```
>!p
ComprendPas jarvis.atoms.StringAtom@2a139a55
>
```

Ici, le sélecteur qui a été envoyé est le StringAtom « asdf ». Ce sélecteur ne correspond à aucune méthode ou aucun membre de la classe int, la StringAtom « ComprendPas » est donc retournée par ObjectAtom.message(). Ce type de conflit est particulièrement frustrant à débugger. Choisissez donc des noms de membres significatifs lorsque vous écrivez des classes de tests. Évitez de réutiliser sans cesse les mêmes noms de variables. Finalement, si vous mettez des guillemets pour le sélecteur, ce problème ne surviendra pas (c'est cependant un peu fastidieux).