Creatures

User Documentation for the Greenfoot Scenario

Version 0.3

Contents

1 Introduction

This documentation first explains in Section 2 how to run the Greenfoot system and the MLS-AI-Project scenario »Creatures«. Section 3 gives a step-by-step description how to run the scenario and how to start writing your own brain implementation. Section 4 gives a comprehensive overview of the scenario features. Section 6 contains a reference, which explains in detail how the game works as well as all the methods of an animal that are available to the brain.

2 Installation

This section describes how to install the Greenfoot system on a Linux computer and how to load and run the MLS-AI-Project scenario. Since Greenfoot is a Java program, this should be similar for other operating systems, although path names will probably be different.

Before installing Greenfoot, make sure that a Java JDK is installed on the system, for example by typing `javac -version` on the command line. If the output is something like `javac 1.7.xyz` the JDK is installed, otherwise the output will be something like "`command not found`".

The Greenfoot environment can be downloaded from:

    http://www.greenfoot.org/download

Choose the "Pure Java" version, which downloads a JAR-file named "Greenfoot-generic-212.jar" (the version number may differ). Copy this file to a new directory somewhere on the filesystem (here we name the new directory "greenfoot"). Installing Greenfoot from the command line can then be done with the following commands:

```
java -jar Greenfoot-generic-212.jar
```

The installer will ask you for:

- a directory where to install the system (leave this unchanged for the current directory).
- the path to the Java JDK. On Linux this will probably be `/usr/lib/java` or `/opt/java`.

After clicking on "Install", Greenfoot can be started from the `greenfoot` executable:

```
./greenfoot
```

Download the »Creatures« project scenario and unpack it to a folder somewhere on the filesystem. You can now load the scenario by clicking on "Choose a scenario" and then selecting the previously created folder. You should then see a window which looks similar to the one shown in figure 1.

The major part of the window shows the world, which is build up by a number of cells. In the middle of the world there are some plants, and there are also some obstacles like the water around and the rocks. Later versions of the scenario may introduce other obstacles that will be loaded by default, but you can always go back to the first scenario by right-clicking on the `Scenario1` class in the "World classes" section on the right of the window and select "new Scenario1()".

On the right part of the window the classes for the source code of the scenario are displayed, as well as a button to compile them.

The three buttons below the world and the "Speed" slider are there to control the simulation. When you click on them not much will happen by now, because there is no animal living in the world.

3 Tutorial

This tutorial describes first how we can add an animal with one of the existing brains to the world and see how the animal moves around. We are then going to develop our own implementation of an animal brain.

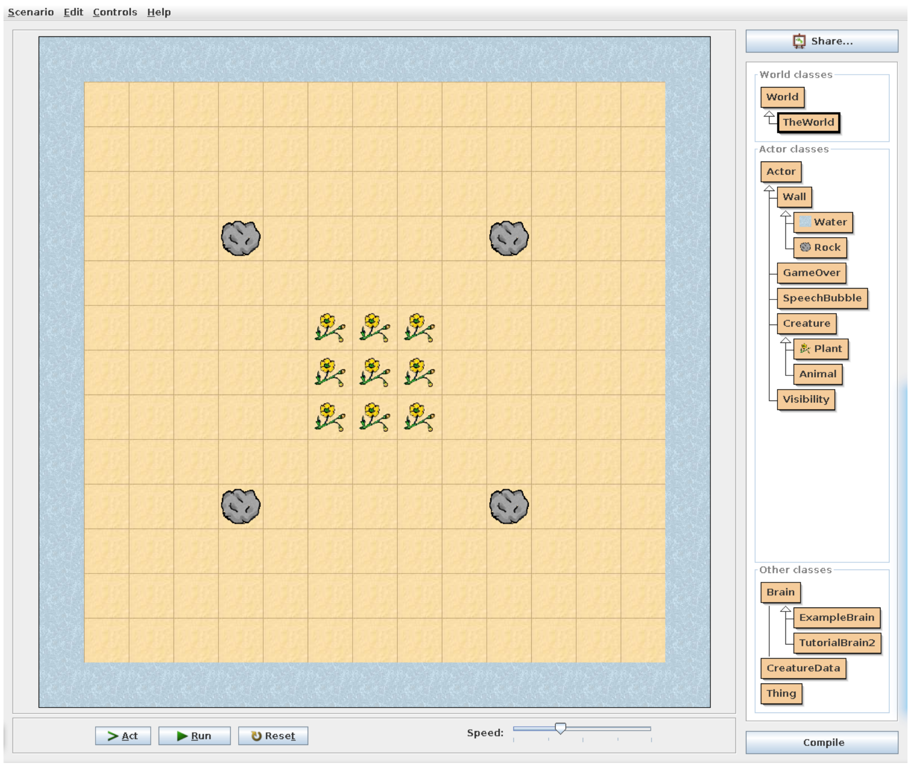

Figure 1: The first scenario. You may have to click on the “Compile” Button on the right side to see the world.

3.1 Adding an Animal

After opening the first scenario, we see the world with a few plants in the middle and some obstacles around it. If we click on the “Run” button, we will immediately get a notification that the game is over. This is because there was no animal in the world. For anything to happen, we need to add at least one animal first.

We go back to the starting configuration of the scenario by clicking the “Reset” button. To add a new animal, we right-click on the `Animal` class in the right window and choose *new Animal(Brain brain)*. If we then move the mouse over to the world, we can place it somewhere by clicking on a cell. Greenfoot will ask for a `Brain` instance as parameter for the constructor. We create a new instance of the first tutorial brain implementation here by typing `new TutorialBrain()` as the parameter, which is shown in figure 2. This creates a new animal instance which should then appear in the world. The animal has a brain attribute which is set to the tutorial brain instance we just created. The brain instance is responsible for the actions of the animal. Note that in order to run the simulation at least two animals have to exist in the world.

When we now click on the “Run” button, the animal is running around trying to eat plants. The bar on the left of the animal displays the amount of energy the animal has left, which decreases in every turn and increases when the animal eats a plant. We can also see that the plants are growing from time to time. The animal has to find and eat the plants in order to survive, while avoiding to eat all of them so that none is left. There are also some obstacles in the world which can block a cell, so that the animal has to move around them.

Actions like moving around or looking at the environment cost the animal more energy than just staying at one position and doing nothing. Therefore the animal should try to avoid unnecessary actions like looking too far or moving around in an area with no plants. However since the tutorial brain that controls the animal is not (yet) very smart, it will let the animal spend more energy than necessary and it should not take very long until the animal has no energy left and dies.

This example already gives us a brief overview for the scenario:

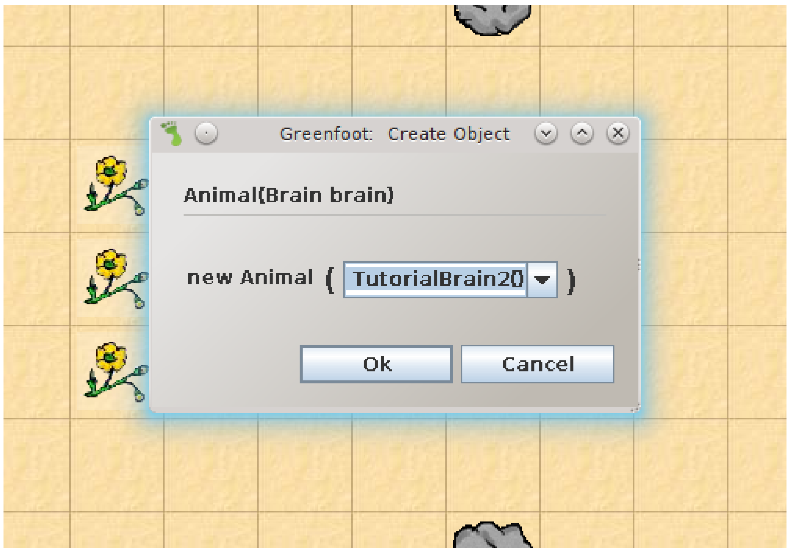

Figure 2: Adding an animal to the world.

- There have to be at least two *animals* living in the two-dimensional *world*.
- The world consists of *cells*, animals can move horizontal or vertical from cell to cell (but not diagonal).
- Beside animals there are also things like *obstacles* and *plants* in the world.
- The game is *turn*-based. Each animal gets the chance to do something (like moving, eating, looking around) in each turn.
- The *energy* of the animal decreases in each turn (even if it is only standing still), actions like moving cost additional energy.
- Animals may gain new energy by eating *plants*.
- An animal that has zero or less energy dies.
- The *goal of the game* for each animal is to survive as long as possible. The last animal which is alive, wins the game.

3.2 Moving

We have seen an animal that was controlled by the tutorial brain that is already included in the scenario. Now we are going to write our own implementation of a brain.

First we need a subclass [1] of the `Brain` class, similar to the `TutorialBrain` class. To create this class, we right-click on the `Brain` class in the lower right corner of the window and choose *New subclass...*. The name of the new class should be unique, for this tutorial we will go with the name `MyBrain`. We will get a new class on the same level as the `TutorialBrain` class. If we double-click on it, an editor opens where we can write our implementation. Whenever we change something in the implementation, Greenfoot indicates that we need to recompile the scenario before we can run it again by showing dashes above the classes that are affected by the changes. Clicking the "Compile" button will compile all classes.[2]

The created `MyBrain` already contains some code and comments, we can remove most of it and replace it with the code shown in listing 1. As we can see from this code, the brain has a `creature` attribute from which we can control the animal, for example by writing `this.creature.move(NORTH)`.

Listing 1: The basic `MyBrain.java`

```java
public class MyBrain extends Brain {

    // We have to use the init method instead of the
```

[1]Do not worry if you have never heard about the concept of subclasses. It is not necessary to know anything about it here.
[2]Depending on the Java and Greenfoot version there may appear some compiler warnings. They can safely be ignored.

```
    // constructor for initialization code.
    public void init() {
        this.creature.setCreatureImage("snake2.png");
    }

    public void think() {
        this.creature.move(EAST);
    }

    public String getName() {
        return "Tutorial Brain";
    }
}
```

We can set the image which is displayed for the animal by using the `setCreateImage(String path)` method of the creature attribute within the `init` method. The path can either be an absolute path to an image on the filesystem or just the name of an image from the "images" folder of the project. The `init` method can be used for all the initialization code of our implementation, which is useful because we can not use the constructor of this class. The `String getName()` method defines the name displayed for the animal, therefore we should implement it. Actions of the animal are implemented within the `think()` method. Here we simply move the animal one step in the same direction each turn. Note that the animal can only move once each turn. Adding another call to the move method within the think method would not move the animal twice in one turn (even though both calls would cost energy).

Now, if we want to add a new animal to the world which is controlled by our `MyBrain` (do not forget to compile the added code before). We can do this in the same way as we previously added the animal controlled by the tutorial brain, except that we type `new MyBrain()` for the brain parameter. Clicking on the "Run" button will then move the animal in one direction until it gets to an obstacle. Because the animal is not looking at the environment, it does not see the obstacle and tries to move in the same direction all the time until it has no energy left. It would even move through the plants without eating them (if placed at a corresponding starting position). Our next step will be to improve this by extending the `think()` method with code to see and eat plants.

3.3 Eating Plants

In each turn we can only either eat or move, because both are actions. Therefore we need to look at our environment (which is not counted as an action) to see whether there is a plant at our current position, or if we should move on.

The `lookAround(int range)` method allows us to look at our environment within the given range. The animal can not look arbitrarily far, as the range parameter is limited by the value returned by the `getMaxViewRange()` method. A call like `lookAround(creature.getMaxViewRange())` will therefore yield the maximal number of objects we can see from our current position. However this call is also quite expensive, as it will cost the animal quite a bit of energy. If we call the method with `lookAround(0)`, we will only see objects which are from the same cell as our current position, but this call will not cost any energy. In general we have to pay the more energy the further we want to look. Details for the costs can be found in the reference section.

As we can only eat plants which are located in the same cell as our current position, we only need a call of `lookAround(0)` to decide if we should eat. Listing 2 shows the modified think method of the tutorial brain which lets the animal eat plants if it sees them on the current cell. If it eats a plant, it also displays a message with the new energy level.

Listing 2: The extended `think()` method that allows the creature to eat plants

```
...
public void think() {
    // Get all things from the current cell
    Thing[] things = this.creature.lookAround(0);
    for(int i=0; i<things.length; i++) {
        Thing t = things[i];
        if(t.getType() == Thing.PLANT) {
            // If one of the things is a plant...
            // ... our action this turn is to eat.
            this.creature.eat();
            // Get new energy level
            int energy = this.creature.getEnergy();
            // and display it.
            int maxEnergy = this.creature.getMaxEnergy();
            this.creature.say("New energy level: [" + energy
                                                    + "/"
                                                    + maxEnergy + "]");

            // We return from the method,
            // because we do not want to move in this turn.
            return;
        }
    }
     this.creature.move(EAST);  // Otherwise just move to the given direction.
}
...
```

There were quite a few new methods that we have used here. First we get an array of things at our current position. Then we iterate through this array to see what kind of things there are. We are only interested in plants for now, so we check if the type of the thing is `PLANT`. If there is at least one plant our action this turn is to eat. The type of a thing object can be determined with the `getType()` method, which will return one of the following constants in this scenario:

– `Thing.ROCK`
– `Thing.WATER`
– `Thing.PLANT`
– `Thing.ANIMAL`

Other scenaros may contain different `Things` like `BEAR` or `MOUSE`.

One call to the `eat()` method will cause the animal to eat all plants on the current cell. Note that we return from the think method after the call to the eat method. This is to avoid the call of the move method, which would not move the animal in this case because our action in this turn already was to eat, but would cost us the same energy as if the animal would actually move.

We also use two other methods, the `getEnergy()` method which gives us the current energy level of the animal and the `getMaxEnergy()` method which gives us the upper limit of energy the animal can have. When we eat a plant the current energy level changes, and we use the `say(String msg)` method to display the current values in a speech bubble above the animal for a short time. All of these method calls do not cost any energy.

If we now place an animal somewhere in the world from where it moves through the plants, we can see that it actually eats them. Sooner or later however it will again run against an obstacle without noticing, trying to move in the same direction. Our next step will be to detect obstacles in front of us and change the direction in this case.

3.4 Detecting Obstacles

Looking only at the current cell can tell us if there are plants we can eat, but we can not see obstacles. We have to look at the cells around us, which means that we must increase the parameter for the `lookAround(int range)` method. Because we can only move one cell each turn we only need to look at the cells directly around us, as this costs us the least energy. So we choose `range = 1`, which means that we look at all 8 cells around us as well as the current one (so overall we look at 9 cells in this case). An alternative which would probably cost about the same amount of energy would be to look at more cells first to decide in which direction we should go, and then only look again whenever we moved a

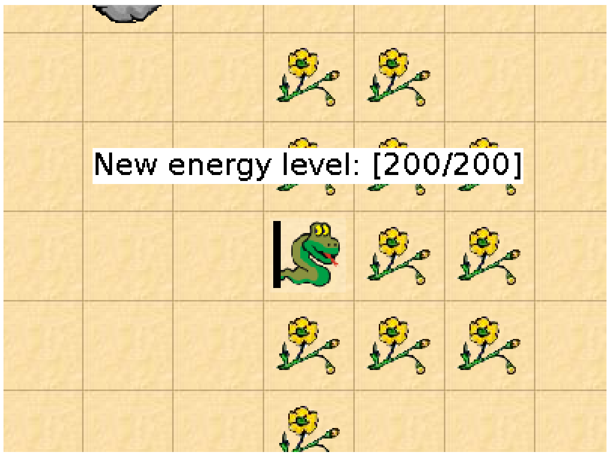

Figure 3: The animal eats a plant.

few steps into that direction. We will not implement this here, but this may be an idea to improve the implementation.

From calling the `lookAround(1)` method we again get an array of things, but because these things are now somewhere around us we need to look at their coordinates. This is done with the `getX()` and `getY()` methods (we need both our own coordinates as well as the coordinates of the obstacles in this case). Listing 3 shows the additional code which looks for obstacles before moving. Again, this is only done if we did not see a plant on the current cell before. We iterate through the array of things we get, and for each object that we see, we check the following two properties:

1. If the thing is an obstacle.
2. If it actually is an obstacle, whether it is located one cell east of us.

If there is no obstacle to the east of us we move there, and, if we have seen an obstacle there, we try to move to the north. To keep the code for this example simple, we do not check if we can actually move to the north or if there also is an obstacle. So if there is also an obstacle to the north, we repeatedly try to move there and spend energy for it without actually leaving the current cell. To avoid this, we should check if there is an obstacle for every direction where we might want to move and only move to a direction when there is no obstacle. Of course we can also change our direction if we do not see an obstacle, for example if we see a plant somewhere.

Listing 3: Looking to the east.

```java
Thing[] things = this.creature.lookHere();
for(int i=0; i<things.length; i++) {
    ...
}

int x = this.creature.getX();  // Get the current position of
int y = this.creature.getY();  // our animal.

// Get all things within a range of one cell around us.
things = this.creature.lookAround(1);
boolean east = true;
for(int i=0; i<things.length; i++) {
    Thing t = things[i];
    if(t.getType() == Thing.ROCK || t.getType() == Thing.WATER) {
        // If we see an obstacle,
        if(x + 1 == t.getX() && y == t.getY()) {
            // and the obstacle is one cell to the east
            // of us, we do not want to move there.
            east = false;
        }
    }
}

if(east) {
    // If we can move to the east, we move there.
    this.creature.move(EAST);
  }  else {
    // Otherwise we try to move north.
    this.creature.move(NORTH);
  }
```

Now with the methods we have seen so far, we can already implement a brain which is able to move an animal around the world, finding and eating plants while avoiding the obstacles. With the implementation developed here however, the animal will just move to one of the corners.

This can certainly be done better, some ideas for improvements may be:

- Instead of only looking at the current cell for plants, we could look around us to see if there are any plants, and then find a way to move there?
- What is the best path to move somewhere? When there are obstacles in the way?
- Once we have seen an obstacle, maybe we could remember the position so that we do not have to look at this cell again?
- Maybe we should only eat when our energy level is below a certain threshold and otherwise let the plants grow?
- If we do not see anything interesting in the area where we are currently, maybe we should just move a few steps to another area instead of moving around in the same area?

A summary of the methods described in this tutorial (as well as a few other methods) can be found in the reference section, which also describes what exactly happens during a round of the game.

3.5 Exporting the Brain Implementation

Exporting our brain implementation from the scenario can be useful when we want to re-use it with new versions of the scenario, or if we want to share our implementation with others.

To export our brain implementation, we can simply copy the file "MyBrain.java" from the source code folder of the scenario to the same folder of another project. To recognize the new class, the Greenfoot system has to be restarted. Remember to also copy the image representing your creature, if you use an image that is not shipped with the scenario.

4 Other Game Features

This section describes all the features available within the scenario.

4.1 Creature Speed

Creatures (this includes animals, bears, and mice) have different speeds. If a creature for example has speed 5, this means that the creature has 5 turns during one round of the game. The creature can then do 5 actions during the round (as there is one call to the `think()` method for each turn), while another creature may for example only be able to do 1. However it is not guaranteed that all turns of a creature during a round are consecutive. Actually if there are two creatures with the same speed, their turns will be alternating as shown in table 1 to minimize the advantage of having the first turn. Table 2 shows an example for when there are two creatures with different speed.

We can see the speed of our animal using the following creature method call:

```
int speed = this.creature.getSpeed();
```

Round 1				Round 2			
A	B	A	B	A	B	A	B

Table 1: Turns of a round with two creatures A and B, where both have speed 2. Both have two turns during each round, but they never have two turns in a row.

Round 1			Round 2		
A	B	A	A	B	A

Table 2: Turns of a round with two creatures A and B, where A has speed 2 and B has speed 1. Now A has two turns in a row.

Knowing the speed of our animal can help us if there is action that we only want to do once each round. Maybe our speed is 5 and only in our first turn each round we want look as far as we can because it would be too expensive to do this more than once a round. In the following turns we then just want to move without looking again, based on what we have seen before. We could implement this by having a counter that we increase every turn: `counter = (counter + 1) % speed;`. Then each time the counter is zero we can look around, and otherwise move or do something else.

4.2 Attacking

Creatures can attack each other, which causes the attacked creature to lose energy. Basically we can attack any creature, bears, other animals, and even plants which does not make much sense. We can only attack creatures on the same cell or directly next to us (not diagonal ones). An attack is an action, which means we can only attack once each turn. The `int getAttackValue()` creature method shows us the amount of damage we can do.

To attack, we call the `attack(Thing t)` method with the thing we want to attack as argument. The attack itself does not cost us energy, but we need a thing object which represents the creature we want to attack. We get this object from the `lookAround(int range)` method which may cost us energy depending on the range parameter. If the thing object is not representing a creature or is not on a cell within the allowed range, the attack is not carried out. Anyway the call is counted as the action of the turn. Listing 4 shows some example code to attack a bear:

Listing 4: Attacking a Bear

```java
// Look only one field around us, because we can only attack within this range.
Thing[] aroundUs = this.creature.lookAround(1);
for(int i=0; i<aroundUs.length; i++) {
    Thing t = aroundUs[i];
    // Test if we see a bear:
    if(t.getType() == Thing.BEAR) {
        // Also check if at least one coordinate of the bear is the same as ours,
        // so that we do not try to attack bears on a diagonal cell:
        if(t.getX() == this.creature.getX() || t.getY() == this.create.getY()) {
            // Attack the bear:
            this.creature.attack(t);
        }
    }
}
```

4.3 Bears

In some scenarios there are bears living in the world. We can see them using the `lookAround` method just like we can see plants or other animals. The `Thing.BEAR` constant tells us that a thing object represents a bear.

While the bears move only very slowly and do not eat plants, they can look pretty far and when they see animals, they try to attack them. Unfortunately for us the bears are also very strong, so when they attack us we lose a significant amount of energy. However, they can only attack us if we are standing on the same cell or right next to them and since an animal can move faster than a bear, we can always run away. We can also attack the bears just like they attack us, but they have a lot of energy and we are not nearly as strong as them. Therefore running away may often be the better option. If we really want to attack a bear we have to be careful and use our advantage in speed.

4.4 Mice

Besides of bears, there can also be mice in a scenario. Mice do not have much energy, but they are faster than bears and they eat our precious plants. If we do not attack them, they can become a real plague. They can also attack us, but they can not do much damage and attacking is not their focus. They walk around the plants they want to eat a few times, so this gives us some time to attack them before they actually eat the plant. However once they ate the plant, they have enough energy to give birth to a new mouse.

If we attack a mouse and it dies, we can eat it and increase our energy.

4.5 Energy and Calories

Creatures have an additional property called calories. When creature A eats creature B, the energy of A is increased by the calorie count of B (instead of the energy from B as before). Otherwise the energy of a creature is used as before, to define whether a creature is alive or not.

Creatures can not only plants, but any other dead creatures (plants can still be eaten if they are alive). Dead creatures stay a while in the world, with their calorie count decreasing. When the calorie count is zero, they disappear. The calorie value from a thing object is available from the `getCalories()` method.

4.6 Giving Birth

With the creatures `giveBirth()` method it is possible to create a new creature, which is controlled by a new instance of the same brain as the current creature. The new creature starts to act at of the next round of the game. The energy of the current creature is split up between both creatures, so that both will have half of the current one's energy. The new creature will inherit all the skills from the old creature, so that it has for example the same speed, visibility range and attack value. The new creature will appear somewhere next to the current one, therefore at least one of the surrounding cells (including the diagonal ones) has to be empty.

The following listing shows how the method can be called. It is not necessary to have a certain amount of energy to call the method, but if the animal has not much energy left dividing is probably not a good idea. Because then there will be two animals with even less energy each and the idle energy will be subtracted for both. So this method should probably only be called when the animal has "enough" energy or when there are many plants around.

```
if(this.creature.getEnergy() > 100) {
    this.creature.giveBirth();
}
```

5 Nice to Know

5.1 Changing the size of a cell.

If you can not see the whole world without scrolling because your screen resolution does not allow this, you may want to change the size of the cells. You can do this by editing the static `CELL_SIZE` variable defined at the top of the `TheWorld` class. Instead of 60 (pixels) you can set the value to maybe 50 or 40 and recompile the scenario, so that no scrolling should be necessary.

5.2 Display who is thinking at the moment.

If you right-click on a cell where no other object is located, you can run the "*Inherited from TheWorld* → `showLog()`" method from the displayed context menu by left-clicking on it. This shows a message during the game with the name of the creature whose turn is at the moment. This may be helpful when there are a number of animals with different brains in the world, and one of them runs into an endless loop but it is difficult to see which one.

5.3 Using the Eclipse IDE to implement a brain.

It is possible to use the Eclipse IDE to edit the brain implementation. To do this you can create a new Eclipse project from an existing source folder, which has to be the folder with the scenario. Then add the "greenfoot.jar" file from "greenfoot/lib/extensions/" as library to the build path of the Eclipse project.

However it is not possible to compile the project from Eclipse, so whenever you change something you have to click the Greenfoot compile button. Also when you add or remove classes, you have to restart the Greenfoot system.

6 Reference

6.1 One Round of the Game

Each *round* the game gives the brain of each animal the possibility to control the actions of the animal, after subtracting the idle energy. The idle energy is subtracted once every round for every animal. For each animal, at most one *action* is possible per *turn*. There may be more than one turn per round for an animal according to the speed of the animal. Each turn is defined by one call to the `think()` method of the brain.

The following steps describe what is happening during one round:

1. The idle energy is subtracted for each creature.
2. If there is no or only one animal left in the world, the last one which was alive wins the game.
3. If there is no plant left, a new plant may appear at the center of the world.
4. The `think()` method for each creature in the world which is alive is called. Energy for method calls which cost the creature energy is immediately subtracted during the call of such a method. If a creature attacks any other creature, the other creature immediately loses energy.

Creatures may lose energy at various steps during a round. If the remaining energy is below or equal to 0, the animal dies.

If an animal tries to do something that is not possible (like moving twice during one turn, moving against a wall, looking at a larger range than possible) the system will prevent the animal from doing so. It will however substract the energy for the impossible action.

6.2 Method Reference

The next sections describe all methods of the animal that the brain implementation is intended to use. Every brain instance has a `creature` attribute, where these methods can be called. So to move the animal one cell, the brain should call something like `this.creature.move(NORTH);`, to get the current x-coordinate of the animal `int x = this.creature.getX();`.

There are 3 different categories for the animal methods:

Action methods These are methods that specify an action, which may cost the animal some energy. Action methods can be called only once per turn. For example a creature can either move one cell per turn or eat something. Every following call to one of the activity methods in the same turn will not perform the action, but will cost energy nevertheless if the original action costs energy.

Info methods These methods provide information about the current status of the creature or the environment and can be called arbitrarily often each turn. However some of these methods may also cost a certain amount of energy.

Other methods Methods in this category do not have a effect for the game itself. They can be called arbitrarily often and do not cost energy. One example is the `say(String msg)` methods, which may be used for debugging.

6.3 Animal Activities

`void attack(Thing t)` Costs: 0

This tells the animal to attack the creature represented by `t`. The attack is only carried out when the following two conditions are fulfilled:

1. The thing has either type `Thing.BEAR`, `Thing.ANIMAL`, `Thing.MOUSE`, or `Thing.PLANT`.
2. The thing is either placed on the same cell as the animal or on one of the cells directly next to it (without the diagonal ones).

If so, the creature represented by the thing object will be attacked and probably lose some energy depending on the attack value of the animal.

`void giveBirth()` Costs: `creature.getEnergy() / 2`

This method creates a new animal with a new instance of the same brain as the "old" animal has. The old animal will lose half of its energy, which is the amount of energy the new animal will have. The new animal will inherit all skills (e.g. speed, visibility range) from the old animal. The new animal will appear on a random free cell around the old animal (including the diagonal ones), therefore there must be at least one such cell with no other object on it. Otherwise no new animal will be created but the call will anyway be counted as the action of the turn.

`void eat()` Costs: 0

This causes the animal to eat all plants and dead creatures on the current cell. The calories from each eaten object are added to the energy of the animal (there is an upper limit for the energy of an animal, which is available through the `getMaxEnergy()` method). If there are no plants or dead animals on the current cell, this will anyway count as the one allowed action for this the turn.

`void move(int direction)` Costs: `creature.getMoveCost()`

This moves the animal one cell to the specified direction, where `direction` should be one of

- `EAST`,
- `WEST`,
- `NORTH` or
- `SOUTH`.

The animal can not move diagonal. If the cell where the animal should move to is blocked, the animal will not move there. Nevertheless this will count as the one allowed action for the turn, and energy will be spend.

6.4 Animal Info Methods

`int getAttackValue()` Costs: 0

Returns the damage the animal can do when attacking.

`int getCalories()` Costs: 0

Returns the calorie value of the animal.

`int getIdleCost()` Costs: 0

Returns the amount of energy the animal spends in each turn (even when doing nothing). The idle cost may increase whenever the animal develops new skills.

`int getMoveCost()` Costs: 0

Returns the amount of energy the animal spends while moving one cell.

`int getEnergy()` Costs: 0

Returns the amount of energy the animal has left.

`int getMaxEnergy()` Costs: 0

Returns the maximal amount of energy the animal can have. If this is 100, and the current energy level is also 100, eating plants will not increase the energy level.

`int getMaxViewRange()` Costs: 0

Returns the maximal number of cells the animal can look into each direction.

`int getLookCost()` Costs: 0

Returns the energy cost of looking at the environment with `range = 1`. Looking at an environment with `range = 2` will increase the energy cost by 1, and looking at an environment with `range = 3` will increase cost by another 1.

`int getSpeed()` Costs: 0

Returns the speed of the animal.

`int getWorldHeight()` Costs: 0

Returns the height (in cells) of the world.

`int getWorldWidth()` Costs: 0

Returns the width (in cells) of the world.

`int getX()` Costs: 0

Returns the current x-coordinate of the animal.

`int getY()` Costs: 0

Returns the current y-coordinate of the animal.

`Thing[] lookAround(int range)` Costs: `getLookCost() * range`

Returns an array of things which are placed somewhere within the given range around the creature. For example if the range is 1, the method looks at the current cell and all 8 surrounding cells (including the diagonal ones). The `range` parameter has to be between 0 and `creature.getMaxViewRange()`, otherwise it will be set to the nearest one of these two values. Costs will depend on the original range parameter however (at least if it was not negative).

6.5 Other Animal Methods

`void say(String msg)` Costs: 0

Displays the given message above the animal for a short time.

6.6 Thing Objects

`Thing` objects are representations of other objects that the animal sees. They are returned by the `lookAround(int range)` method of the animal. The objects provide the following methods:

`int getX()` Costs: 0

Returns the x-coordinate of the represented object.

`int getY()` Costs: 0

Returns the y-coordinate of the represented object.

`int getType()` Costs: 0

Describes the type of the represented object. Returns one of:

- `Thing.ROCK`
- `Thing.WATER`
- `Thing.PLANT`
- `Thing.ANIMAL`
- `Thing.BEAR`
- `Thing.MOUSE`

`int getEnergy()` Costs: 0

Returns the current energy of the represented object. If the energy is equal or less than zero, the object is dead.

`int getCalories()` Costs: 0

Returns the calories of the represented object.