



## Players Map Traversal:

The traversal of maps was achieved as follows. Firstly, a second map was created to the current world as so:

```
GameMap secondMap = new GameMap(groundFactor, secondMap)
world.addMap(secondMap).
```

I then needed to add the instance variable “GameMap playersMap” to the world class, to always keep track of which map the player is currently on.

A class called crossMapsAction which inherits from action was also created. During world.run(), every time we process a players turn, we check to see whether or that player can cross into anoter following map.

If the player is at the northern border of the original map:

```
actorLocations.locationOf(player).y() == 0 and playersMap.equals(gameMaps.get(0))
```

If the player is at the southern border of the second map:

```
actorLocations.locationOf(player).y() == 24 and playersMap.equals(gameMaps.get(1)).
```

If either of these is true, a crossMapsAction will be added to the players action. Within this crossMapsAction, say if a player is moving from the northern border of the original map, they will then end up at the southern border of the secondMap.

So, if the players current “y coordinate” is 0, its new “y coordinate” in the second map will be 24, representing the last row of the secondMap. Its “x coordinate” will remain constant as it is moving north, so along the y-axis. The corresponding newLocation is then retrieved from the second map based on these new (x, y) coordinates.

1. This will then be the following process:

Remove the player from map one:

```
mapOne.removeActor(actor)
```

2. Add the player to map two:

```
mapTwo.addActor(actor, newLocation)
```

3. Now the entire world, will only focus on the second map, so the actorLocations get updated:

```
world.actorLocations = mapTwo.actorLocations
```

4. Lastly, the instance variable in world, “GameMap playersMap”, needs to be updated to the secondMap since the player has now moved here:

```
world.playersMap = world.actorLocations.locationOf(actor).map().
```



## Dinosaurs

The Dinosaur class is an abstract class that inherits from the Actor class. The following inherit from: Allosaur, Stegosaur, Agilisaur and Archaeopteryx.

### Hungry/Thirsty Dinosaur

On any given player turn, Dinosaurs across all the maps will be “ticked”. The newTick() function in the Dinosaur class achieves this.

Within newTick(), we call the getFoodLvl() on the dinosaur instance. If getFoodLvl() returns an integer  $\leq 10$ , a message get printed out in the console indicating that a at coordinates (x, y) is getting hungry. This lets the player know that this dinosaur may die soon and needs to be eat/drink.

### Grazing of Grass

For vegetarian and omnivorous dinosaurs, the newTick() method in the Dinosaur class gets the current location of these dinosaurs. Based on these location, we then call getGround(). Get ground will return the type of ground at this locations eg. Grass. If grass is returned, we increase the dinosaurs food level by 5 like so: `dinosaur.setFoodLvl(dinosaur.getFoodLvl() + 5)`.

From there, since we have the location and the grass here has been eaten, we revert the ground at the location back to dirt as so: `location.setGround(new Dirt())`.

### Breeding

To see if a dinosaur can breed, we first check if its food level is above 50 as so: `dinosaur.getFoodLvl > 50`.

From there, I retrieve all the adjacent locations to this dinosaur by calling `location.getValidAdjacentLocations()`, which returns an array list of the surrounding location. By looping over this array list of adjacent locations, we check whether the location contains an actor and if that actor is also a dinosaur:

`adjacent.containsActor() && adjacent.getActor() instanceof Dinosaur`.

If this passes we then proceed to call the breed function with the current dinosaur and its adjacent one.

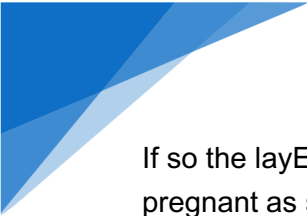
The breed method ensures that both dinosaurs are of the same type eg. Stegosaur, and that they are of opposite genders, are both adults and both have a food level above 50.

If so, the dinosaurs will breed. This is achieved by increasing the femalse instance variable numTurnsPregnant as so:

`dinosaur.setNumTurnsPregnant(dinosaur.getNumTurnsPregnant() + 1)`

### Laying an Egg:

Laying of eggs is also checked for during every newTick() call in the Dinosaur class. This is done by first checking whether: `dinosaur.isPregnant()` evaluates to true.



If so the layEgg() method gets called. We first update the number of turns this dinosaur is pregnant as so dinosaur.setNumTurnsPregnant(dinosaur.getNumTurnsPregnant() + 1). Then, if dinosaur.getNumTurnsPregnant() returns an integer greater than 10, and add a new egg instance with the same type as its parent to the current location like so:

```
Egg egg = new Egg(dinosaur.name)
```

```
location.addItem(egg).
```

### Attacking:

Only carnivorous dinosaurs initiate an attack. That is why we have the Boolean instance variable “isCarnivorous” set within the Dinosaur class. Carnivorous/Omnivorous dinosaurs are all types except for stegosaurs.

So when running newTick() in the Dinosaur class, we check to see if the current dinosaur is first carnivorous as so: d.isCarnivorous()

If that returns true the following happens:

All items at the current location are retrieved by location.getItems(). If any eggs are found, these carnivorous dinosaurs will eat those eggs.

Then, as seen before we retrieve the adjacent locations to the current carnivorous dinosaur as so: location.getValidAdjacentLocations(). By looping over this array list of locations, we check for any of adjacent dinosaurs as so: adjacent.containsActor() && adjacent.getActor().instanceof Dinosaur.

If that returns true, we then check the specific type of dinosaurs. If the adjacent dinosaur is an instance of Agilisaur, it will be killed straight away like so: agilisaur.setDead(true).

For all other dinosaur types, an integer array containing 2 numbers is created as follows: int [] array = {50, 100} and a randomiser is used to pick either 50 or 100 from the array.

The number chosen is assigned to the int variable hitPoints. The current dinosaur will attack the adjacent dinosaur as so: adjacentDinosaur.attack(hitPoints).

Since all dinosaurs maximum health is 100, and hitPoints can either take the value of 50 or 100, this ensures that the adjacent dinosaur can be killed in 1 or 2 hits.

We then check if hitPoints == 100, and if so, the killed dinosaur will be set to dead as so: adjacentDinosaur.setDead(true).

Since all dinosaurs have the instance variable int carcassFoodLvl, we increase the victorious dinosaurs food level by this amount as shown by this line of code:

```
currentDinosaur.eatCarcass(adjacentDinosaur.getCarcassFoodLvl()).
```

The killed dinosaur will then be removed from the gameMap:

```
gameMap.removeActor(adjacentDinosaur)
```



## Going Unconscious/Dying

Within the newTick() method, for dinosaurs with a food level below 50, we will set it to unconscious as so:

```
dinosaur.setUnconscious(true).
```

We then increment the number of turns this dinosaur has been unconscious as so:

```
dinosaur.setNumTurnsUnconscious(dinosaur.getNumTurnsUnconscious() + 1).
```

Then we check if dinosaur.getNumTurnsUnconscious() is  $\geq 20$ . If this evaluates to true, this dinosaur will be set to dead: dinosaur.setDead(true), and will be removed from the gameMap: gameMap.removeActor(dinosaur)

## Archaeopteryx and water traversal

This new functionality is implemented in assignment 3.

the flyAround() function in the archaeopteryx class is called within the Dinosaur class newTick() method on every player turn.

What the flyAround() method does is first retrieve all the adjacent locations to the archaeopteryx as so: locationOf(archaeopteryx).validAdjacentLocations.

Then an arraylist of strings called directions was created and by looping over array list of valid adjacent locations, by checking whether they were north, south, east or west to the archaeopteryx's current location, the strings "North", "South", "East" and "West" would be added to direction.

A randomiser randomly picks one of these valid directions, say North. I then traversed over each location North of the current location. This was simply achieved by retrieving the location at (x, currentLocation.y() - 1), since (y-1) indicates we are moving "up" or "north" along the y-axis. For each of these locations, their ground type is not water, and also if a player is not currently at that location. This is check as follows:

```
!nextLocation.containsActor() && !nextLocation.getGround() instanceof Water.
```


If the following location is not of ground type water, and no actors are at this following location, the archaeopteryx moves to that new location, thereby traversing pools of water.

## Ground

An ArrayList of type Object called items will be used to hold all the objects currently standing on a Ground at coordinates (15,3) for example. We chose to make it an ArrayList and not an Array since ArrayLists are mutable, and we can easily add or subtract objects from an ArrayList. Ground instances will have another 2 attributes: x and y coordinates. If a Map is created with dimensions 30 x 30, 900 Ground instances will automatically be instantiated starting with the coordinates (1,1), (1,2), (1,3) ... (30, 30). These coordinates **must** be unique to avoid having duplicate locations on a Map.

## EcoPoint

EcoPoint will be the currency used throughout the game. We did not find it necessary to create an EcoPoint class because a single EcoPoint instance could simply act as \$1.00, and would not need to have a value attribute unlike paper money in the real world. As such, we



let EcoPoints be an attribute for the Player class that can simply be added to or subtracted from when a Player makes a purchase.

## Egg

Eggs will be laid 10 turns after a female Dinosaur has bred. A type attribute has been added so that it can only take on the values of: allosaur, stegosaur, agilisaur, archaeopteryx.

When an Egg hatches, a baby dinosaur will be born as follows:

```
if(egg.type == "stegosaur"){  
    baby = new Stegosaur()  
}
```

Since all dinosaur instances are by default adults, we make use of the dinosaur instance variable stage, and set this to baby: `baby.setStage("baby")`.

Then this new baby dinosaur instance will be added to the current location of the hatched egg: `location.addActor(baby)`

And the egg can now be removed from the items at the current location:

```
location.removeItem(egg)
```

If an egg is successfully hatched, the `eggHatchRewards()` method gets called, which increases a players ecopoints based on the type of dinosaur that was just born.

Initially we thought that Egg should be a part of the stage attribute for a Dinosaur instance such as egg, baby, adult. However, we decided against this because it did not seem fit to be able to purchase a Dinosaur instance with stage "egg" from a VendingMachine.

We also faced the issue that once a female Allosaur lays an Egg, it might simply eat it because they are carnivorous. To avoid this we decided to add the "isEdible" boolean attribute to Eggs

This was mainly to ensure that Carnivorous dinosaurs did not immediately eat an egg once it was laid.


## Food

The following classes will all inherit from Food: Egg, Fruit, Grass, Hay, MealKit.

We chose to create this inheritance because since all of the above can be purchased and eaten by Dinosaurs, they will all have the same attributes like cost and the amount they will increase a Dinosaur's food level by when consumed.

The only reason we chose not to add Stegosaur to this list is because even though it can be eaten by an Allosaur, Adult Stegosaur cannot be purchased from a vending machine.

## Fruit



The attributes Fruits will have (cost and increase\_food\_level\_by) will be inherited from Food. Fruit instances will be deleted completely after 20 turns from lying on the Ground. This will be to mimic food rotting. This would help prevent Ground instances items from being over populated by rotten fruit and will help us save memory.

### Grass/Harvesting

Grass is an will inherit from Ground. In order for a player to harvest grass, we first need to check whether the current location the player is at has a ground type of grass. This is achieved as so: `actorLocations.locationOf(actor).getGround() instanceof Grass`. If that evaluates to true, the `harvestGrassAction` will be added to the players current actions. When a player chooses to harvest grass, we create a new instance of hay and add it to the players inventory as so: `actor.addItemToInventory(new Hay())`. Then, we set the ground at the current location back to dirt, to mimic the grass being harvested and the ground resuming to soil: `map.locationOf(actor).setGround(new Dirt())`

### Hay

Hay is created and stored within the ArrayList of Inventory whenever a grass instance is harvested by a player. It is also stored in the ArrayList of VendingMachine.

### LaserGun

Even though a LaserGun has very similar attributes and capabilities as the other food items in a VendingMachine, since a LaserGun is inedible, it will not inherit from Food. The LaserGun instances will have a constant cost attribute of 500. As it is not mentioned in the requirements whether a LaserGun can attack an Allosaur, for now we will assume that this is not possible.

### MealKit

The MealKit is a simple class that will inherit from Food. An additional attribute we will add to it is "type" to determine whether it is a vegetarian or carnivore MealKit. Based on this, we can determine whether a specific Dinosaur can eat a specific MealKit instance.

A MealKit instance will be deleted once it has been fed to a Dinosaur. If a Dinosaur already has a maximum food level, a MealKit will not be fed to a Dinosaur. This is to prevent Players from accidentally wasting this item and their EcoPoints.

### Player

To mimic a roguelike game, players will be able to choose their own names and be equipped with EcoPoints and an empty Inventory when starting the game. The Player class will likely be the most extensive in the game alongside the Dinosaur type classes.



## Stegosaur

The Stegosaur class will inherit from Dinosaur. This is because we expect any type of Dinosaur to have the same capabilities such as eating, breeding, getting hungry, attacking etc.

Since Allosaurs have capabilities and methods that Stegosaur do not, we decided to create 2 separate inheriting classes rather than simply adding a “type” attribute to the Dinosaur class.

Once a Stegosaur dies we do not intend to delete it right away, this allows time for Allosaurs to potentially feed on its carcass. Only after 15 turns and provided that an Allosaur is not currently eating a carcass, will we delete the Stegosaur instance as detailed above. This is to prevent our map from being littered with the carcasses of dead Stegosaur, mimicking a carcass turning to bare bones and disappearing into the ground.

## Tree

Trees can also be contained in the items of a Ground instance. This is to help us decide the likelihood of adjacent locations growing Grass. We initially thought of adding an ArrayList attribute to Tree to contain all of its Fruits. We decided not to implement it this way because there was no need to. Instead we simply decided to instantiate a Fruit instance (with a 40% chance of success) and add it into a Player's inventory every time they try to pick from a Tree.

Tree's will also contain an unlimited number of Fruit. We chose to make this number unlimited as it helps us standardize the chances of the Tree dropping Fruit and a Player picking Fruit.

If we had set a number of Fruit contained in a Tree, the chances of a Player picking fruit would be dependent on the number of fruit available, and we wanted to prevent this number from constantly changing. The same reasoning was applied with the dropping of fruit.


## VendingMachine

Vending machine contains a number of items which are stored in a hashmap. Items in vending machine are unlimited as there is only one vending machine in the whole map.

Since everything sold in the vending machine is either of type food or weapon, and both of these inherit from the class Item, the merchandise sold at the vending machine is initialised as an attribute as follows: `vendingMachine.merchandise = new ArrayList<Item>()`.

## World

When a player first starts playing the game, a World and GameMap will be created based on a set of dimensions such as 30 x 30. Once created, 900 Ground instances will instantly be



created, we will loop through these and based on an algorithm will add Grass to them. To do so, we will need to ascertain whether a plot of Grass is next to another location with Grass.

### The Grid

If a player created a 3 x 3 World.

(1,1)	(1,2)	(1,3)	(1,1)	(1,2)	(1,3)	x (1,1)	(1,2)	(1,3)
(2,1)	x (2,2)	(2,3)	(2,1)	(2,2)	x (2,3)	(2,1)	(2,2)	(2,3)
(3,1)	(3,2)	(3,3)	(3,1)	(3,2)	(3,3)	(3,1)	(3,2)	(3,3)

For each and every object with a game that needs awareness of its adjacent location, we will base it off of the Grid above. If “x” is at coordinates (2,2) it can only move among the highlighted blue squares. Likewise, to ascertain the probability of grass growing at “x”, adjacent locations will be evaluated to see if they too contain Grass or a Tree.