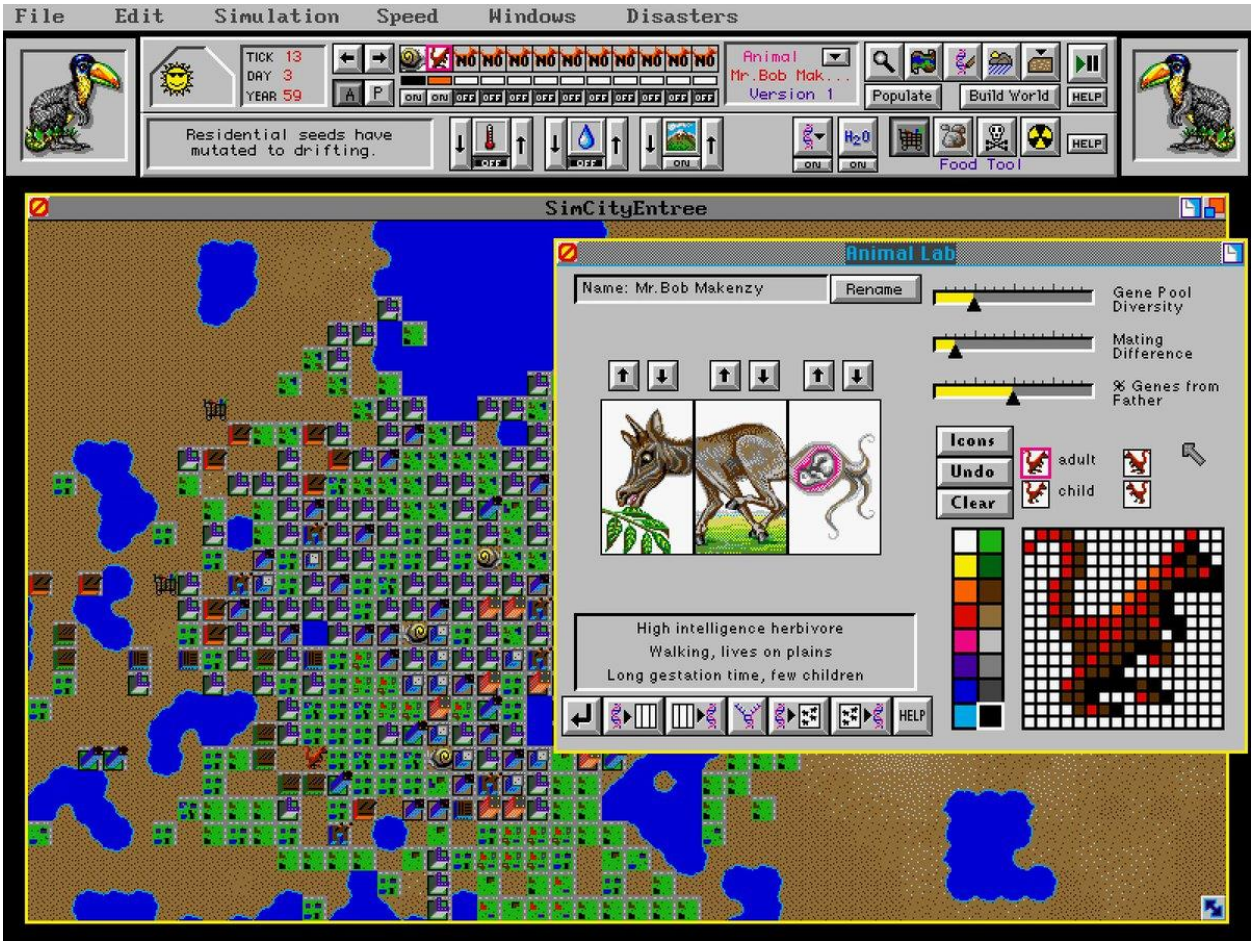
Travail pratique I

Le lac

Les écosystèmes atteignent un équilibre fragile malgré le nombre élevé et la diversité des organismes qui s'y trouvent. L'apparition des ordinateurs permet aux biologistes, bio-informaticiennes et bio-informaticiens de tester des théories par simulation. Ce travail pratique n'a pas la prétention d'être aussi sérieux. Il est inspiré de *SimLife*, développé par le studio Maxis en 1992. Ce logiciel est décrit comme un jouet numérique plutôt qu'un jeu vidéo. On peut gagner ou du moins terminer un jeu vidéo, mais comment gagner une simulation?



But du travail pratique

Vous devez créer un simulateur d'écosystème. Vous utiliserez les principes de programmation orientée objet pour modéliser les différents organismes. Pour simplifier le problème, nous nous limiterons à suivre la distribution de l'énergie. De plus, nous modéliserons un lac et nous supposerons que la seule interaction avec le monde extérieur est l'arrivée de la lumière du soleil.

Partie 0 : préparation

Lecture

La première étape, la plus importante, est de lire et comprendre ce document en entier.

Structure

Ce travail pratique est divisé en trois parties. Chaque partie est une extension de la précédente : vous devez terminer la partie I avant de faire la II et la II avant la III. Chaque partie ne dépend pas de la suivante, c'est-à-dire que, s'il vous manque de temps, vous pouvez par exemple arrêter avant la partie III et avoir tous les points pour les deux premières parties.

Fichiers

Vous trouverez sur le OneDrive du cours des fichiers Java pour vous aider. Ne les copiez pas tous de suite dans votre projet : les instructions dans chaque partie vous diront quand et comment le faire.

Il y a aussi un dossier **`exemples`**. Il contient des fichiers d'entrées à votre programme de simulation. Certains servent à expliquer des éléments à implanter, certains sont seulement des exemples pour tester votre programme. Assurez-vous que votre implantation accepte tous ces exemples : ce sont vos tests d'intégration.

Partie I : les plantes (40 %)

Les premiers organismes à pouvoir faire de la photosynthèse sont apparus il y a environ un milliard d'années, soit plus de 300 millions d'années avant les premiers animaux. La première partie de ce travail pratique est de simuler un environnement avec seulement des organismes se nourrissant de la lumière du soleil.

Code de départ

Le dossier **`plantes`** contient les fichiers suivants.

- **`Simulation.java`** : c'est le fichier qui contient la fonction **`main`**, qui lance la simulation.
- **`Lac.java`** : modélise l'environnement dans lequel vivent nos organismes.
- **`ConditionsInitiales.java`** : gère la lecture de fichier de configuration.
- **`ConditionsInitialesInvalides.java`** : une exception qui signale que le fichier en entrée est invalide.

Lisez ces fichiers. Il n'est pas nécessaire de tout comprendre, mais ayez une idée de leur fonctionnement parce que vous devrez les modifier plus tard. Notez qu'ils ne compilent pas encore : c'est ce que vous corrigerez à la première étape.

Création d'une plante (20/40)

Si vous essayez de compiler les sources Java, vous obtiendrez une erreur. Le fichier qui charge les conditions initiales essaie de créer une plante par un objet-usine, mais ni la plante ni l'usine n'existent.

L'usine (*factory*) est un patron de conception commun en Java. Une méthode usine est une méthode dont la responsabilité est de créer une instance. Une classe usine est une classe utilisée principalement pour ses méthodes usine. On utilise de telles méthodes et classes dans les cas suivants.

- On veut donner des noms aux constructeurs. Par exemple, au lieu d'écrire `new Colis()` dans une application de poste, on pourrait écrire `enregisterEtPosterColis()`.
- La nouvelle instance est « contenue » dans la classe qui offre la méthode usine. Par exemple, si on a un magasin en ligne, les produits sont « dans » ce magasin. On peut utiliser une méthode usine pour créer un produit : `magasin.creerProduit(nom, prix)`. Le magasin pourra calculer les attributs comme les taxes et la politique de retour sans qu'on ait à les spécifier dans l'appel à `creerProduit`.
- On veut configurer incrémentalement une instance, mais une fois créée, elle doit être immuable. La classe finale n'aura pas de *setters*, mais la classe usine oui. C'est pour cette raison qu'on utilise l'usine dans ce travail pratique.

Créez une classe `Plante` avec les attributs suivants. Leur signification est expliquée dans la section suivante.

Type	Nom	Lecture?	Écriture?	Contraintes
`String`	`nomEspece`	✓	✗	non-vide
`double`	`energie`	✓	✗	≥ 0
`int`	`age`	✓	✗	≥ 0
`double`	`besoinEnergie`	✓	✗	> 0
`double`	`efficaciteEnergie`	✓	✗	$\in [0, 1]$
`double`	`resilience`	✓	✗	$\in [0, 1]$
`double`	`fertilite`	✓	✗	$\in [0, 1]$
`int`	`ageFertilite`	✓	✗	≥ 0
`double`	`energieEnfant`	✓	✗	> 0

Créez une classe `UsinePlante`. Elle a les mêmes attributs que Plante, sauf pour `age` et `energie`, mais ses attributs sont en écriture seulement. Ajouter à cette classe une méthode `creerPlante` qui crée une instance de plante. Assignez `0` à l'`age` de la plante nouvelle-née et `energieEnfant` à `energie`.

Tous les autres attributs sont obligatoires. Si un ou plusieurs attributs n'ont pas été spécifié avant l'appel à `creerPlante`, vous devez lancer une exception avec de créer l'instance. **Suggérez dans votre `README.txt` deux façons de garder une trace de quels attributs ont été initialisés. Laquelle avez-vous choisie et pourquoi ?**

Le code de l'étape I doit compiler si cette étape a été faite correctement.

Simulation de la vie d'une plante (20/40)

On divise la simulation en tranches de temps discrètes que nous appellerons *cycles*. À chaque cycle, les plantes absorberont la lumière du soleil. La classe `Lac` contient une méthode `tick` qui ne fait rien pour l'instant. L'objectif de cette étape est d'implanter cette méthode selon les règles suivantes.

- L'énergie E du soleil est distribuée aux plantes proportionnellement à leur taille. Nous supposerons que leur taille est proportionnelle à leur `energie`. S'il y a n plantes, l'énergie E_e absorbée par la plante x est donc de $E_e = E \cdot E_x / \sum_0^n E_i$.
- Chaque plante a besoin de `besoinEnergie` à chaque cycle pour survivre.
- Pour chaque unité d'énergie manquante, la plante a la probabilité `resilience` de survivre. Par exemple, si une plante reçoit 8 unités d'énergie, mais que `besoinEnergie` est à 10 et `resilience` est à 0.98. Elle a $0.98 \cdot 0.98 = 0.9604$ chance de survivre.
- Si elle survit, enlever la quantité d'énergie manquante à son `energie`.
- Pour chaque unité d'énergie supplémentaire, si l'`age` de la plante est supérieur ou égal à `ageFertilite`, il y a une probabilité `fertilite` que la plante fasse un enfant. L'enfant nait avec `energieEnfant` unités d'énergie. Cette énergie n'est pas créée, mais transférée. Elle est prise d'abord de l'énergie supplémentaire et ensuite de l'énergie de la plante parent elle-même. S'il reste de l'énergie supplémentaire, on continue à rouler.
  Par exemple, si on a 5 unités d'énergie supplémentaire, `fertilite` est à 0.02 et `energieEnfant` est à 3. On roule d'abord 0.00, ce qui donne un enfant qui nait avec 3 d'énergie. Parmi ces trois énergies, on a déjà roulé pour la première. On n'a pas roulé pour les deux autres, mais on ne roulera jamais parce qu'elles ont été consommées par l'enfant.
  Il reste 2 d'énergie supplémentaire, on roule maintenant 0.44 qui ne donne pas d'enfants.
  Il reste une unité, on roule encore et on a 0.01. On prend les deux unités qu'il reste et une unité de la plante parent pour faire un deuxième enfant.
- S'il reste encore de l'énergie, on l'ajoute à l'organisme à un taux `efficaciteEnergie`.

Indice : utilisez `Math.random()` pour générer un nombre (pseudo-)aléatoire entre 0 et 1.

Vous pouvez ajouter des méthodes à la classe `Plante` pour implanter ces règles. **Expliquer dans votre README.txt comment les responsabilités entre les classes Lac et Plante ont été divisées et pourquoi. Quels changements avez-vous dû faire?**

Après cette étape, vous devez pouvoir exécuter la simulation `plantes.xml`.

```
javac -d out src/*.java
java --class-path out Simulation < exemples/plantes.xml
```

Partie II : les herbivores (30 %)

Les premiers animaux sont apparus il y a plus de 600 millions d'années, mais c'est lors du cambrien (-544 à -510 millions d'années) que les groupes que nous connaissons aujourd'hui, comme les mollusques, apparaissent.

Fichiers de départ

Il y a une nouvelle version du fichier `ConditionInitiales.java` dans le dossier `herbivores`. Remplacez la copie du fichier qui vient de l'étape précédente par cette version.

Création d'un herbivore (15/30)

Comme pour l'étape *création d'une plante*, le code qui charge les conditions initiales s'attend à ce qu'une classe-usine existe. Créer cette classe-usine, `UsineHerbivore`, et la classe `Herbivore`. Les attributs sont les mêmes que pour une plante, avec ceux-ci en plus.

Type	Nom	Lecture?	Écriture?	Contraintes
`double`	`debrouillardise`	✔	✘	$\in [0,1]$
`double`	`voraciteMin`	✔	✘	$\in [0,1]$
`double`	`voraciteMax`	✔	✘	$\in [0,1]$
`Set<String>`	`aliments`	✔	✘	non-null

Il y doit aussi avoir la contrainte supplémentaire que `voraciteMin` doit être inférieur ou égal à `voraciteMax`.

Il y a beaucoup de répétition entre les classes `Plante` et `Herbivore`. Créer une classe ou une interface nommée Organisme. Modifier nos deux classes pour hériter de la classe Organisme ou d'implanter l'interface Organisme selon ce que vous avez choisi. **Expliquer votre choix dans le fichier** `README.txt`**.** Il y a aussi de la répétition entre les deux classes-usines. **Expliquez aussi dans le** `README.txt` **comment l'éliminer et faites les changements proposés.**

Il ne doit plus avoir d'erreurs dans votre code à la fin de cette étape.

Simulation de la vie d'un herbivore (15/30)

Il faut ajouter nos herbivores au `Lac`. Inspirez-vous du code déjà présent pour les plantes pour aussi avoir une `List` d'herbivores. Ajouter à la méthode `tick` du code pour simuler la vie d'un herbivore selon les principes suivants.

- Un herbivore a la probabilité `debrouillardise` de se nourrir. Cet attribut est une abstraction de l'intelligence, la mobilité et n'importe quel autre attribut de l'animal qui l'aide à se nourrir. Roulez jusqu'à obtenir un négatif et le nombre de positifs est le nombre de fois où l'animal se nourrira. Par exemple, pour une `debrouillardise` de 0.5, on roule un 0.2, un 0.1, puis un 0.6. On arrête à ce moment et on note que l'animal se nourrira deux fois.
- Pour chaque fois, choisissez au hasard une plante dont l'espèce apparait parmi les `aliments` de l'animal. Calculez une quantité d'énergie qui est une fraction entre

`voraciteMin` et `voraciteMax` de l'énergie totale de la plante. Par exemple, un animal ayant une voracité entre 0.4 et 0.6 qui mange une plante qui contient 100 unités d'énergie va absorber entre 40 et 60 unités d'énergie.

- Les règles pour les déficits et surplus d'énergie sont les mêmes que pour les plantes.

Expliquez dans votre README.txt comment vous avez séparé les responsabilités entre Lac et Herbivore. Avez-vous changé de stratégie par rapport à la première partie?

Après cette étape, vous devez pouvoir exécuter la simulation `herbivores.xml`.

```
javac -d out src/*.java
java --class-path out Simulation < exemples/ herbivores.xml
```

Partie III : les carnivores (20 %)

Certains carnivores modernes comme les chats ne font pas que préférer la viande, mais doivent en consommer parce que leur corps ne peut pas synthétiser plusieurs molécules essentielles.

Fichiers de départ

Il n'y a pas de nouveau fichier pour cette partie.

Création d'un carnivore (10/20)

Créez une classe `Carnivore` qui a les mêmes attributs qu'`Herbivore`, sauf pour les deux `double`s qui modélisent la voracité. Créez aussi une usine.

Cette fois, c'est vous qui devez modifier la classe `ConditionsInitiales` pour supporter les carnivores. Inspirez-vous du code pour la création d'herbivores. **Avez-vous réutilisé du code ? Si oui, expliquer comment et pourquoi c'est désirable dans votre README.txt. Si non, expliquez pourquoi c'est indésirable.**

Simulation de la vie d'un carnivore (10/20)

Un carnivore a le même comportement qu'un herbivore à la différence qu'il mange sa proie en entier. On peut croquer une branche d'une plante et s'attendre à ce qu'elle survive, mais ce n'est pas le cas d'un autre animal ! Ajouter ce comportement au Lac selon les mêmes principes que pour les autres organismes.

Après cette étape, vous devez pouvoir exécuter la simulation `carnivores.xml`.

```
javac -d out src/*.java
java --class-path out Simulation < exemples/ carnivores.xml
```

Notre simulation est loin d'être réaliste, mais vous y ajouterez une dernière chose : les petits poissons ne peuvent pas manger les grands. Ajouter un attribut `tailleMaximum` aux herbivores et carnivores. Modifier `ConditionsInitiales` pour lire cet attribut s'il est présent, sinon avoir une valeur par défaut qui est de 10 fois l'attribut `energieEnfant`. Cela signifie par exemple qu'un adulte est 10 fois plus gros qu'un enfant.

Modifier le comportement des carnivores pour ignorer les organismes plus gros lorsqu'ils se nourrissent.

Après cette étape, vous devez pouvoir exécuter la simulation `carnivores-taille.xml`.

```
javac -d out src/*.java
java --class-path out Simulation < exemples/carnivores-taille.xml
```

Remise

Vous devez remettre une archive de format `zip` ou `tar.gz` avec le contenu suivant, qui doit être placé à la racine de l'archive et non dans un dossier.

- Un fichier nommé `README.txt`, qui contient les réponses aux questions.
- Un dossier nommé `src`, qui contient tous les fichiers Java.

Votre code Java doit compiler avec la version 15 d'OpenJDK sans erreur avec la commande suivante.

```
javac -d out src/*.java
```

On doit ensuite pouvoir lancer une simulation avec la commande suivante.

```
java --class-path out Simulation
```

Évaluation

Les trois parties valent respectivement 40 %, 30 % et 20 % de la note finale. Vous pouvez consulter les sections expliquant chaque partie pour les détails. Les 10 % restants sont distribués comme ceci.

- Respect des consignes (5 %) : en particulier, vous aurez 0 % pour cette section si je dois modifier votre code pour qu'il compile.
- Qualité du français et du Java (±5 %) : vous perdrez des points pour une quantité déraisonnable de fautes d'orthographe ou si votre code est systématiquement mal indenté par exemple. Vous pouvez cependant gagner jusqu'à 5 points si votre texte et votre code sont impeccables !

Remarques et conseils

- Vous pouvez utiliser toute la puissance du JDK. N'hésitez pas à importer vos classes préférées ! Les autres bibliothèques sont interdites.
- Le volume de code pour ce travail est assez volumineux. Je recommande d'utiliser un IDE qui peut synthétiser ce qui est répétitif comme les accesseurs. Aussi, moins de code n'est pas toujours meilleur si ça signifie qu'il est plus compliqué.

- J’ai surement sous-spécifié certaines parties du problème. Par exemple, est-ce qu’un carnivore peut manger des membres de sa propre espèce ? Si vous rencontrez une telle situation, vous pouvez choisir de l’implanter comme vous voulez.