CS51 PROBLEM SET 7:
LIFE OF COW

This problem set is due MONDAY, April 18, 2016 at 5:00pm EST.
You may work with a partner on this problem set.

0. Introduction

0.1. **Getting started.** As in the previous assignments, all of your programs must compile. **Programs that do not compile will receive an automatic zero. Absolutely no exceptions will be made. You must submit code that compiles.**

Make sure that the functions you are asked to write have the correct names and the type signature specified in the assignment. Please pay attention to style and follow the style guidelines posted on the course web site. Think carefully before writing the code, and try to come up with simple, elegant solutions.

As usual, to retrieve the problem set, follow the instructions found here.

Testing: This time, we are not requiring that you include explicit tests. Of course, you must still perform your own testing to ensure your code works as required!

Grading: Your code will be evaluated on correctness, design (including testing), and style (see style guide).

Collaboration policy: You are encouraged to work with a partner on this problem set.

0.2. **The world.** In this problem set you will implement a simulated world of objects that interact in interesting ways. We have staged the process in six parts, each described in a separate section below. We recommend that you read through the entire problem set to get a sense of what you'll be implementing before doing any programming.

In the simulated world, there are three animals: BEES (yellow circles), BEARS (brown circles), and cows (purple circles). Bees pollinate FLOWERS (pink circles) and bring pollen back to the BEEHIVE (the teal circle in the middle) to be turned into honey, which in turn gets turned into more bees. Bears come out of the CAVE (lower left of the world) to steal honey when they can smell that the beehive has accumulated a large amount of it. When the hive is attacked, it alarms its worker bees of danger and the bees respond by attacking the bear. Cows come out of the pasture (upper right) to eat flowers when they smell that the world has accumulated enough of them. Flowers multiply and propagate, but die if not

BEES
BEARS
COWS
FLOWERS
BEEHIVE
CAVE

pollinated. The red bars above objects in the world displays the object's remaining life. When a bee or flower dies, it leaves behind DUST, which remains for a moment before it disappears. There are also PONDS (blue circles), which serve as obstacles for the world's inhabitants.

The world is implemented as a reactive system – EVENTS are triggered in the system and objects react to the events.

You will use objects in OCaml for two purposes: to give a common interface to all world objects, and to share code among objects that exhibit similar behavior.

After the simulation starts you can press 'space' to pause, 'f' to make it run faster, 's' to make it run slower, and 'q' to quit. To make things a bit easier, we've created a series of videos that demonstrate what your problem set should look like after you complete each part. You can find them in the Problem Set 7 folder under Checkpoint Videos.

0.3. **General tips.** This is a list of general tips (and, in some cases, requirements that we will look for when grading your assignment). Please read the entire section and keep it in mind for *every* part of the assignment.

Version control: Using `git` is more important on this problem set than it has been on any previous problem set, as there are many files to change, and you are *likely* to want to either reference or return to previous versions of your code. **Commit and push early and often, especially when you have working code after solving a problem.**

Staging: At the end of each part, we provide a summary task list, listing each of the tasks you should do to implement the section. This problem set has been designed to do in the order presented in these task lists. Doing this problem set out of order will not be a happy experience.

Objects reference: For your inquiries about the OCaml object system, consult Chapter 3: The Object Layer in the OCaml manual. What does mutable do? What does virtual mean? This reference reveals all!

Graphics reference: As a reference for the graphical and event system we are building on top of, you may find the OCaml Graphics Module documentation useful. However, you shouldn't need to use this library directly for anything other than colors.

File structure: While there are many files provided with this problem set, the major infrastructure can be found in the following files:

- `Draw.ml`: This should fulfill all of your drawing needs for this assignment, namely drawing circles and status bars. There are helper methods in the `world_object` class that call these functions for your convenience.

- `Event.ml`: Our abstraction over the primitive OCaml graphics event system. This module allows you to add and remove listeners to events.
- `World.ml`: The representation of the world along with path helpers.
- `Direction.ml`: A file containing the direction data type and helper functions.
- `UI.ml`: A nice interfaces that glues together the event and graphical frameworks; also sets up the standard keyboard events for 'space', 'f', 's', and 'q'.
- `Helpers.ml`: A bunch of helper functions that we found useful in developing the staff solution for this assignment. Look at all the functions in this file; every one of them is useful!

Compiling: You can compile the program using `ocamlbuild` by executing the following command from the problem set directory.

```
% ocamlbuild Main.byte
```

If a build fails at any point during your progress on this assignment, try removing all compiled bytecode and your `_build` directory (`rm -rf _build *.byte`) and building again. These build inconsistencies typically arise as you recompile code against a changed interface.

You might want to create a `Makefile` for both of these commands (and perhaps others) to make your life easier.

Following instructions: When we tell you the name a method should have (like `do_move` as an event handler to the move event) and the location it should go in the code (like the `Event Handlers` section), we really mean it. Don't change the name or locations of these methods.

TODOs: Most of the code for any given part of this assignment should be placed in a `(* ### TODO: Part X XXX ### *)` block. Feel free to delete these blocks as you fill them in. Not all code is expected to be placed in these blocks. For example, changing the super-class of a class or importing another module will not correspond to a `TODO` marker. All new methods and most extensions to methods will correspond to a `TODO` marker.

Opening modules: Not all modules are opened at the top of every file. So, if you need frequent access to functions from a module that is not opened in a file, feel free to open the module.

Random integers: You should use `World.rand` as a random integer generator. Its signature is `(int -> int)`, and given an integer `n`, it generates a random number between 0 and n-1.

Other randomness: You may also find `Helpers.with_inv_probability_or` and `Helpers.with_inv_probability` useful. You may use these functions for probabilistic actions (but you are not required to).

Event listeners: Any class that inherits from `world_object` MUST register handlers using `self#register_handler`, and *not* `Event.add_listener`. This is because `self#register_handler` will automatically remove the listener if the object dies and bad things will happen if you don't do this.

Base objects: You should not be modifying `WorldObjectI.ml` or `WorldObject.ml` *at all* for this assignment. Points will be deducted if you modify these files.

Directions: You should never use `Direction.natural` to get a direction between points. Use `World.direction_from_to` which returns a random direction if there is an obstacle between the points.

Simulation speed: The speed of the simulation may vary on your computer. Feel free to edit the initial delay in `UI.ml` to a more desirable starting delay (this is the value that 'f' and 's' decrement and increment).

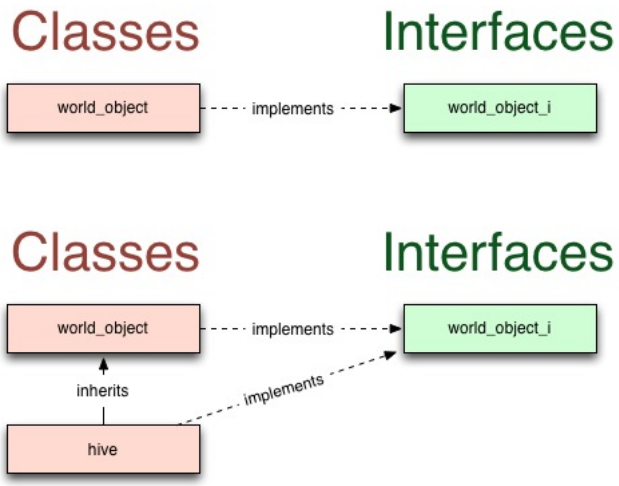
1. Basic objects

For this part of the assignment, you should only need to place code in locations that are marked `(* ### TODO: Part 1 Basic ### *)`.

Compile the program from the command line, then execute `./Main.byte part1`. You should see green circles in the lower left hand corner. Look at the `part1_initializer` function in `Main.ml`, and you can see 8 objects being created: a pond, a flower, a hive, a bee, a cave, a bear, a pasture, and a cow. Each of these objects inherit from the `world_object` class. Observe how in the `world_object` class, a `world_object` object adds itself to the world upon initialization. There is lots of other functionality in `world_object`, which all derived classes inherit.

The `world_object` class is an implementation of the class type `world_object_i`.

The class type `world_object_i` declares methods that an object in the world needs in order to interact with the rest of the world. All of the objects in the



world will inherit from `world_object` and will be placed in the world as objects of type `world_object_i`. (See the up-cast in `initializer` in `WorldObject.ml`.) This abstraction allows the world to call only those methods that are relevant to world interaction, and no others.

If you look in `world_object`, you will see that its `draw` method draws a green circle and that its `get_name` method returns `object`. Override both of these methods, for all 8 classes. Use the helper functions in the `Draw` module (or, even better, the wrapper methods for these functions in the `world_object` class), not primitive `Graphics` operations. Some objects draw text on top of their circles. Reference the following table for a description of how each object should look.

Object	Name	Text	Text Color	Color
Pond	`pond`	None	N/A	`Graphics.blue`
Flower	`flower`	# of pollen	`Graphics.black`	`Graphics.rgb 255 150 255`
Hive	`hive`	# of pollen	`Graphics.black`	`Graphics.cyan`
Bee	`bee`	# of pollen	`Graphics.black`	`Graphics.yellow`
Cave	`cave`	capital 'C'	`Graphics.white`	`Graphics.black`
Bear	`bear`	# of honey	`Graphics.black`	`Graphics.rgb 170 130 110`
Pasture	`pasture`	capital 'P'	`Graphics.white`	`Graphics.rgb 70 100 130`
Cow	`cow`	None	N/A	`Graphics.rgb 180 140 255`

There is a `draw_z_axis` method in the `world_object_i` interface. When there are multiple objects positioned at the same point in the world, the one with the highest z-axis will be drawn on top. Give all (or some) of the 8 objects a `draw_z_axis` method which causes the objects to be drawn in this order (from drawing-first

to drawing-last):

$$\texttt{cow} > \texttt{bear} > \texttt{bee} > \{\texttt{pasture}, \texttt{cave}, \texttt{hive}, \texttt{flower}, \texttt{pond}\}$$

For example, a cow must draw on top of everything else, but there need not be any draw order between a pasture and a cave.

We also want the pond to be an obstacle. Override `is_obstacle` in the `pond` class to return `true`.

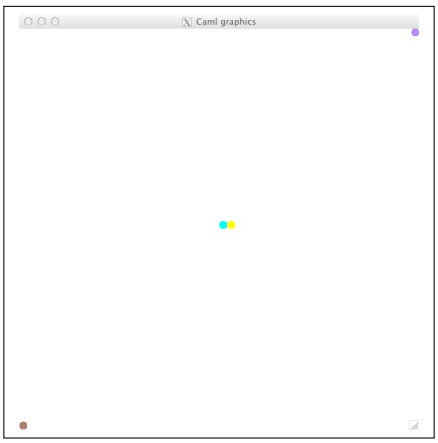
Now compile your code and execute `./Main.byte part1` again and see the objects have changed to draw with pretty colors.

1.1. **Summary task list for part 1:**

- Override `get_name`, `draw`, and (potentially) `draw_z_axis` in the 8 classes.
- Make the pond an obstacle.
- `./Main.byte part1` should compile and run with expected behavior.
- Using `git`, commit and push your working solution so that you have a checkpoint and a backup to which to return. After commiting, running `git status` should report that you have *no* untracked changes to any `.ml` files. You will be happy that you have used version control.

2. Movement

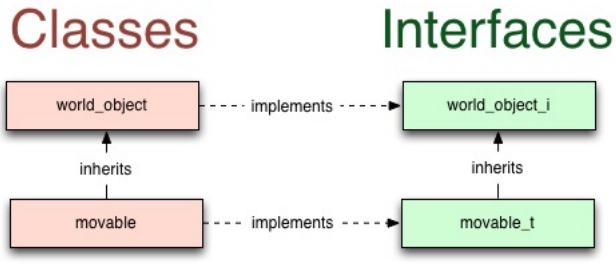
For this part of the assignment, you should only need to implement new methods in locations that are marked `(* ### TODO: Part 2 Movement ### *)`.



Execute `./Main.byte part2`. The world is now set up with a hive in the middle with a bee next to it, a bear in the lower left corner, and a cow in the upper right corner. Having objects just sit around isn't much fun; let's make them move around.

Note that `event_loop` function in `Main.ml` (which gets called every time the world is re-drawn) fires the `World.move_event`. You will make objects move around by reacting to this event.

We want bees, bears, and cows to move. Rather than implement movability separately for each of these three classes, we will implement it once in a `movable` base class, and then customize it in sub-classes. Code reuse is a good thing!



Look at `Movable.ml`. Notice that a class type `movable_t` is defined. This is a class type for objects which inherit from the `movable` class.

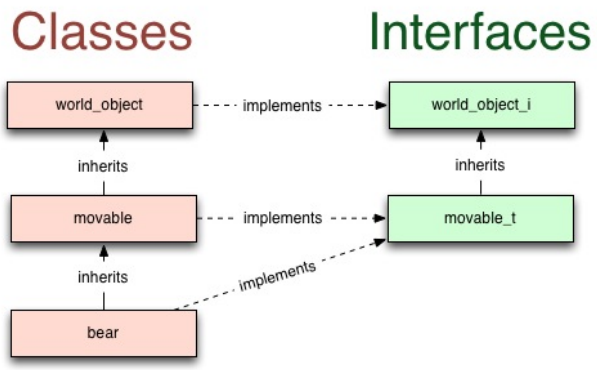
`Movable` is designed to take in a value of type `int option` as a speed. It is called `inv_speed` because the larger the number, the slower the objects will move. Add

an initializer to the `movable` class which adds an event listener to `World.move_event`. A `movable` object which is initialized with `inv_speed (Some s)` should move every `s` fires of the `World.move_event`.[1] A `movable` object which is initialized with `inv_speed None` should never move.

One way to implement this is to buffer `World.move_event` using the buffer combinator in the `WEvent` module. See `WEvent.ml` to learn how the buffer combinator works.

The listener function should call `self#do_move` in reaction to a move event. **Define `do_move` in the `Event Handlers` section of `Movable.ml`.** `do_move` should move the object (see `world_object#move`) in the direction of `next_direction` which is declared in the `movable_t` interface. Note that `do_move` must be private because it is not exposed by class type `movable_t`.

In the `Movable Methods` section, you should implement the method `next_direction` to have a sensible default.



Now that we have a `movable` interface, make `bee`, `bear`, and `cow` inherit from `movable`. Use the speeds for each object already defined at the top of each class file.

Override the method `next_direction` for each of the `bee`, `bear`, and `cow` classes. You may find the `Direction` and `World` packages helpful for this task. Each classes should move in response to move events based on its `inv_speed`, if relevant. See the table for descriptions of how each class should move.

Class	Movement direction
Bee	Random direction
Cow	Towards the hive with probability `2/World.size`, otherwise random direction
Bears	Towards the hive

[1] See the note about `self#register_handler` in the General tips section of this writeup.

You should pass the hive as an argument to the `cow` and `bear` objects so they can know about the hive's position. Fix `Main.ml` to account for these additional arguments to the constructor calls `new cow` and `new bear`.

After compiling, execute `./Main.byte part2` again and watch the objects wiggle around. Remember that you can press 'f' to make the simulation go faster, 's' to go slower, and 'space' to pause.

2.1. **Summary task list for part 2:**

- Make `bee`, `bear`, and `cow` inherit from `movable`.
- Movable objects should move in a single direction once every `inv_speed` fires of `World.move_event` (or never, if `inv_speed` is `None`).
- Make the bee, bear, and cow move in response to `World.move_event` as described above.
- Both `./Main.byte part1` and `./Main.byte part2` should compile and run with expected behavior.
- Using `git`, commit and push your working solution so that you have a checkpoint and a backup to which to return. After commiting, running `git status` should report that you have *no* untracked changes to any `.ml` files. You will be happy that you have used version control.

3. Actions

For this part of the assignment, you should only need to implement new methods in locations that are marked `(* ### TODO: Part 3 Actions ### *)`.

3.1. **Introduction.** Execute `./Main.byte part3`. The world has now been populated with spawns of flowers and ponds and 20 bees have been placed in the center of the world. We want these objects to start interacting with each other through the `world_object_i` interface.

3.2. **The bee.** A Bee contains a list of pollen, which is used later on for cross-pollinatin of flowers. A Bee also has a `deposit_pollen` method and a `receive_pollen` method. Its action has the effect of attempting to exchange pollen with all of its neighbors, which are any object at the same location. Relevant code is marked with `Part 3` comments.

Right now, the Bee never performs its actions. Every time an action event is fired in the world, the Bee should respond by performing its actions.

3.3. **The flower.** Flowers slowly produce pollen and offer it to visiting bees. Flowers also reproduce to bloom new flowers. We've provided some methods that would carry out these actions for you. Again, right now, they don't respond to the world!

Your task is to change the Flower so that it responds to actions in the World.

3.4. **The hive.** Hives slowly produce pollen like flowers and accept honey from worker bees. *It should be noted that for a hive, pollen and honey are the same resource. Collecting pollen from bees and losing honey to a bear manipulate the same value.*

We've implemented the majority of the Hive's functionality for you, but we made a mistake![2] To allow the bear to steal honey from the hive, we tried to expose a method called `forfeit_honey`. We failed.

Change the method `forfeit_honey` so that the bear can successfully steal honey from the hive. This will require changing the hive and its interface so that the `forfeit_honey` method is exposed to other objects.

3.5. **The bear.** Bears go directly to the hive and steal honey from it.

As with the other objects, bears don't currently follow directions. Make them respond to the world's action events.

Additionally, bears don't currently steal from the hive. They steal nothing. Using the hive method that you just exposed, make the bear steal from the hive.

Make a bear contain the amount of honey which it has stolen. The bear should update its `stolen_honey` to include the value returned by `hive#forfeit_honey`.

[2] On purpose, this time.

You can assume that the hive that is passed to the bear as an argument is the only hive in the world.

3.6. **The cow.** A cow walks around randomly eating objects that smell like pollen. You've already done the walking around part! We provide the eating part for you, but right now, the cow doesn't do what it's told. Make the cow respond to action events.

3.7. **Putting it all together.** Your simulation should now look something like this: Part 3 Final

3.8. **Summary task list for part 3:**

- Bee, Flowers, Hives, Bears, and Cows should all respond to action events.
- Hives should expose a `forfeit_honey` method in their public interface.
- Bears should steal honey from the hive, using this `forfeit_honey` method.
- Each command `./Main.byte part1`, `./Main.byte part2`, and `./Main.byte part3` should compile and run with expected behavior.
- Using `git`, commit and push your working solution so that you have a checkpoint and a backup to which to return. After commiting, running `git status` should report that you have *no* untracked changes to any `.ml` files. You will be happy that you have used version control.

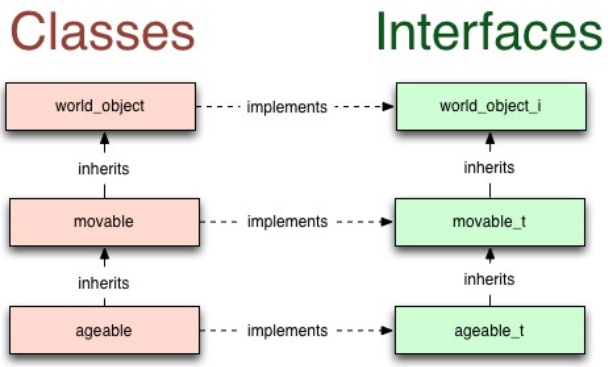
4. Aging

For this part of the assignment, you should only need to implement new methods in locations that are marked `(* ### TODO: Part 4 Aging ### *)`.

4.1. **Introduction.** Execute `./Main.byte part4`. The initial bees, bear, and cow have been removed so you should just see a world of multiplying flowers. In this part of the assignment we are going to make the hive generate new bees, and then make both bees and flowers slowly die over time. Will the bees harvest pollen quickly enough to keep both the flower and bee population alive?

4.2. **Bee production.** Before we make bees have a limited life span, we need to make the hive continually produce bees from its pollen storage. In `Hive.ml`, make a private method in the `Helper Methods` section called `generate_bee` which creates a new bee at the same position as the hive. Augment the `do_action` method in the `Method Handlers` section to decrement its pollen amount by `cost_of_bee` and call `self#generate_bee` with probability `(1/spawn_probability)`. Naturally, you should never spawn a bee if there is less than `cost_of_bee` pollen in the hive.

Execute `./Main.byte part4` and watch the ever growing population of bees and flowers.



4.3. **Ageable.** Rather than implement aging separately for each object, you will add the aging functionality to a common class `ageable` and have the `bee` and the `flower` inherit from this class (just like you did for `movable`).

The `ageable` class takes two parameters: the lifetime to create the object with, and the maximum lifetime the object can possibly have. The maximum lifetime is supplied to support the ability for an object to 'heal', which means resetting its lifetime to the maximum value.

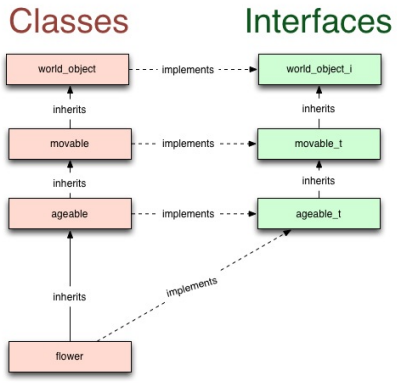
An `ageable` object maintains its current life in a mutable variable which is initialized to `starting_lifetime`.

The world manages time for the simulation, and it even tells objects how to age. Add a listener for the `World.age_event`, and have that listener call a method `self#do_age`. `self#do_age` should decrement the life of an object, if it is greater than 0, and call `self#die` if the decrement results in a lifetime of 0.

We want ageable objects to *automatically* draw life bars over their regular pictures. We can't just override the `draw` method to include a life bar because a class which inherits from `ageable` will likely want to customize its `draw` method. We need a way for derived types of ageable to specify the way to draw the picture underneath the life bar and then have the life bar automatically drawn on top. Thereare two ways to do this: the `ageable` class could expose a `draw_lifebar` method which each subtype is responsible for calling at the end of its `draw` method. Alternatively, the `ageable` class could call a separate method named `draw_picture` before it draws the lifebar in *its* `draw` method, with the expectation that derived classes override `draw_picture` and not `draw`. You will do the latter, because it is cleaner and a common pattern in object-oriented code.

The code for `draw_life` and `draw` are already given to you, but now you know why they are there. Change the definition of `draw_picture` in the `ageable` class to have a sensible default.

Implement `reset_life` to restore the lifetime of the object to its maximum lifetime.

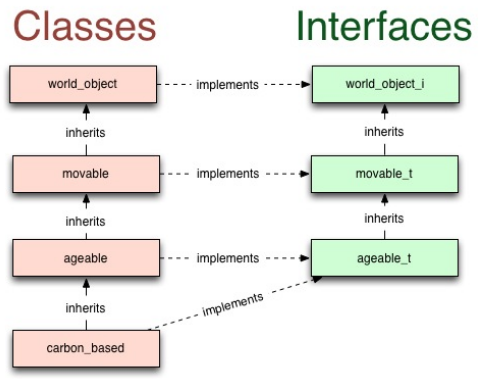


Now make the `bee` and `flower` classes inherit from `ageable`. Use (`World.rand bee_lifetime`) for the starting lifetime and `bee_lifetime` for the `max_lifetime` when inheriting from `ageable`, and likewise for `flower` and `flower_lifetime`. You will also need to alter the classes to inherit the life bar drawing as described above. Now, you should be able to run your code and watch everything die ever-so-slowly.

Now that flowers can die, you should give them the ability to live longer in the event of cross pollination. Implement `receive_pollen` in the `flower` class to

reset its life (`self#reset_life`) if the received list of pollen contains any identifier different from the flower's `pollen_id`. `flower#receive_pollen` should always return the same pollen list it was passed, and should not modify the state of pollen for offer within the flower.

4.4. **Carbon based.** You will notice that when objects die they just disappear without warning or visual ceremony. Let's change this so that objects leave behind dust when they die.



First, let's look at the `dust` class in `Dust.ml`. Notice it inherits from `ageable`! If you were wondering why we didn't just make all `ageable` objects leave behind dust when they die, this is the reason. We want dust to eventually go away and we don't want it leaving behind more dust. Alter the `dust` class to draw a circle with color `(Graphics.rgb 150 150 150)` and the first two letters of the dead objects name in black text. Note that the dust should also display a life bar.

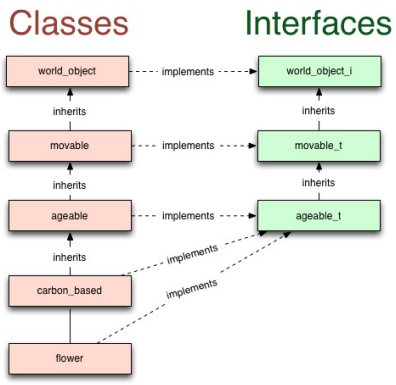
Now that we have a reasonable `dust` class, alter the `carbon_based` class to leave behind dust when it dies. Do this by registering a listener to `self#get_die_event` in the initializer of `carbon_based`. The listener should spawn a `dust` object at the same location as the `carbon_based` object, and it should pass the dead object's name to the `dust` object to be displayed.

Finally, change the `bee` and `flower` classes to leave behind dust by inheriting from `carbon_based`.

Your simulation should now look like this, with bees and flowers dying all over the world: Part 4 Final

Notice that the bees are not harvesting pollen quickly enough to sustain the bee population, and the lack of pollination leads to the flowers eventually dying off. You will fix this in the next section by making smarter bees.

4.5. **Summary task list for part 4:**

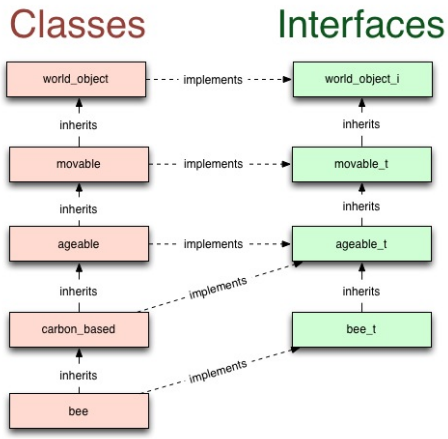


- The hive should generate bees from pollen as described above.
- `ageable` should react to `World.age_event` by aging, as described above.
- `CarbonBased` should leave behind dust when it dies.
- Dust should inherit from `ageable`.
- Bees and flowers should:
  - Inherit from `carbon_based`
  - Eventually die
  - Display a life bar
  - Leave behind dust when they die
- Flowers should reset their life in the event of cross-pollination.
- Each command `./Main.byte part1`, `./Main.byte part2`, `./Main.byte part3`, and `./Main.byte part4` should compile and run with expected behavior.
- Using `git`, commit and push your working solution so that you have a checkpoint and a backup to which to return. After commiting, running `git status` should report that you have *no* untracked changes to any `.ml` files. You will be happy that you have used version control.

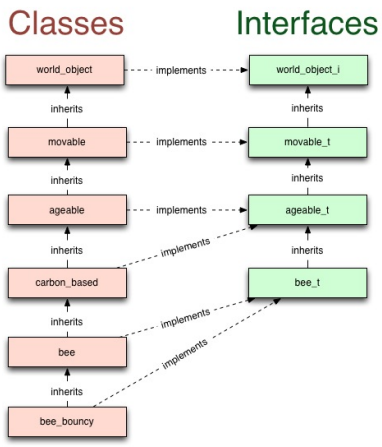
5. Smart bees

For this part of the assignment, you should only need to implement new methods in locations that are marked `(* ### TODO: Part 5 Smart Bees ### *)`.

Execute `./Main.byte part5`; it should look the same as `./Main.byte part4` because it is exactly the same setup. In this section you will give the bees some smarts so they are better at finding pollen and returning it to the hive. Because not all bees are created equal, you will be creating a range of bees of varying smartness.



5.1. **Bee types.** The first step to making multiple types of bees is to pack the common functionality into a single class to be extended and specialized. (Are you seeing a pattern?) You will use the existing `bee` class as the class to specialize, with `bee_bouncy` and `bee_random` inheriting from `bee` as derived types.



Alter the bee class so that it is declared with this signature:

```
class type bee_t =
object
  inherit Ageable.ageable_t

  method private next_direction_default : Direction.direction option
end
class bee p (home:world_object_i) : bee_t = object(self)
```

and give next_direction_default the sensible default value of None in the Bee Methods section of Bee.ml. Now make the next_direction method (which defines the movement direction for the movable class) dispatch directly to the next_direction_default method. (this will change as we put more common movement logic in Bee.ml.)

Now that you have made bee a class worth inheriting, modify bee_bouncy in BeeBouncy.ml and bee_random in BeeRandom.ml to inherit bee and have the following behavior:

- A bouncy bee will travel in a single direction until it can no longer move in that direction, at which point it moves in a random unobstructed direction. You may find Direction.move_point and World.can_move useful to implement this.
- A random bee moves in a random direction.

Also make bee_bouncy have the name bee_bouncy and bee_random have the name bee_random. Now that we have two bee classes, change the definition of generate_bee in Hive.ml to spawn either a bouncy_bee or random_bee with equal probability. The hive should pass itself as the home argument to the bees that it creates.

Execute ./Main.byte part5 and watch your random and bouncy bees fly (randomily and bouncily) around.

5.2. **The smarts.** Now we need to make bees smart. This means making bees sense flowers which are close by and return home when enough pollen has been collected. Add the following lines to the Instance Variables section of Bee.ml:

```
val sensing_range = World.rand max_sensing_range

val pollen_types = World.rand max_pollen_types + 1
```

sensing_range is how many squares away a flower can be sensed by a bee, and pollen_types is how many different types of pollen a bee will collect before it returns home. Because we have initialized these with random values, created bees will vary in their ability.

Write a (private) method called `magnet_flower` in the `Helper Methods` section for `Bee.ml`.

```
method private magnet_flower : world_object_i option = ...
```

`magnet_flower` should only return flowers which are within `sensing_range` from the bee, and **only flowers which smell like pollen that the bee has not already collected**. The returned flower should also be the closest flower which meets this criteria. The sensing radius is non-euclidian: any direction is considered to be of distance 1. For example, a bee at location `(5,5)` with `sensing_range 2` should consider all points in the square with corners `(3,3)` and `(7,7)`. You may find `World.objects_within_range` a useful function.

Now that you have a way to sense flowers nearby to a bee, change the definition of `next_direction` to have the following behavior:

- If the bee has collected more than `pollen_types` unique pollen identifiers, then go back to the hive.
- Otherwise, if the bee can sense a magnet flower nearby, go towards it.
- Otherwise, go in the `next_default_direction`.

Now execute `./Main.byte part5` and your bees should be smart enough to maintain a healthy increasing population of bees and flowers. Part 5 Final

5.3. **Summary task list for part 5:**

- The hive should spawn a `bouncy_bee` or a `random_bee` with equal probability.
- All bees should return home when they have collected enough unique types of pollen. Failing that, they should go towards a nearby magner flower if there is one. Failing *that*:
  - A `bouncy_bee` should travel in a single direction as described above.
  - A `random_bee` should travel randomly as described above.
- Each command `./Main.byte part1`, `./Main.byte part2`, `./Main.byte part3`, `./Main.byte part4`, and `./Main.byte part5` should compile and run with expected behavior.
- Using `git`, commit and push your working solution so that you have a checkpoint and a backup to which to return. After commiting, running `git status` should report that you have *no* untracked changes to any `.ml` files. You will be happy that you have used version control.

6. Custom events

For this part of the assignment, you should only need to implement new methods in locations that are marked `(* ### TODO: Part 6 Custom Events ### *)`.

Execute `./Main.byte part6`; it should look the same as `./Main.byte part5` except the cave and pasture have been placed back in the world. In this section you will make the cow and bear come out of their hideouts when they sense an abundance of honey and flowers respectively, and you will make the bees react to the bear stealing honey by chasing after and trying to sting it.

6.1. **The hive.** To make the cave spawn a bear in reaction to collecting pollen, we will give the hive a pollen event which fires every time it collects pollen.

Add the following method declarations to the hive's interface:

```
method get_pollen_event : int Event.event

method get_pollen : int
```

and give them implementations in the `Hive Methods` section of `Hive.ml`. You should *not* implement `get_pollen_event` by returning `(Event.new_event ())`, as this will return a new event stream every time. Rather, create a single event object once during the instantiation of a `hive` object and expose it with `get_pollen_event`. `get_pollen` should expose the pollen (or honey) in the hive. Finally, make a hive fire its pollen event after it receives pollen.

6.2. **The cave.** Now make the cave listen to the pollen event of a hive (the hive should be passed in as a constructor argument) and spawn a bear if 1) the pollen in the hive is above `spawn_bear_pollen` and 2) there is currently no bear attacking the hive (or alive at all). When a cave spawns a bear it should be at the same location as the cave and you should print 'omg bears! ' (note the space at the end) followed by a call to `flush_all`.

Finally, allow the bear to deposit honey in the cave by implementing `receive_pollen` in `Cave.ml` to consume all offered pollen.

6.3. **The pasture.** Make the pasture react to an action event by calling `self#do_action` which you should define in the `Event Handlers` section. In `Pasture.ml`, `do_action` should create a new `cow` object on its same location if 1) there are more than `smelly_object_limit` objects which smell like pollen in the world and 2) there is currently no cow out grazing on flowers (or alive at all). You may find `World.fold` helpful. When a pasture spawns a cow you should print 'mooooooooo ' (there are 9 'o' letters, note the space at the end) followed by a call to `flush_all`.

6.4. **The bear.** Change the bear as follows:

- A bear should have an instance variable called `life` which is initialized as `starting_life`
- The bear should take an additional object argument indicating its home spot (the cave). You will need to modify several functions in `Main.ml` so that the program will still compile and run correctly; feel free to pass in dummy arguments to new bears where necessary.
- In response to an action event, a bear should deposit its honey into the cave if it and the cave are neighbors. If the hive has less than (`pollen_theft_amount/2`) pollen after the bear deposits its honey in the cave, the bear should go back into the cave. You should call `self#die` to signify going back into the cave.
- If stung, a bear should lose 1 life. If this results in 0 life, the bear should die.
- If the bear has no stolen honey it should move towards the hive, otherwise it should move towards the cave.

6.5. **The cow.** Change the cow as follows:

- A cow is considered still hungry if it has consumed fewer than `max_consumed_objects` number of objects.
- A cow should only eat neighbors that smell like pollen if it is still hungry.
- In response to an action event, a cow should return inside the pasture (implemented as `self#die`) if it and the pasture are neighbors and the cow is no longer hungry.
- As with the bear, the cow should now take an additional argument representing its home, and you will have to make any necessary changes to `Main.ml`.
- If the cow is no longer hungry it should move towards the pasture.

6.6. **The bee.** Change the bee as follows:

- A bee should respond to a danger event from the hive by moving towards the offending object and stinging it once they become neighbors. Stinging should take place only in response to an action event.
  - Bees should only move in response to a `World.move_event` and they should only move one space.
  - If the object which attacked the hive dies before the bee gets a chance to sting it, the bee should go about its normal business.
- Stinging an objects consists of calling `receive_sting` on that object and then making the bee die.

6.7. **Putting it together.** All code should still compile without any warnings. Make sure you pass the right constructor arguments to all objects in `Main.ml`. All `./Main.byte partN` commands should still work and yield similar results to what you saw previously (minor differences, like the presence of life bars, are OK).

Congratulations, you're finished! Execute `./Main.byte part6` and enjoy your completed simulation that looks like this: Part 6 Final

6.8. **Summary task list for part 6:**

- The hive should expose a pollen event which gets fired when it receives pollen.
- The cave should spawn a bear and accept pollen deposits as described above.
- The pasture should spawn a cow as described above.
- The bear should:
  - Move and deposit honey as described above
  - Lose life and potentially die when stung
- The cow should move as described above
- The bee should react to the hive being in danger and try to sting the adversary as described above.
- The program should compile without warnings, and each command `./Main.byte part1`, `./Main.byte part2`, `./Main.byte part3`, `./Main.byte part4`, `./Main.byte part5`, and `./Main.byte part6` should run with expected behavior.
- Using `git`, commit and push your working solution so that you have a checkpoint and a backup to which to return. After commiting, running `git status` should report that you have *no* untracked changes to any `.ml` files. You will be happy that you have used version control.

7. Karma

This part is optional, and only for karma!

- Send Sam an email wishing him a happy 21st birthday. He was born on April 18th, a.k.a. the due date of this problem set!
- Add at least two new actors to the world. These actors should interact with each other and at least one of the objects used for the assignment.
- Get the world to draw actual images rather than colored circles.
- Add more user interaction. For example, you might implement a status panel that displays more information about an object when the mouse is hovered over that object.
- Do anything else fun or interesting with the world.

IMPORTANT: If you implement Karma you should submit a file named `Karma.txt` which describes your Karma extension. You must also allow for *all* karma extensions to be disabled for the purposes of grading the core functionality of the assignment.

8. Submit!

To submit the problem set, follow the instructions found here. Please note that only one of your partners needs to submit the problem set. That's it for Problem Set 7!