Exercises

W07H03 - Pingu Sim

Points	Submission due date	Status	Difficulty	Categories
0 / 10	in 5 days	No graded result		Hausaufgabe

Tasks:

Pingu Sim

Den Lehruinen ist aufgefallen, dass die teils sehr komplexen Zusammenhänge einzelner Populationen in einem gemeinsamen Lebensraum bisher nur rein theoretisch besprochen wurden. Dabei wäre ein einfaches Interaktives Beispiel sehr viel einfacher. Deshalb haben sie die Gameunine beauftragt, ein kleines Spiel zu entwickeln. Während einer der Gameuine an der graphischen Oberfläche arbeitet, sollst du schon mal die Logik implementieren.

In dieser Aufgabe sollst du ein Programm schreiben, das die Entwicklung von Populationen verschiedener Lebewesen (genauer: Pflanzen, Hamstern, Pinguinen und Wölfen) sehr vereinfacht simuliert. Dafür werden die Lebewesen mit Hilfe von Interfaces und Vererbung modelliert.

1. Allgemeines

1. Alle Lebewesen nehmen eine Zelle auf dem Spielfeld ein. Es gibt, wie gesagt, Pflanzen, Hamster, Pinguine und Wölfe.
2. Das Spielfeld ist als eindimensionales Array gespeichert. Der Ursprung ist oben links.
3. Alle geforderten Methoden und Attribute sind auch wie gefordert zu implementieren und werden auch einzeln bewertet.
4. `width` und `height` sind immer > 0. Außerdem haben die den Methoden `tick()`, `place()`, `eat()` und `move()` mitgegebenen Arrays immer genau die Größe $\mathrm{width} \cdot \mathrm{height}$ und die übergebenen Parameter `x` und `y` dieser Methoden beschreiben immer valide Positionen im Array.
5. `null` repräsentiert eine leere `Cell` im Spielfeld. Dementsprechend kann `null` als `Cell` Parameter gegeben werden und soll entsprechend behandelt werden. Insbesondere ist eine leere Zelle beispielsweise kein `Pingu`.
6. Wenn gefordert ist, die Zugehörigkeit zu einer gewissen Klasse zu prüfen, soll das nicht auf der Implementierung dieser Klasse oder der Existenz anderer Basieren. Zum Beispiel sollen dafür nicht Instanzen von `CellSymbol` verglichen werden. Sie soll auch funktionieren, wenn zu einem späteren Zeitpunkt weitere `Cell` Untertypen hinzugefügt werden, obwohl diese nicht Teil der Aufgabe sind.
7. Es ist immer genau festgeschrieben, wann `RandomGenerator` benutzt werden soll, und wie das Ergebnis Interpretiert werden soll. Daher soll mit gleichem Seed auch immer das Ergebnis gleich sein.
8. Abstrakte Methoden werden mit einem ~ davor markiert.
9. In dieser Aufgabe musst du die meisten **Klassen und Methoden selbst erstellen**. Achte darauf, dass du die Namen, Parameter, Rückgabetypen, und Vererbungen aller Klassen und Methoden richtig definierst!
10. Du kannst deine Methoden testen, indem du die `main`-Methode in der Klasse `Main` ausführst. Diese Aufgabe startet eine UI, die die Simulation ausführt und euch sogar erlaubt, die Parameter der Simulation zur Runtime zu ändern.

2. Übersicht

Deine Aufgabe ist es, dieses Spiel zu vervollständigen. Dazu hier eine kurze Übersicht über die einzelnen Komponenten des Spiels.

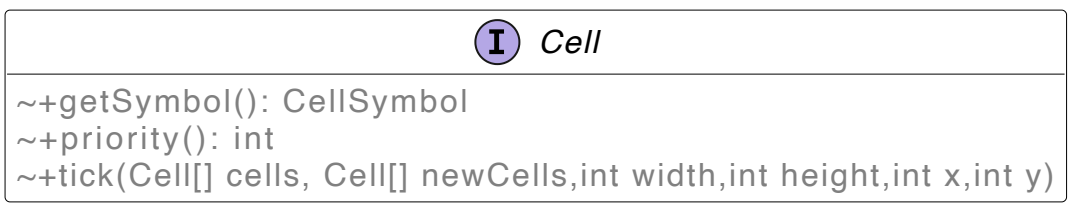
1. `SimConfig` enthält einige globale Variablen, mit denen einzelnen Aspekte des Spiels eingestellt werden können. Hier musst und sollst du nichts ändern.
2. `Simulation` enthält das Spielfeld als ein eindimensionales Array. Die `tick()` Methode führt Operationen auf allen `Cell`s auf dem Spielfeld aus. Der Parameter `cells` von `tick()` repräsentiert dabei das alte Feld, der Parameter `newCells` das neue.
3. `Cell` wird von allen Zellen implementiert und definiert gemeinsame Operationen. Außerdem ist die `place()` Methode schon implementiert. Sie wird benutzt, um neue Zellen auf dem Spielfeld zu platzieren und kann natürlich als Inspiration benutzt werden. Wenn `place()` auf einem Objekt, dessen Typ `Cell` implementiert, aufgerufen wird, wird dieses zufällig auf dem übergebenen Spielfeld `newCells` platziert. Wenn dabei eine Zelle zum Setzen gewählt wird, die schon belegt ist, wird das Objekt nicht platziert und `false` zurückgegeben. Wenn eine Zelle gewählt wird, die noch frei ist, wird das Objekt in diese platziert und `true` zurückgegeben. An `Cell` musst du nichts mehr ändern.
4. `Plant` ist eine Zelle die sich nicht bewegen kann aber dafür passiv Nahrung generiert. Sie wird von Hamstern und Pinguinen als Nahrung benutzt.
5. `MovingCell` ist eine Oberklasse, die gemeinsames Verhalten der sich bewegenden Zellen implementiert. Dazu gehört das Essen anderer Zellen, Bewegung, Vermehrung und anders als bei `Plant` eine kontinuierliche Abnahme der *"gespeicherten"* Nahrung. Sie können also anders als Pflanzen verhungern.
6. `Hamster` sind effizienter im Essen von `Plant`, dafür können sie von `Wolf` gegessen werden.
7. `Pingu` sind weniger effizient im Essen von `Plant`, würden also von `Hamster` verdrängt werden. Dafür sind sie nicht auf dem Speiseplan eines `Wolf`.
8. `Wolf` können keine `Plant` Zellen essen, dafür `Hamster`.
9. `CellSymbol` wird benutzt, um das Symbol zu definieren, das für die Zelle gerendert werden soll. Auch an dieser Klasse musst du nichts ändern.
10. `RandomGenerator` gibt eine Zufallszahl zurück. Bei gleichem Seed und gleicher Reihenfolge der Aufrufe sind auch die zurückgegebenen Zufallszahlen identisch. Auch an dieser Klasse musst du nichts ändern.

3. Das Interface `Cell`

Wir deklarieren nun ein paar Prototypen, die wir später in den Unterklassen implementieren.

1. `~getSymbol(): CellSymbol` gibt das Symbol für diese Zelle an.
2. `~priority(): int` wird beim Erstellen neuer Zellen benötigt.
3. `~tick(Cell[] cells, Cell[] newCells,int width,int height,int x,int y)` wird in jeder Runde der Simulation aufgerufen. Hier wird in den Unterklassen implementiert, wie diese sich jeweils in einem Tick verhalten (Vermehrung, Bewegung, Wachstum etc.).

Das Interface `Cell` ist bereits vollständig implementiert. Hier musst du nichts mehr tun.



4. **Plant** No results

`Plant` implementiert `Cell`.

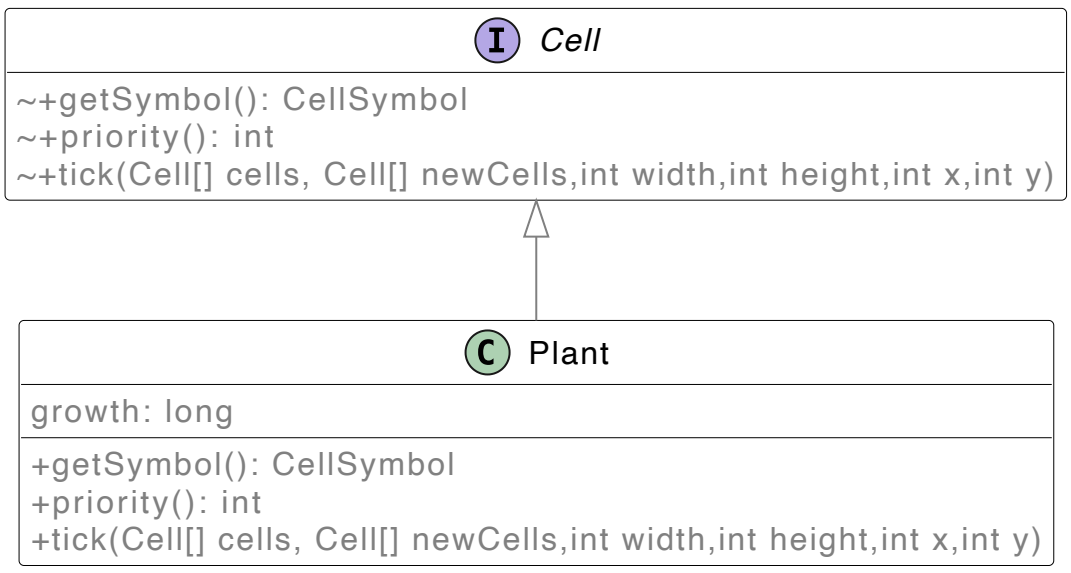
1. **growth:long** No results
   Dieses Feld speichert den aktuellen Wachstumswert dieser Zelle. Sie wird beim Erstellen einer neuen `Plant` auf 0 initialisiert.

2. **getSymbol(): CellSymbol** No results
   Diese Methode gibt lediglich `CellSymbol.PLANT` zurück.

3. **priority(): int** No results
   Diese Methode gibt lediglich `0` zurück.

4. **tick(...)** No results
   Diese Methode führt die folgenden Schritte aus:

   1. Setze die mit `x` und `y` beschriebene Position in `newCells` auf `this` (eine `Plant` bewegt sich nicht).
   2. Addiere einen Integer zwischen `SimConfig.plantMinGrowth` und `SimConfig.plantMaxGrowth` (exklusiv) zu `growth`. Nutze dafür `RandomGenerator`. (Das simuliert ein zufälliges Wachstum der `Plant`.)
   3. Wenn danach `growth`$\geq$`SimConfig.plantReproductionCost` gilt, platziere eine neue `Plant` Zelle in `newCells` mit `Cell.place()`. Falls das erfolgreich war (Rückgabewert von `place()` war `true`), ziehe die `plantReproductionCost` von `growth` ab. Wiederhole Schritt 3 solange, bis entweder `growth` nicht groß genug ist, um weiter zu machen oder `place()` einmal nicht erfolgreich war.



5. **MovingCell** No results

`MovingCell` ist eine abstrakte Klasse, die `Cell` implementiert. Sie ist Oberklasse aller `Cell`s, die sich bewegen können, also für uns hier von `Hamster`, `Pingu` und `Wolf`.

1. **food: long** No results
   Dieses Feld speichert die aktuelle Nahrung dieser Zelle. Sie wird auf den Rückgabewert von `initialFood()` initialisiert.

Die Methoden 2. bis 7. sind abstrakt und werden erst in den Unterklassen implementiert.

2. **~canEat(Cell other): boolean** No results
   Diese abstrakte Methode sagt aus, ob die übergebene Zelle von der Zelle, auf der die Methode aufgerufen wird, gegessen werden kann.

3. **~foodConsumption(): int** No results
   Diese abstrakte Methode gibt die Menge an Nahrung zurück, die diese Zelle bei jedem Aufruf von `tick` verbraucht.

4. **~consumedFood(): int** No results
   Diese abstrakte Methode gibt die Menge an Nahrung zurück, die diese Zelle bei dem Verspeisen einer anderen Zelle bekommt.

5. **~reproductionCost(): int** No results
   Diese abstrakte Methode gibt die Menge an Nahrung zurück, die diese Zelle für die Vermehrung benötigt.

6. **~initialFood(): int** No results
   Diese abstrakte Methode gibt die Menge an Nahrung zurück, die jede Zelle diesen Types hat, direkt nach dem sie sich vermehrt hat.

7. **~getNew(): Cell** No results
   Diese abstrakte Methode gibt eine neue Instanz der nicht abstrakten Subklasse zurück.

Die Methoden 8. bis 11. erhalten bereits in `MovingCell` eine Implementierung.

8. **move(Cell cells, Cell newCells,int width, int height, int x, int y)** No results
   Diese Methode bewegt diese `Cell`. Dabei werden die folgenden Schritte ausgeführt:

   1. Wähle ein zufälliges Feld im 3 mal 3 Feld mit der Zelle im Mittelpunkt. Nutze dafür *eine* Zahl aus dem `RandomGenerator` Die Felder sind dabei wie folgt durchnummeriert.

0	1	2
3	4	5
6	7	8

   2. Wenn dieses Feld innerhalb des Spielfelds ist und die Position in `cells` *und* `newCells` frei ist, wird in `newCells` an dieser Position diese Zelle eingetragen und die bisherige Position in `cells` als frei markiert. Falls nicht, wird die Zelle an der bisherigen Position in `newCells` gespeichert (in anderen Worten: dann konnte sich die `Cell` nicht bewegen).

9. **eat(...)** No results
   Diese Methode isst umliegende Zellen. Dabei wird wie folgt vorgegangen: Für jede Zelle innerhalb des 3 mal 3 Quadrats um diese Zelle innerhalb der Grenzen des Spielfelds:

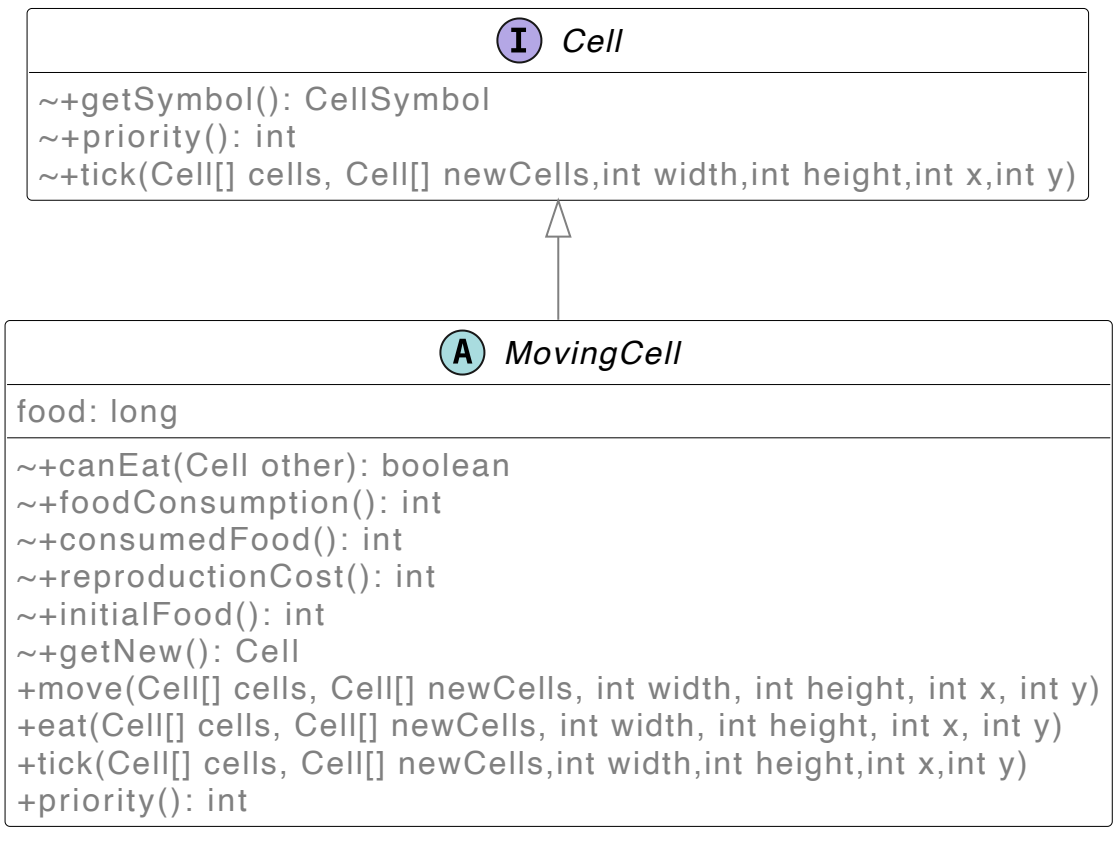
   1. Überprüfe mit `canEat` ob die Zelle an dieser Position in `cells` gegessen werden kann. Falls ja:
   2. Setze diese Position der gegessenen `Cell` in `newCells` auf `null`
   3. Addiere den Rückgabewert von `consumedFood()` zu `food`

Diese 2 `Cell` Arrays `cells` und `newCells` sind notwendig, um nicht Zellen, die weiter oben oder links sind, zu bevorzugen.

10. **tick(...)** No results
    Diese Methode implementiert alle Aktionen von sich bewegenden Zellen außer Essen:

    1. Wenn `food`$\geq$`reproductiponCost()` gilt, wird mit `getNew()` eine neue Zelle erstellt und mit `Cell.place()` platziert. `place()` bekommt dafür die Position der Elternzelle übergeben.
    2. Wenn die neue Zelle erfolgreich auf dem Spielfeld platziert wurde, wird `food` auf `initialFood()` gesetzt, sonst geschieht nichts.
    3. `foodConsumption()` wird von `food` abgezogen (das simuliert die Basis-Food-Kosten einer `MovingCell` unabhängig von Vermehrung).
    4. Wenn `food` nicht negativ ist, wird `move()` aufgerufen.

11. **priority(): int** No results
    Gibt 1 zurück.



6.

⦿ **Hamster, Pingu & Wolf** No results

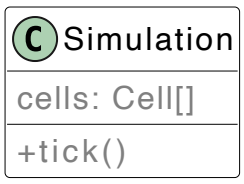
Diese Klassen sind sich sehr ähnlich, daher werden sie zusammen erklärt. Hier werden nun die abstrakten Methoden aus Unterpunkten 2. bis 7. von `MovingCell` für die drei Unterklassen individuell implementiert.

1. ⦿ **getSymbol(): CellSymbol** No results
   Hier wird die entsprechende Konstante aus `CellSymbol` zurückgegeben. Für `Hamster` wäre das beispielsweise `CellSymbol.HAMSTER`.
2. ⦿ **canEat(Cell other): boolean** No results
   Gibt zurück, ob diese Zelle gegessen werden kann. `Hamster` und `Pingu` essen `Plant` und `Wolf` isst `Hamster`.
3. ⦿ **foodConsumption(): int** No results
   Gibt die entsprechende Variable aus `SimConfig` aus.
4. ⦿ **consumedFood(): int** No results
   Gibt die entsprechende Variable aus `SimConfig` aus.
5. ⦿ **reproductionCost(): int** No results
   Gibt die entsprechende Variable aus `SimConfig` aus.
6. ⦿ **initialFood(): int** No results
   Gibt die entsprechende Variable aus `SimConfig` aus.
7. ⦿ **getNew(): Cell** No results
   Erstellt eine neue Instanz dieser Klasse und gibt sie zurück.

7.

⦿ **Simulation** No results

1. ⦿ **tick()** No results
   Diese Methode wird für jeden Schritt der Simulation einmal aufgerufen. Dabei führt sie die folgenden Schritte aus:
   1. Erstelle eine Kopie (z.B. `copyOfCells`) von `cells`.
   2. Rufe auf jeder Instanz von `MovingCell` in `cells` `eat()` auf. Dabei ist die Kopie `copyOfCells` der Parameter `newCells`.
   3. Nun ist in `copyOfCells` das aktuelle Spielfeld enthalten, nachdem alle `Cell`s gegessen haben. Fülle also das ursprüngliche Array `this.cells` mit lauter `null`-Einträgen (ohne dessen Größe zu ändern).
   4. Führe auf jeder Cell aus dem in 1. erstellten neuen Array `tick` aus. Der Parameter `cells` ist dabei jetzt `copyOfCells` und der Parameter `newCells` das aus `this.cells`. Damit steht nun das Endergebnis von `tick()` wieder in `this.cells`.



Extra-Aufgaben (unbewertet)

Equilibrium

Du kannst die Simulationsparameter konfigurieren um das Verhalten der Tiere und Pflanzen anzupassen. Zum Beispiel kannst du einstellen, dass sich Wölfe schneller vermehren. Gehe dazu in die Klasse `SimConfig` und verändere die Standardparameter.

Kannst du eine Parameter-Konfiguration finden die zu einem Equilibrium, also einem stabilen Gleichgewicht führt?

Falls du damit erfolgreich bist, poste es in den Zulip Channel, der zu dieser Aufgabe gehört!

Eye Candy

Dir ist sicher aufgefallen, dass die Grafiken in der Simulation ziemlich... langweilig sind. Wenn du Lust hast, gestalte gerne neue Grafiken und schicke sie in den Zulip Channel dieser Aufgabe. Die schönsten Grafiken werden nächstes Jahr für das Spiel verwendet.

Hier zählen nur selbstgestalte Grafiken.

Wenn du Bilder in den Channel schickst stimmst du explizit zu, dass wir deine Bilder für die PinguSim Aufgabe zukünftig verwenden dürfen.

Exercise details

Release date	Dec 5, 2025 17:00
Submission due date	Dec 14, 2025 17:00
Complaint possible	No