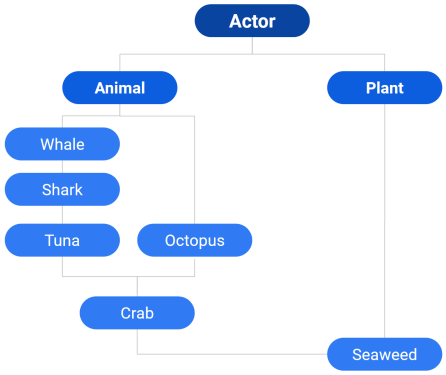
Coursework 3- Predator Prey Simulation

Made By: Sharlotte Koren (21051259) & Khushi Jaiswal (21008516)

Overview



The theme of our project is based on sea life. We used five different species in our model: whales, sharks, tunas, octopuses and crabs, as well as one type of plant: seaweed. The diagram pictured shows the food-chain hierarchy we used to implement in our simulation, with whales being our top predator and crabs being the only herbivores. As the food-chain descends, we added octopus as a competing predator with the tuna, meaning that crabs will be used to feed both species. Both animals and plants are affected by time, weather and temperature, while only animals are affected by disease.

Expanding the Simulation (species, behaviour and interactions)

Description	Implementation
Adding species	We first added more animals to the simulation and established a hierarchy with the whales eating the sharks, the sharks the tuna, and the tuna and octopus both competing for the crab. Each animal has its own class, each having very similar methods which enables all of them to only live to a certain age, have a fixed litter size, have a minimum age after which it can breed, a probability of giving birth, the food needed to survive and the food obtained from eating its prey.
Animal superclass	Because there are many attributes that are repeated in each animal class, naturally, we decided to create an animal superclass which contains some of their shared fields and methods and is inherited by all the subclasses using the *extends* keyword in each subclass. Inherited Methods include `breed()`, `setLocation()`, `setDead()` etc. We also created an object of class Random in the field of the animal class which enabled us to use it whenever we needed a random number in any of the 5 animal subclasses. Another example of the Animal superclass proving more useful was when we created an abstract `act()` method in the animal class. Because this Animal superclass do not have access to its subclasses (hence its fields), we added extra abstract methods in the animal class that would have to be overwritten in each subclass and would retrieve the required information from the subclass fields e.g `getMaxLitterSize()`, `getCanBreed()`.
Predator competition	We made sure that at least two of the predators competed for the same prey; the tuna and octopus both feed on the crab. This was quite a simple task but due to this change we had to make sure that the number of crabs in the simulation was always greater than the other animals as they were eaten quicker than the other animals. This abundance of crabs was attained by altering fields such as the litter size, the maximum age and the breeding probability.
	Previously, an animal would breed with any animal in its neighbouring cell as long as it was of the same species, but we opted for a more realistic

Differentiation between gender	approach and created two genders of each species; male and female. For this we created a boolean field *gender* in the animal superclass which is initialised and given a random boolean value in the constructor using the `nextBoolean()` method of the Random class. Once initialised, the `isFemale()` method gives each animal a gender. This information can then be used in all animal subclasses.
Mating	All of our animals have the ability to find a partner of the opposite sex and use it to mate and produce more animals of the same species. To do this, the animal has to look in its adjacent cells and check whether the animal next to it is of the same species and opposite gender. First, we made a method called `isFemale()` that uses the two boolean values as being true for females and false for males, so that only the females would be looking for the males. In each of the species' individual classes, we created a method called `oppositeGender()` which searches the nearby cells for an animal that fits both of the criteria above, and returns a boolean value to confirm this. In their `act()` methods, we then used both methods to confirm whether or not the female can then give birth.
Day and Night	Next, we wanted to add the concept of daytime and nighttime. In order to avoid code duplication and cohesion, we created a separate class dedicated just for the time. In there, we decided to initialise a variable for the hours, and use that to keep track of the days. To make sure the hours looped properly, we made sure that in the `incrementHours()` method, as soon as the hours reached 24 hours, we used the modulus sign to ensure that it went back to zero. Then, we decided that daytime should last between seven a.m. to seven p.m., so we used a boolean variable `isDay()` to switch in between the two.
Effect of time on animals	We decided to give some animals the ability to 'sleep' at night: essentially, they stop moving and reproducing during the night hours. We chose that the whales and sharks would have the ability to do this. To do this, we had to link the hours of the day to the simulation. So, in the simulator class, we assigned one step to be equivalent to one hour, which means that every twelve steps, the conditions should switch from daytime to nighttime. So, in the `act()` method, we added the daytime condition so that, when false, nothing in the method gets executed.
Adding species onto the GUI grid	After the creation of the animals was complete, we went into the SimulatorView class and added all of the new species into the grid and assigned them different colours. We then added a *Day* label, which we used to display the number of days that have passed. In order to get this, we added a day tracker into the Simulator class. In addition, we added a label that shows whether the current step is during the day or night, which we did directly in the *SimulatorView* class. Lastly, we used the hour variable from the Time class to keep track of the hours using an hour label that we created.

Extension Tasks

Description	Implementation
(1) Simulate plants	**"Plants grow at a given rate, but they do not move"** Initially, we created the simulation so that plants are assumed to always be available, and they never die. However, we now wanted to formally add seaweed to the simulation and we wanted to create it in a more realistic way. So, we decided that the most efficient and legible way to do this is by extending the inheritance hierarchy. The *Animal* class is now a subclass as well as a superclass as we made room for an *Actor* class: this now represents all creatures that exist in the simulation. We then moved all methods and fields that were going to be in common with both animals and plants into the Actor class, like their age, field and location as well as the `act()` and `setDead()` methods. This then allowed us to create a *Plant* class, which we extended into *Seaweed*. By creating a plant class, we allowed for opportunities to create more than one type of plant if the code were to be improved in the future. In the Plant class, we added all the methods and fields that were then common to all of the plants, like `breed()` and `incrementAge()`. For the plants to grow at a steady rate, we let their age increment every time they act. They also do not move, only reproduce: in their `act()` method, their only ability in the simulation is to breed and age.
	The influence of the time of day on plant reproduction In the *Seaweed* class, instead of creating one *BREEDING_PROBABILITY* variable just like we had for animals, we decided to create two: one for the seaweed's probability to reproduce during the day, and one for the night. In our method called `getBreedingProbability()`, we made sure that during the day, the correct breeding probability is used to reproduce, which we made higher than their likelihood to breed at night.
	"Some creatures eat plants" In our model of the simulation, only crabs have the ability to eat the seaweed. Therefore, in our *Crab* class we programmed it to eat the seaweed in its `findFood()` method.
	"[The creatures] will die if they don't find their food plant" We used similar methods to other predators in the simulation to program the crabs to die of hunger, using the `incrementHunger()` method.
(2) Simulate weather	**"Weather can change"** To implement weather into the simulation, we added a separate *Weather* class to keep everything uniform. Since our project is based in the ocean, we decided to come up with four different weather types that each have an influence on sea life. We assigned two different types of weather to occur in the daytime: sunny and cloudy, and two at nighttime: hurricane and windy. We gave each one a different probability of occurring using the `setWeather()` method, and five weather types occur in each 24 hour period, which we ensured would happen in the *Simulator* class. We also added a label in the *SimulatorView* class so that we can keep track of the weather on the screen.
	"It influences the behaviour of some simulated aspects" Biologically, the weather affects the way species interact. So, in our simulation we chose this to affect the seaweed. When the weather is

	sunny or cloudy, seaweed (on average) produces a higher number of offspring than other conditions like wind and hurricane. Therefore, as the weather changes, seaweed produce a different number of offspring. We made sure to do this in the *Seaweed* class by changing the `getOffspringSize()` method to return the correct maximum size according to the weather conditions, which we set using two different fields.
(3) Simulate disease	**"Some animals are occasionally infected"** For animals to catch an infection we created a new disease class. The disease class has an `act()` method which has two main functions: ● It gets the array of all actors from the simulator ● It is responsible for starting a new wave of disease if it has been a certain number of steps since the previous disease ended. It does this by calling the `startDisease()` method. The `startDisease()` method called in the `act()` method iterates through the array of all actors and picks a random number of actors and if the actor is an animal, it gives it the disease using the `getInfection()` method. The `getInfection()` method is in the animal superclass because any animal can get the infection. This method sets the relevant animal to have the infection by setting the field variable *hasInfection* to true. If a single animal has the disease for a certain number of steps, it will die, this number can be specified in the *Disease* class' field variable *STEPS_TILL_DEATH*. It can also be healed if it's body responds to the infection. The method `setHealOrDie()` will pick a random number less than specified *STEPS_TILL_DEATH* times two and heal if the body responds in less steps than it takes to die.
	"Infection can spread to other animals when they meet" Once an animal has an infection, it can pass it on to an animal of any species using the `transferDisease()` method. As this method is applicable to all animals, we put it in the *Animal* class. The `transferDisease()` method gets all the actors in an animal's adjacent cells and if the actor is an animal, is alive, does not already have the infection and the spread probability is met, the disease is passed on to the actor.
(4) Simulate temperature (our own challenge)	**Temperature can change** For our own challenge we created another class called *Temperature*, the temperature class declares a string field variable, *temperature*, whose value is altered depending on the weather at a given point in time.
	Determining the temperature The temperature is set in the `SetTemperature()` method which has a different range of temperatures for each weather type discussed above. The range is implemented by picking a random number from a small range and then adding that randomly selected value to a base temperature. The "range to pick from" and" base temperatures" are unique for each weather.