Week 13 - Monday

COMP 1800

Last time

- What did we talk about last time?
- Work time for Assignment 8
- Before that:
  - More class examples
  - Solar system

Questions?

Assignment 9

Simulation

Continuous simulations

- The example we did of the solar system was a simulation
  - Using (totally unrealistic) physics
- Those kinds of simulations can be useful for scientists trying to model behavior
- Real simulations are much more complex
  - Important example: weather forecasting
- These kinds of simulations are **continuous simulations** because they show the system evolving continuously as time goes on

Discrete event simulations

- Discrete event simulations are another kind of simulation
- In these, events happen at particular times
- Then, the system progresses onward after each time step, based on what happened
- The elements of the system that can act are sometimes called **agents**
- Discrete event simulations are good for modeling situations like agents shopping, standing in line, visiting the BMV, etc.
- Another possibility is modeling an ecosystem

Ecosystem

- Our ecosystem simulation will contain fish and bears
- They will exist on a grid
- Only one creature can exist at any location on the grid
- Each turn, one creature is randomly selected to come alive and do actions
- Fish can breed, move, and die
- Bears can breed, move, eat, and die
- To model this simulation, we will create objects for the world, for fish, and for bears

`World`

- A **`World`** object knows:
  - Its maximum x and y dimensions
  - All the lifeforms present inside it
  - A grid with the locations of each lifeform
- A **`World`** object should be able to:
  - Return its dimensions
  - Add a lifeform to a specific location
  - Delete a lifeform
  - Move a lifeform to a new location
  - See if a location is empty
  - Return a lifeform at a specific location
  - Allow a lifeform to live for one time unit
  - Draw itself

Bear

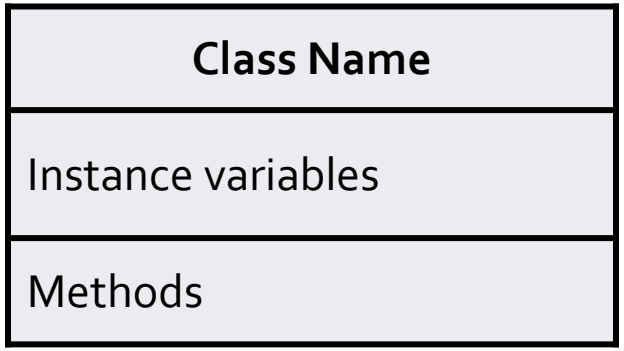
- A **`Bear`** object knows:
  - The **`World`** it's inside of
  - Its location in the world (*x* and *y*)
  - How long since it has eaten
  - How long since it has bred
- A **`Bear`** object should be able to:
  - Return its location (*x* and *y*)
  - Set the **`World`** it belongs to
  - Show up if it's been born
  - Hide if it's died
  - Change locations
  - Live for a time unit

`Fish`

- A **`Fish`** object knows:
  - The **`World`** it's inside of
  - Its location in the world (*x* and *y*)
  - How long since it has bred
- A **`Fish`** object should be able to:
  - Return its location (*x* and *y*)
  - Set the **`World`** it belongs to
  - Show up if it's been born
  - Hide if it's died
  - Change locations
  - Live for a time unit

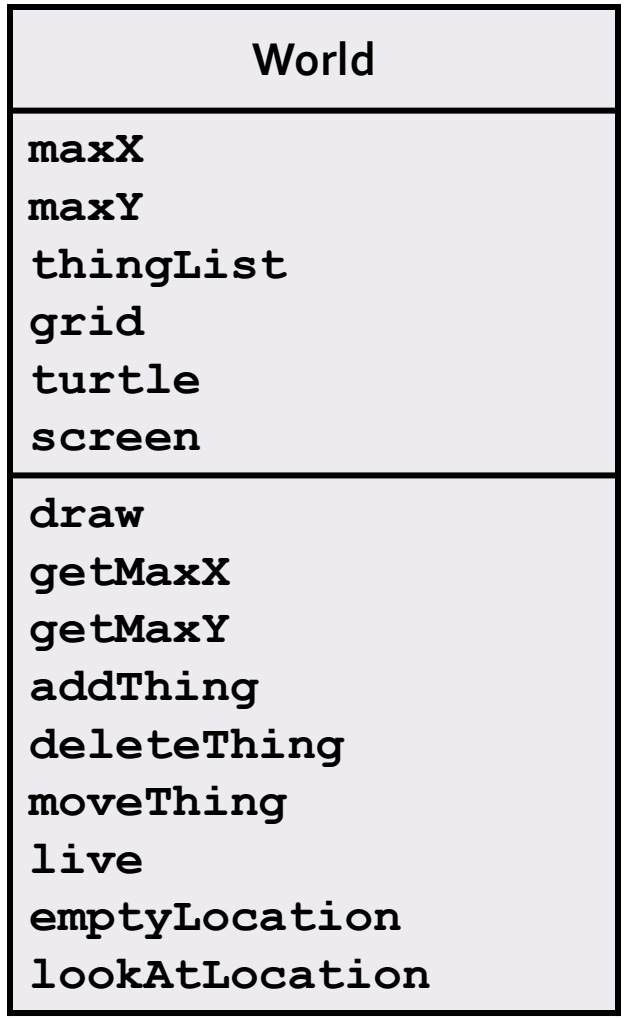
UML

- The **Unified Modeling Language** (UML) is an international standard for making diagrams of software systems
- One of the most commonly used diagrams is called a **class diagram**
- One standard for class diagrams has three sections:
  - Name
  - Instance variables
  - Methods
- To the right is an example of what that looks like



Class diagram for `World`

- Here is a UML class diagram for the **`World`** class



Class diagram for `Bear`

- Here is a UML class diagram for the **`Bear`** class

Class diagram for `Fish`

- Here is a UML class diagram for the **`Fish`** class

World Class

Implementing `World`

- Create a constructor for **`World`** with the following header:

```
def __init__(self, maxX, maxY):
```

- It should:
  - Initialize the **`maxX`** and **`maxY`** instance variables
  - Make **`thingList`** an empty list
  - Make **`grid`** a 2D list (a list of lists) containing **`maxY`** rows and **`maxX`** columns, all of which should contain **`None`**
  - Create a turtle
  - Create a screen
  - Set the screen's world coordinates to match the **`maxX`** and **`maxY`**
  - Hide the turtle

Accessors for `World`

- Write the following accessors for **`World`**

```
def getMaxX(self):
```

```
def getMaxY(self):
```

```
def emptyLocation(self, x, y):
# True if grid at y, x is empty
```

```
def lookAtLocation(self, x, y):
# Returns contents of grid at y, x
```

addThing() method for World

- Write a method with the following header:

```
def addThing(self, thing, x, y):
```

- It should:
  - Set the x and y of **thing** to the appropriate values
  - Put **thing** into the **grid** at the appropriate location
  - Set the world of **thing** to the appropriate value
  - Add the **thing** to the **thingList**
  - Tell **thing** to appear

deleteThing() method for World

- Write a method with the following header:

```
def deleteThing(self, thing):
```

- It should:
  - Hide the **thing**
  - Set the location of **thing** in the **grid** to **None**
  - Remove **thing** from **thingList**

moveThing() method for World

- Write a method with the following header:

```
def moveThing(self, oldX, oldY, newX, newY):
```

- It should:
  - Set the new location in **grid** to whatever is in the old location
  - Set the old location in **grid** to **None**

`live()` method for `World`

- Write a method with the following header:

```
def life(self):
```

- It should:
  - Check to see if there's anything in **`thingList`**
  - If there is, pick a random one
  - Tell that thing to live

draw() method for World

- To save time, here's code to draw the grid:

```
def draw(self):
        self.screen.tracer(0)
        # draw bounding box
        self.turtle.forward(self.maxX - 1)
        self.turtle.left(90)
        self.turtle.forward(self.maxY - 1)
        self.turtle.left(90)
        self.turtle.forward(self.maxX - 1)
        self.turtle.left(90)
        self.turtle.forward(self.maxY - 1)
        self.turtle.left(90)
        # draw horizontal lines
        for y in range(self.maxY - 1):
            self.turtle.forward(self.maxX - 1)
            self.turtle.backward(self.maxX - 1)
            self.turtle.left(90)
            self.turtle.forward(1)
            self.turtle.right(90)
        self.turtle.forward(1)
        self.turtle.right(90)
        # draw vertical lines
        for x in range(self.maxX - 2):
            self.turtle.forward(self.maxY - 1)
            self.turtle.backward(self.maxY - 1)
            self.turtle.left(90)
            self.turtle.forward(1)
            self.turtle.right(90)
        self.screen.tracer(1)
```

`Fish` Class

Implementing `Fish`

- Create a constructor for **`Fish`** with the following header:

```
def __init__(self):
```

- It should:
  - Create a turtle
  - Put the turtle's tail up
  - Hide the turtle
  - Set the turtle's shape to a triangle (since we don't have cool bear and fish pictures like the book does)
  - Set the **`x`** and **`y`** to 0
  - Set the **`world`** to **`None`**
  - Set the **`breedTick`** to 0

Accessors for `Fish`

- Write the following accessors for **`Fish`**

```
def getX(self):
```

```
def getY(self):
```

Mutators for `Fish`

- Write the following mutators for **`Fish`**

```
def setX(self, x):
```

```
def setY(self, y):
```

```
def setWorld(self, world):
```

```
def appear(self): # move turtle to x and y and show
```

```
def hide(self): # hide turtle
```

`move()` method for `Fish`

- Write a method with the following header:

```
def move(self, newX, newY):
```

- It should:
  - Tell world to move a thing from the current x and y to the new ones
  - Set the **`x`** and **`y`** values to the new ones
  - Move the turtle the new location as well

Upcoming

Next time…

- **`Bear`** class
- Adding behaviors to **`Fish`** and **`Bear`** objects
- The **`isinstance()`** function

Reminders

- Work on Assignment 9
  - **Due Friday**
- Keep reading Chapter 11