Making the assignment

You should make this assignment **individually**. You are allowed and encouraged to discuss ideas necessary to solve the assignment with your peers, but you should write the code by yourself. We will employ fraud checking software specifically created for programming classes such as this.

In case we suspect fraud, we can invite you to explain your code. Due to university policy cases of (suspected) fraud must be reported to the exam committee. The exam committee will then decide on possible disciplinary actions.

The deadline for this assignment is **Thursday, 1st of October 2020, 23:00, CEST**.

Grade and Scores

The grade of your assignment will depend on two important criteria:

1. Correctness: does your code work according to the specification in the assignment?
2. Code Quality and Style: is your code well written and understandable?

This assignment graded via **Codegrade** and counts for 20% towards your final grade. Note that this automated grade is not final: there is absolutely no guarantee that the automated grade provided by Codegrade will be your actual grade, as your work will also be assisted by human review. The weights for different elements of the assignment are:

Part of Assignment	Points
World	2
Cell	2
Creature	2
Herbivore	1
Carnivore	1
Code Quality and Style	2
Total	10

When we ask you to implement a function using a **specific** algorithm, you will get no points if you use a different algorithm.

Handing in your assignment

- Any scores generated by **Codegrade** only give an indication of your possible grade. Final grades will only be given after your code has been checked manually, and may differ from the preliminary automated feedback provided. In some cases there may be additional testcases that are only executed after you hand in your final code.
- Deliberate attempts to trick **Codegrade** into accepting answers that are clearly not a solution to the assignments will be considered *fraud*.

Code Quality and Style

Part of your grade will also depend on the quality and style of your code. **Codegrade** will check a number of rules automatically and compute a tentative score based on these rules. **Note that in the actual assignments, this score will likely be adjusted based upon manual inspection by human graders**. The automated tools do not find all issues, in and rare cases the automated graders may be too strict.

The rules applied by the automated grader come in four categories: *mandatory rules*, *important rules*, *slopy code rules* and *layout, style and naming rules.* This score is only an indication and is computed as follows:

- If any *mandatory rules* are broken, you get no points for code quality and style.
- If no *important rules* and *sloppy code rules* are broken, you get 1.5 points. For each *important rule* you break, 0.2 points are deducted. For each *sloppy code rule* you break, 0.1 points are deducted. It doesn't matter how often a particular rule is broken.
- If no *layout, style and naming rules* are broken, you get 0.5 points. For the first three violations, no points are deducted, but for each violation beyond that 0.1 point is deducted.
- Note that this score is only an indication of your code quality. Manual grading will always overrule the score indicated by Codegrade!

The detailed rules are explained in the "*Code quality and style guide*" of this course. They can be found on `https://erasmusuniversityautolab.github.io/FEB22012-StyleGuide/` There is a separate document on implemeting correct Javadoc comments. This can be found at: `https://erasmusuniversityautolab.github.io/FEB22012-StyleGuide/javadoc.html`

General Remarks

- The lecture slides usually contain a number of useful hints and examples for these assignments. If you get stuck, always check if something useful was discussed during the lectures. If you have question, feel free to contact us by e-mail.
- You can write your Java code with any editor your like. In the computer labs you will have access to BlueJ and Eclipse. We encourage you to use a good IDE, such as Eclipse, Netbeans or IntelliJ, as those have better features to help you with simple tasks and warn you in case of possible mistakes.

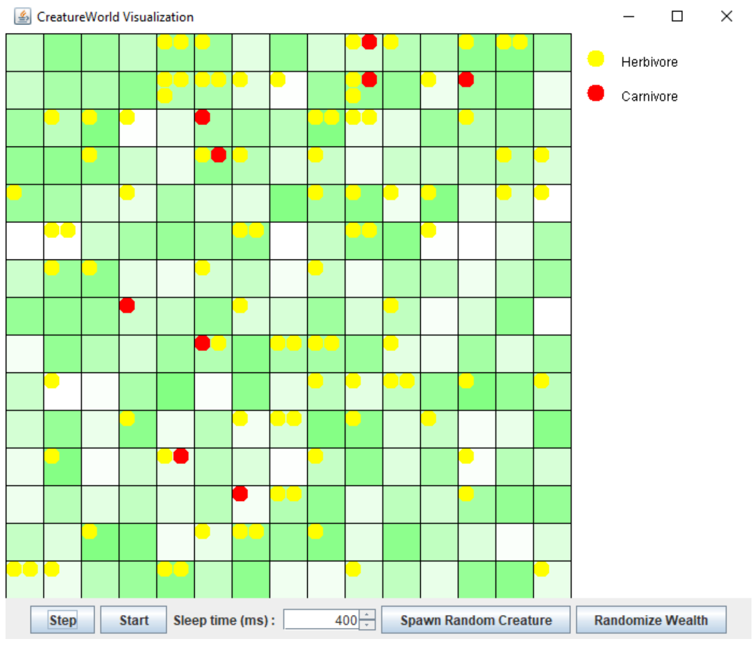

Figure 1: Screenshot of the visualizer that you can use after you complete this assignment

The Assignment

In the assignment of this week we will write a simulation of a world in which herbivores and carnivores live. To keep our simulation under control, we will restrict ourselves to a 2-dimensional world of distinct cells. You will create a `World` class and a `Cell` class for this in the first two exercises. A cell can contain herbivores, carnivores, or any other kind of `Creature` someone is willing to implement. Creatures in our world can all move or act, so we will first create an abstract `Creature` class that defines the basic properties of a creature.

Once we've done the steps above, we can create the `Herbivore`s and also the `Carnivore`s! Both of these classes will extend the `Creature` class, since herbivores and carnivores move and act differently.

Hints

Since the classes `World`, `Cell`, and `Creature` all depend on each other, it is good to first read the text of exercise 1 through 3 before starting to write your programs. Create the three classes and write empty methods, and fill them in as you make progress on the assignments.

Visualizing the World

After you've written the code for the exercises, you can visualize the simulation of the world using the code provided on Canvas. On Canvas you'll find the files `Controller.java`, `VisualizerMain.java`, `WorldWindow.java`, and `WorldPanel.java`. The `Controller` class controls the simulation of the interaction of the creatures, while the `WorldWindow` and `WorldPanel` classes are used for the visualization. Using these files and the classes you'll create in the exercises you can visualize the simulation using by running the `VisualizerMain` class. Figure 1 shows what this will look like.

World

In this exercise you will create a world for the creatures to live in. A world is a 2-dimensional area consisting of cells. Since these cells represent the state of their own part of the world, it is important that these `Cell` objects are managed by their World object. The world has a width (that corresponds to the x coordinate) and a height (that corresponds to the y coordinate). I should be possible to retrieve the width and height through specific get methods. In addition, the `World` should have a method which can return a `Cell` instance at specified coordinates and a method for getting all the creatures in the world. The public interface of the `World` class should thus be as follows:

```java
public class World
{
   // The following are required for half points
   public World(int w, int h) { ... }
   public int getWidth() { ... }
   public int getHeight() { ... }
   public Cell getCell(int x, int y) { ... }

   // The following are required for full points
   public List<Creature> getCreatures() { ... }
   public List<Cell> getCellList() { ... }
}
```

The corners of the world are at positions $(0, 0)$, $(0, h - 1)$, $(w - 1, 0)$ and $(w - 1, h - 1)$. You can view the world as a rectangle, where the corners and the edges of the world form an uncrossable border.

The constructor of the World class should initialize all the `Cell` instances in the world, using the constructor of the `Cell` class. When the `getCell` method is called with invalid coordinates, an `IllegalArgumentException` should be thrown. The variables `w` and `h` in the constructor are respectively the width and the height of the world, which should be obtainable using the `getWidth` and `getHeight` methods.

The `getCreatures` method should return a list of all the creatures in the current world by using the `getCreatures` method of the `Cell` class.

The `getCellList()` method should construct a new list of `Cell` objects, that contains all the cells in the world. Be careful that you are not allowed to return the same `List` object when `getCellList()` is called multiple times, whereas the `Cell` objects stored in these lists should be the same.

The next page contains some example code that can be used for testing, aside from the `VisualizerMain` class provided in the zip package with student files.

Example: every print statement in the following example code should print `true`. Note that the classes in this assignment are closely interrelated: mistakes in one class can lead to undesired behavior in another class. Futhermore, these test cases do not cover all cases and properties of the class. It is never a bad idea to write additional test cases yourself.

```java
// Testing width and height
World w = new World(5,3);
System.out.println(w.getWidth() == 5);
System.out.println(w.getHeight() == 3);

// Testing cell coordinates
System.out.println(w.getCell(1,2).getX() == 1);
System.out.println(w.getCell(1,2).getY() == 2);

// Testing if the cells are persistent
System.out.println(w.getCell(1,1) == w.getCell(1,1));

// Testing getCellList()
List<Cell> cells = w.getCellList();
System.out.println(cells.size() == w.getWidth() * w.getHeight());
System.out.println(cells.contains(w.getCell(0,0)));
System.out.println(cells.contains(w.getCell(4,0)));
System.out.println(cells.contains(w.getCell(0,2)));
System.out.println(cells.contains(w.getCell(4,2)));

// Testing getCreatures()
Carnivore c1 = new Carnivore();
Carnivore c2 = new Carnivore();
c1.moveTo(w.getCell(1,2));
c2.moveTo(w.getCell(2,1));
List<Creature> allCreatures = w.getCreatures();
System.out.println(allCreatures.size() == 2);
System.out.println(allCreatures.contains(c1));
System.out.println(allCreatures.contains(c2));
```

Note: this code is also available in the `MainStudent.java` file in the zip package with student files.

Cell

In this exercise you will create the `Cell` class. A cell is an element of our 2-dimensional world. As such, a cell has an x and an y coordinate specifying its location in the world. In addition, cells in our world can contain *plants*. These plants can be used by the creatures currently living in a cell for sustenance, for instance. Each cell can contain a maximum of 100 units of plants and a minimum of 0. Initially the value is also 0. The `Cell` class must therefore have a method for keeping track of the amount of plants it contains. It also has a method to change the number of plants. If calling this method would decrease the plants below the minimum, an `IllegalArgumentException` should be thrown. If it increases the plants past the maximum, it should just become the maximum. The maximum number of plants should be defined as a constant, i.e. a `public static final` variable of the `Cell` class with name `MAX_PLANTS`.

In addition, the `Cell` class must define a *natural order* on the cells. Therefore, the `Cell` class must implement the `Comparable` interface. The ordering of the cells should be implemented such that a cell with more plants is preferable over one with less plants. If two cells contain an equal number of plants, then a cell with less creatures is preferable over one with more creatures. More preferred cells should come earlier in the order than less preferred cells. The order does not need to be consistent with equals.

The public interface of the `Cell` class is defined as:

```java
public class Cell implements Comparable<Cell>
{
   // The following are required for half points
   public static final int MAX_PLANTS = ... ;
   public Cell(World w, int x, int y) { ... }
   public int getX() { ... }
   public int getY() { ... }
   public World getWorld() { ... }
   public int getPlants() { ... }
   public void changePlants(int amount) { ... }

   // The following are required for full points
   public void addCreature(Creature c) { ... }
   public void removeCreature(Creature c) { ... }
   public List<Creature> getCreatures() { ... }
}
```

Note that the constructor of a `Cell` takes a `World` class as input argument. The `addCreature()` and `removeCreature()` methods must be used to add or remove creatures to the cell. The `Cell` class should therefore maintain a list of the creatures currently residing in the cell. The `getCreatures` method should return a copy of this list, or a version of the list which can't be modified. You can either use a *copy constructor* or the static utility method `Collections.unmodifiableList` to achieve this.

The next page contains some example code that can be used for testing whether the natural order defined in a cell is compliant with the above specification.

Example: every print statement in the following example code should print `true`. Note that the classes in this assignment are closely interrelated: mistakes in one class can lead to undesired behavior in another class. Futhermore, these test cases do not cover all cases and properties of the class. It is never a bad idea to write additional test cases yourself.

```java
// Testing Cell Order
World w = new World(4,1);
Cell a = w.getCell(0,0);
Cell b = w.getCell(1,0);
Cell c = w.getCell(2,0);
Cell d = w.getCell(3,0);
a.changePlants(20);
b.changePlants(20);
new Carnivore().moveTo(b);
c.changePlants(30);
new Carnivore().moveTo(c);
new Carnivore().moveTo(c);
new Carnivore().moveTo(c);
d.changePlants(30);
new Carnivore().moveTo(d);
new Carnivore().moveTo(d);

// Sorting the cells according to their natural order
List<Cell> cells = Arrays.asList(a,b,c,d);
Collections.sort(cells);

// Check if the final indices are correct
System.out.println(a == cells.get(2));
System.out.println(b == cells.get(3));
System.out.println(c == cells.get(1));
System.out.println(d == cells.get(0));
```

Note: this code is also available in the `MainStudent.java` file in the zip package with student files.

Creature

In this assignment we will create a `Creature` class. A creature lives on a cell in the world and it can move, act, and die. It also has a certain sight, which indicates which surrounding `Cells` it is able to see. In later exercises we will use the `Creature` class as basis to implement more specific creatures. It is therefore an `abstract` class.

The `Creature` class has the following public interface:

```java
public abstract class Creature
{
    // The following are required for half points
    public Creature(int sight) { ... }
    public final void moveTo(Cell newCell) { ... }
    public final Cell getCurrentCell() { ... }
    public abstract void move();
    public abstract void act();
    public final int getSight() { ... }
    public final void die() { ... }
    public final boolean isAlive() { ... }
    public final int getEnergy() { ... }
    public final void changeEnergy(int amount) { ... }

    // The following is required for full points
    public final List<Cell> getVisibleCells() {...}
}
```

The constructor of the `Creature` class takes in the `int sight` variable. The value of `sight` should be final, and stored in the instance. It should also be obtainable using the `getSight` method.

The method `getVisibleCells` should return a `List` of `Cells` which are visible to the creature. We define a `Cell` as being visible to the `Creature` if and only if it has a distance of at most `sight` from the current `Cell`. The distance between two `Cells` is given by the following equation:

$$\mathrm{Dist}(c1, c2) = \max(|x_{c1} - x_{c2}|, |y_{c1} - y_{c2}|) \tag{1}$$

Where x_{ci} and y_{ci} are the respective x and y coordinates of `Cell` i. All the `Cells` which have a distance smaller than the `sight` of the creature should be contained in the returned `List`. A visual depiction of this distance metric can be seen in Figure 2.

To use the `getCurrentCell()` method it is necessary to store the `Cell` instance of the cell the creature is currently living on. The `moveTo` method of the `Creature` can be used to move a creature to a different `Cell`. You should make sure that you record the change of `Cell` also with the `Cell` instance of the current cell using the add and remove methods of the `Cell` class. As a consequence of this, a creature should always be moved by the `moveTo` method and not by adding a creature directly to a `Cell`. In our model a creature is considered dead if and only if it has `null` as its current cell. By default a new creature should not be alive, but become alive only when it is moved to a cell. Therefore, the `die` method should make sure that the creature is removed from its current cell, and change it current cell to null. Be careful to avoid `NullPointerExceptions` when you implement this.

Every creature also has a certain amount of energy, given as an `int`. The initial energy of every creature should be 20. The `changeEnergy` method should be used to alter the energy of a creature. If the energy is no longer positive, the creature dies and the `die` method should be called.

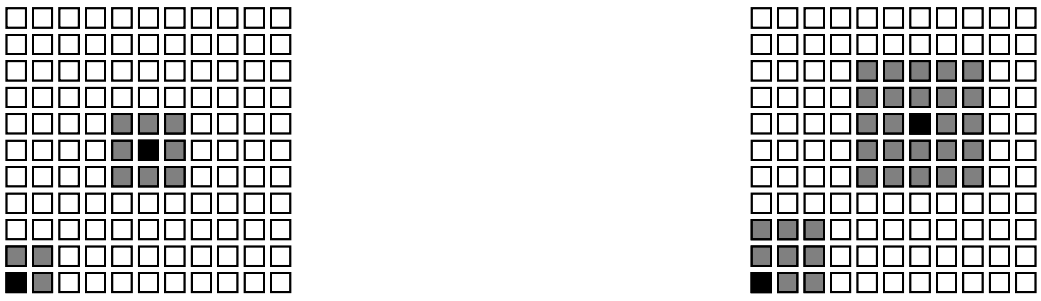
(a) Sight 1 (b) Sight 2

Figure 2: Visible cells for sight 1 and 2. The creature is indicated as a black cell, the visible cells are gray. The cell the creature resides on is always considered visible.

Below is some example code that can be used for testing whether the logic `Herbivore` inherits from the `Creature` class is according to specification.

Example: every print statement in the following example code should print `true`. Note that the classes in this assignment are closely interrelated: mistakes in one class can lead to undesired behavior in another class. Futhermore, these test cases do not cover all cases and properties of the class. It is never a bad idea to write additional test cases yourself.

```java
// Check the default stats of a creature using a new Herbivore
Herbivore h = new Herbivore(1);
System.out.println(h.getSight() == 1);
System.out.println(h.getEnergy() == 20);
System.out.println(!h.isAlive());
System.out.println(h.getCurrentCell() == null);

// Construct a World to check if we can put creatures on it
World w = new World(5,5);
Cell c = w.getCell(2, 2);
h.moveTo(c);
System.out.println(h.isAlive());
System.out.println(h.getCurrentCell() == c);
System.out.println(c.getCreatures().contains(h));

// Checks if the cells at range one are considered visibile
List<Cell> visible = h.getVisibleCells();
System.out.println(visible.contains(h.getCurrentCell()));
System.out.println(visible.contains(w.getCell(1,1)));
System.out.println(visible.contains(w.getCell(3,3)));
System.out.println(visible.contains(w.getCell(1,3)));
System.out.println(visible.contains(w.getCell(3,1)));

System.out.println(!visible.contains(w.getCell(0,0)));
System.out.println(!visible.contains(w.getCell(4,4)));
System.out.println(!visible.contains(w.getCell(4,0)));
System.out.println(!visible.contains(w.getCell(0,4)));
System.out.println(!visible.contains(w.getCell(2,4)));
System.out.println(!visible.contains(w.getCell(4,2)));

// Check if a creature behaves correctly when it dies
h.die();
System.out.println(!h.isAlive());
System.out.println(h.getCurrentCell() == null);
System.out.println(!c.getCreatures().contains(h));
```

Note: this code is also available in the `MainStudent.java` file in the zip package with student files.

Herbivore

In the previous exercises you've created a `World` which consists of `Cells` in which `Creatures` can live. It is now time to create a specific creature: the `Herbivore`. The public interface of the `Herbivore` class should be as follows:

```java
public class Herbivore extends Creature
{
   public Herbivore(int size) { ... }
   public int getSize(){ ... }
   public void move() { ... }
   public void act() { ... }
}
```

The constructor takes in an `int size` variable, which should be stored in the instance and should be obtainable via the `getSize` method. The size parameter given in the constructor should always be positive. If this is not the case, an `IllegalArgumentException` should be thrown. Every Herbivore has a sight of 1.

If a Herbivore is alive, it can move to all cells visible to it. If the herbivore decides to move, it should move to the best visible cell. The best visible cell is that which contains the most amount of trees and the least amount of creatures.

If the Herbivore acts it is able to eat some plants from the cell it is standing on. This increases the Herbivores energy by 1 for every plant eaten. If we denote the number of plants in the current cell by p_{cell}, the amount a Herbivore can eat is equal to the following expression:

$$H_{eat} = \min\left(p_{cell}, \left\lfloor \frac{2 \cdot size^2}{1 + size} \right\rfloor\right)$$

Do note that you also have to remove the appropriate amount of plants from the current cell, as well as increase the energy of the Herbivore. After the Herbivore has eaten, decrease its energy by *size* due to metabolism.

The next page contains some example code that can be used for testing whether the behavior of the `Herbivore` when it moves and acts is according to specification.

Example: every print statement in the following example code should print `true`. Note that the classes in this assignment are closely interrelated: mistakes in one class can lead to undesired behavior in another class. Futhermore, these test cases do not cover all cases and properties of the class. It is never a bad idea to write additional test cases yourself.

```java
// Testing Cell Order
World w = new World(4,2);
Cell start = w.getCell(1, 0);
// Initialize Herb the Herbivore
Herbivore herb = new Herbivore(3);
herb.moveTo(start);
// Is the size of Herb stored correctly?
System.out.println(herb.getSize() == 3);

// Set up some cells to which Herb should move
Cell a = w.getCell(0,0);
Cell b = w.getCell(0,1);
Cell c = w.getCell(1,1);
Cell d = w.getCell(2,0);
Cell e = w.getCell(3,0);
a.changePlants(20);
b.changePlants(20);
new Herbivore(1).moveTo(b);
c.changePlants(30);
new Herbivore(1).moveTo(c);
new Herbivore(1).moveTo(c);
new Herbivore(1).moveTo(c);
// Cell d will be the best choice, as it has only two herbivores
d.changePlants(30);
new Herbivore(1).moveTo(d);
new Herbivore(1).moveTo(d);
// Cell e would be best, but it won't be in range
e.changePlants(50);

// Let Herb move and check if it moves to the correct cell
herb.move();
System.out.println(herb.getCurrentCell() == d);
// Let Herb act, and check if the new numbers are correct
herb.act();
System.out.println(herb.getEnergy() == 21);
System.out.println(d.getPlants() == 30-4);
```

Note: this code is also available in the `MainStudent.java` file in the zip package with student files.

Carnivore

Apart from only Herbivores, we also populate our world with vicious Carnivores! These don't eat plants in order to say alive, but Herbivores. We once again extend the `Creature` class, such that the `Carnivore` class has the following structure:

```java
public class Carnivore extends Creature
{
    public Carnivore() { ... }
    public void move() { ... }
    public void act() { ... }
}
```

Instead of having an initial energy of 20, Carnivores should have an initial energy of 30. Furthermore, they should also have a sight of 2.

A Carnivore moves a bit different compared to Herbivores. Like the Herbivore, they are only allowed to move to `Cells` which are visible to them. However, the `Cell` they choose to move to is the Cell which has the most `Herbivores`. Furthermore, if two cells have an equal number of `Herbivores`, the `Carnivore` should prefer the `Cell` which has the largest `Herbivore`.

When the `act` method is called for a Carnivore, it chooses to eat the largest `Herbivore` in its current cell. In doing so it should call the `die` method of the Herbivore, and add the remaining energy that the Herbivore had left to its own. If there are no `Herbivores` in its current cell, the Carnivore should eat nothing. After `act` has been called, the `Carnivores` energy decreases by 6 due to metabolism.

The next page contains some example code that can be used for testing whether some of the logic of the `Carnivore` class is according to specification.

Example: every print statement in the following example code should print `true`. Note that the classes in this assignment are closely interrelated: mistakes in one class can lead to undesired behavior in another class. Futhermore, these test cases do not cover all cases and properties of the class. It is never a bad idea to write additional test cases yourself.

```java
// Build a small world for a Carnivore
World w = new World(6,1);
Cell cell;
cell = w.getCell(0, 0);
new Herbivore(10).moveTo(cell);
cell = w.getCell(1, 0);
new Herbivore(7).moveTo(cell);
new Herbivore(7).moveTo(cell);
cell = w.getCell(3, 0);
new Herbivore(50).moveTo(cell);
// This cell is the best according to the rule
cell = w.getCell(4, 0);
Herbivore eatMe = new Herbivore(10);
eatMe.moveTo(cell);
Herbivore dontEat = new Herbivore(1);
dontEat.moveTo(cell);
// This cell would be the best if it wasn't out of sight
cell = w.getCell(5, 0);
new Herbivore(60).moveTo(cell);
new Herbivore(60).moveTo(cell);
new Herbivore(60).moveTo(cell);

// Create the Carnivore
Carnivore c = new Carnivore();
System.out.println(c.getSight() == 2);
System.out.println(c.getEnergy() == 30);
cell = w.getCell(2, 0);
// Move the Carnivore
c.moveTo(cell);
c.move();
// Check if the Carnivore selected the best option
System.out.println(c.getCurrentCell() == w.getCell(4, 0));
cell = w.getCell(4, 0);
// Check if Carnivore behaves correctly when it acts
c.act();
System.out.println(c.getEnergy() == 44);
System.out.println(cell.getCreatures().contains(dontEat));
System.out.println(cell.getCreatures().contains(c));
System.out.println(!cell.getCreatures().contains(eatMe));
System.out.println(!eatMe.isAlive());
```

Note: this code is also available in the `MainStudent.java` file in the zip package with student files.