COMP1040

Programming Fundamentals

Assignment 2

Prepared by
Michael Ulpen

August 2017

CONTENTS

INTRODUCTION

This document describes Assignment 2 for Programming Fundamentals. The assignment will require you to write classes to represent Animals and Plants in an ecosystem simulator, called EcoSim. This will be a graphical application, utilizing your own classes and code with help from a provided game engine. You will be required to demonstrate the use of arrays, inheritance, polymorphism and exception handling.

This document outlines how to use the game engine. Documentation for the game engine may be found in Javadoc format, written as comments in the source code and organized into html files in the doc folder. This document also provides instructions for how to complete the ecosystem simulator.

This assignment is an **individual task** that will require an **individual submission**. **You will be required to submit this project in week 12 of the study period.** In the case that you do not fully understand the assignment, please ask your lecturer.

GAMEENGINE

The assignment files include a GameEngine which will automatically handle graphics, input and object updates. Part of the assignment is learning how to use the provided GameEngine to create a graphical application. There are many classes in the GameEngine and many methods in those classes. You will need to read the source code documentation to get a full understanding of how to use them.

Provided Classes

The following is a summary of some of the important classes that have been provided…

engine.GameEngine

GameEngine manages a resizable array of IPaintable, IUpdatable and ICollidable objects. It also manages the Keyboard, Mouse, GraphicsWindow and Game objects. All methods in the GameEngine are static, which means they can be accessed from anywhere without constructing a GameEngine object.

The following is a list of the most important static methods in the GameEngine:

- GameEngine.loadGame(Game g)
  Calls that game's load method and makes the window visible. It then starts the GameEngine update and paint loops.
- GameEngine.add(Object o)
  If the object is an IPaintable, ICollidable and/or IUpdatable, it will be added to the GameEngine's array and will automatically be painted and updated.
- GameEngine.addToBackground(IPaintable p)
  If the object is an IPaintable, ICollidable and/or IUpdatable, it will be added to the GameEngine's background and will automatically be updated and painted behind other IPaintables.
- GameEngine.remove(Object o)
  If the object has been added to the GameEngine it will be removed.
- GameEngine.getGameObjs()
  Returns all game objects in the GameEngine's array.
- GameEngine.getGameObjs(String classPath)
  Returns all objects in the GameEngine's array that are instances of the argument classPath. The classPath must be fully qualified, including the package path. For example, getGameObjs("ecosim.entity.Wombat") will return all the Wombat objects in the GameEngine.

engine.interfaces.IPaintable

The IPaintable interface has one method, paint(Graphics g). The Entity and Tile classes implement IPaintable and already have a suitable paint method.

engine.interfaces.IUpdatable

The IUpdatable interface has one method, update(long millisElapsed). Entity implements the IUpdatable interface and you will have to write your own update method. This method will be called automatically by the GameEngine 60 times per second.

engine.Game

Game is an abstract class including the following abstract methods:

- loadKeyFunctions()
  Creates executable functions and binds them to Keyboard keys.
- loadMouseFunctions()
  Creates executable functions and binds them to Mouse buttons.
- loadImages()
  Calls ImageLibrary.load(String alias, String imagePath) for each image in the assets folder.
- getWinWidth()
  Returns the width of the game window.
- getWinHeight()
  Returns the height of the game window.
- update(long millisElapsed)
  Updates the state of the game. Any code that should be executed 60 times a second should be added here.

The above methods will have to be implemented in any class that derives from Game. Game has one non static method, load(), which will automatically call loadImages(), loadMouseFunctions() and loadKeyFunctions().

engine.math.Vector2D

A Vector2D contains x and y coordinates. It can be interpreted as a point in space or a direction with an origin. The Vector2D contains the following methods:

- add(Vector2D other)
  Adds the argument x to this x and the argument y to this y.
- subtract(Vector2D other)
  Subtracts the argument x from this x and the argument y from this y.
- scale(float scale)
  Multiplies both the x and y by the argument scale.
- normalize()
  Divides the x and y components by the Vector's length. The result is a Vector with a length of 1, also known as a unit vector.
- distance(Vector2D other)
  Returns the distance between this Vector and the argument Vector.

engine.math.BoundingBox

A BoundingBox contains a position, as a Vector2D, and a width and height. The position is the top left point of the box. It contains the following methods:

- move(Vector2D movement)
  Adds the argument vector to this object's position.
- distance(BoundingBox other)
  Returns the distance between this BoundingBox's position and the argument's position.
- contains(Vector2D point)
  Returns true if the given point is inside this BoundingBox.
- intersects(BoundingBox other)
  Returns true if any corner point in the argument is inside this BoundingBox.

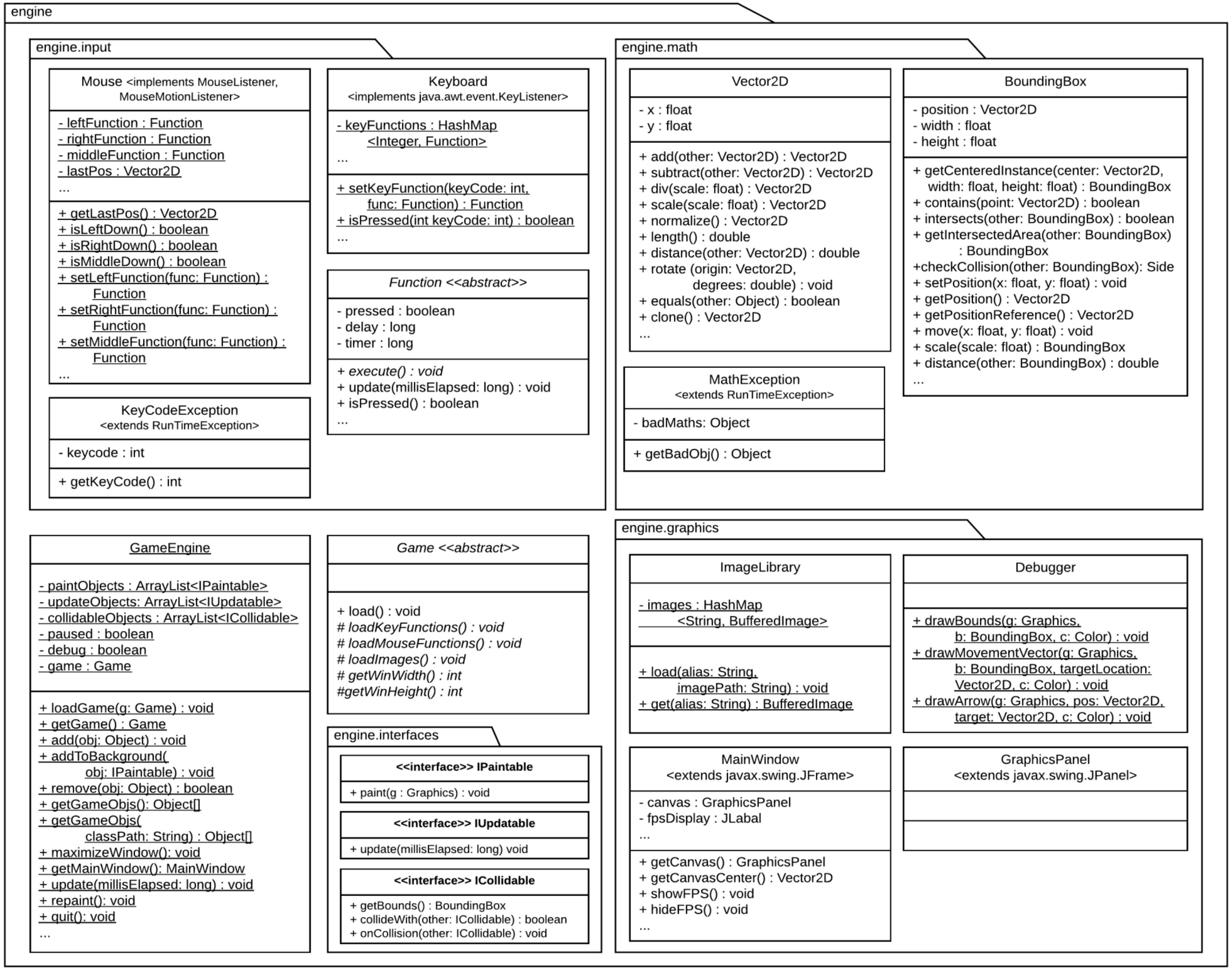
UML Diagram

The following UML diagram is a summary of most of the classes and methods in the engine package.

You will need to read the source code documentation to get a full understanding of each class and the methods therein.

Note that any underlined methods are static and any methods in italics are abstract.

Note that this diagram does not follow the rules of UML exactly. If you are confused by the diagram please ask your lecturer.


engine
engine.input
Mouse <implements MouseListener, MouseMotionListener>
- leftFunction : Function
- rightFunction : Function
- middleFunction : Function
- lastPos : Vector2D
...
+ getLastPos() : Vector2D
+ isLeftDown() : boolean
+ isRightDown() : boolean
+ isMiddleDown() : boolean
+ setLeftFunction(func: Function) : Function
+ setRightFunction(func: Function) : Function
+ setMiddleFunction(func: Function) : Function
...
Keyboard <implements java.awt.event.KeyListener>
- keyFunctions : HashMap <Integer, Function>
...
+ setKeyFunction(keyCode: int, func: Function) : Function
+ isPressed(int keyCode: int) : boolean
...
Function <<abstract>>
- pressed : boolean
- delay : long
- timer : long
+ execute() : void
+ update(millisElapsed: long) : void
+ isPressed() : boolean
...
KeyCodeException <extends RunTimeException>
- keycode : int
+ getKeyCode() : int
engine.math
Vector2D
- x : float
- y : float
+ add(other: Vector2D) : Vector2D
+ subtract(other: Vector2D) : Vector2D
+ div(scale: float) : Vector2D
+ scale(scale: float) : Vector2D
+ normalize() : Vector2D
+ length() : double
+ distance(other: Vector2D) : double
+ rotate (origin: Vector2D, degrees: double) : void
+ equals(other: Object) : boolean
+ clone() : Vector2D
...
BoundingBox
- position : Vector2D
- width : float
- height : float
+ getCenteredInstance(center: Vector2D, width: float, height: float) : BoundingBox
+ contains(point: Vector2D) : boolean
+ intersects(other: BoundingBox) : boolean
+ getIntersectedArea(other: BoundingBox) : BoundingBox
+checkCollision(other: BoundingBox): Side
+ setPosition(x: float, y: float) : void
+ getPosition() : Vector2D
+ getPositionReference() : Vector2D
+ move(x: float, y: float) : void
+ scale(scale: float) : BoundingBox
+ distance(other: BoundingBox) : double
...
MathException <extends RunTimeException>
- badMaths: Object
+ getBadObj() : Object
GameEngine
- paintObjects : ArrayList<IPaintable>
- updateObjects: ArrayList<IUpdatable>
- collidableObjects : ArrayList<ICollidable>
- paused : boolean
- debug : boolean
- game : Game
+ loadGame(g: Game) : void
+ getGame() : Game
+ add(obj: Object) : void
+ addToBackground(obj: IPaintable) : void
+ remove(obj: Object) : boolean
+ getGameObjs(): Object[]
+ getGameObjs(classPath: String) : Object[]
+ maximizeWindow(): void
+ getMainWindow(): MainWindow
+ update(millisElapsed: long) : void
+ repaint(): void
+ quit(): void
...
Game <<abstract>>
+ load() : void
# loadKeyFunctions() : void
# loadMouseFunctions() : void
# loadImages() : void
# getWinWidth() : int
#getWinHeight() : int
engine.interfaces
<<interface>> IPaintable
+ paint(g : Graphics) : void
<<interface>> IUpdatable
+ update(millisElapsed: long) void
<<interface>> ICollidable
+ getBounds() : BoundingBox
+ collideWith(other: ICollidable) : boolean
+ onCollision(other: ICollidable) : void
engine.graphics
ImageLibrary
- images : HashMap <String, BufferedImage>
+ load(alias: String, imagePath: String) : void
+ get(alias: String) : BufferedImage
Debugger
+ drawBounds(g: Graphics, b: BoundingBox, c: Color) : void
+ drawMovementVector(g: Graphics, b: BoundingBox, targetLocation: Vector2D, c: Color) : void
+ drawArrow(g: Graphics, pos: Vector2D, target: Vector2D, c: Color) : void
MainWindow <extends javax.swing.JFrame>
- canvas : GraphicsPanel
- fpsDisplay : JLabal
...
+ getCanvas() : GraphicsPanel
+ getCanvasCenter() : Vector2D
+ showFPS() : void
+ hideFPS() : void
...
GraphicsPanel <extends javax.swing.JPanel>

EcoSim attempts to simulate the relationship between predators and prey in the environment. Each animal will have to find food to survive. If they survive for long enough they will reproduce to create more animals. Some animals eat other animals. Some animals eat plants. In this assignment our animals will be wombats and snakes. The snakes will eat the wombats, the wombats will eat grass. The wombats will breed faster than the snakes but will have to eat more to stay alive.

You will need to complete the ecosim package and add its game objects to the GameEngine. In this assignment, you will have to define five classes. You will need to define the Animal, Wombat and Snake classes. You will also need to define the EcoSim and OutOfBoundsException classes. You have been provided some complete classes to get you started. These include Main, Entity, GrassTuft, Tile, SandTile and GrassTile.

Provided Classes

The following classes have been provided for you…

ecosim.game.Main

Main is the starting point of the program. It constructs an EcoSim object and adds it to the GameEngine. Execute the main method in the Main class to start the EcoSim application.

ecosim.world.Tile

Tile contains a BoundingBox and a BufferedImage. Tile implements IPaintable so it has a paint method which will paint its image to its bounding box position. Tiles will be created and arranged into a grid. This will act as a background image and determine the area in which the Wombats and Snakes are allowed to move. Once added to the GameEngine, using GameEngine.addToBackground(tile), it will automatically be painted to the screen and will appear behind images added on top of it.

ecosim.world.GrassTile

GrassTile extends from Tile. GrassTufts are only allowed to be created in the same position as a GrassTile. GrassTiles should be added to the GameEngine background.

ecosim.world.SandTile

SandTile extends from Tile. A GrassTuft should never be allowed to grow on top of a SandTile. SandTiles should also be added to the GameEngine background.

ecosim.entity.Entity

Entity is the base class of Animal and GrassTuft. An Entity has a bounding box, a target position, an image and an alive status. The bounding box determines the Entity's position and size. Entity implements IPaintable and consequently, has a paint method which will paint the Entity's image to the screen at its position. It implements IUpdatable so it has an abstract update method, which should be overridden in its derived classes to determine how the Entity acts and how it moves. In general, the Entity will move towards its target position over time. It has a die method which, when called, will set alive to false and automatically remove the Entity from the GameEngine. Once removed from the GameEngine, an Entity will not be updated or painted to the screen.

Any class extending from Entity can be added to the GameEngine using GameEngine.add(entity). This will paint it on top of objects added to the background.

ecosim.entity.GrassTuft

GrassTuft extends from Entity and, as a result, has an image, bounding box, and alive status. It will be painted to the screen at its bounding box position. It can die, just like any Entity. It also has an update method, but the update method is empty. Wombats will find GrassTuft objects, move to their location and eat them. If a GrassTuft is eaten by a Wombat, the GrassTuft will die and be removed from the GameEngine.

Required Classes

The following classes will need to be written by you…

ecosim.game.EcoSim

EcoSim extends from the abstract class engine.Game. It must implement the abstract methods loadKeyFunctions(), loadMouseFunctions(), loadImages(), getWinWidth(), getWinHeight() and update(long millisElapsed). Most of these methods have been written for you. The class contains a gridWidth, gridHeight, timer and 2 dimensional array of Tiles. The gridWidth and gridHeight determine the length of the tile array. The timer will be used in the update method.

Complete the EcoSim constructor to save the arguments gridWidth and gridHeight to instance variables. Initialize the length of the tile array to gridHeight arrays of gridWidth length each. Implement the getWinWidth method to return gridWidth multiplied by the Tile width. Implement the getWinHeight method to return gridHeight multiplied by the Tile height.

The EcoSim load method should call super.load(). This will automatically call the loadKeyFunctions(), loadMouseFunctions() and loadImages() methods. After calling super.load(), you will need to create a two dimensional array of Tiles and add each to the GameEngine background. Generate a random number and use it to ensure that approximately 80% of the Tiles are GrassTiles and 20% of the Tiles are SandTiles. If it is a GrassTile, generate another random number and use it to ensure that 50% of the time, you create a GrassTuft object and add it to the GameEngine at the same position as the GrassTile. Generate another random number and use it to ensure that about 20% of the time, you add a Wombat at the same position as the Tile and 10% of the time, you add a Snake. There should be about half as many Snakes as Wombats.

The EcoSim update method should add a GrassTuft object to the GameEngine background every 1000 milliseconds. Add the millisElapsed argument to the timer instance variable. If the timer is greater than 1000 add a GrassTuft object in the same position as a random GrassTile that does not already have a GrassTuft in the same position. You can access the GrassTuft objects from the GameEngine using GameEngine.getGameObjs("ecosim.entity.GrassTuft"). This will return an Object array. You can type cast each item from the array to a GrassTuft and check its position is not the same as the GrassTile you have selected. If none of the GrassTufts are in its position, create the GrassTuft here. If a GrassTuft was successfully created, reset the timer back to zero.

ecosim.entity.Animal

Animal is an abstract class that extends from Entity. It has energy, breedTime, maxBreedTime, speed and a timer. The energy starts at a high number but, if it reaches zero, the Animal will die. The breedTime will start at maxBreedTime and when it reaches zero, the Animal will breed to create a copy of itself and add it to the GameEngine. The breedTime will then be reset to maxBreedTime. The speed determines how far the Animal can move each step. The timer will add up towards 1000. Every time it reaches 1000, it will decrease the energy by 1 and decrease the breedTime by 1. The timer should then be reset to zero.

You will need to complete the constructor, provide getters and setters and implement the update method. In the constructor, initialize the instance variables to the given arguments and initialize the timer to zero. Also call the abstract selectTarget method. Add getters and setters for each of the variables except for the timer. Ensure that the setters only change the variables if the argument is greater than or equal to zero.

The Animal update method should add the millisElapsed to the timer. If the timer is greater than 1000, the timer should reset to zero and the energy and the breedTime should decrease by one. If the energy is zero or less, the animal should call the die() method. Otherwise, if the breedTime is less than or equal to zero, the Animal should call the breed method and add the result to the GameEngine. If the Animal is still alive, the animal should move towards its target position.

The Vector math for moving the Animal is provided below:

```
        Vector2D direction = target.clone().subtract(this.getPosition());
        direction.normalize().scale(this.speed);
        this.getBounds().move(direction);
```

These diagrams demonstrate how the above Vector mathematics works:

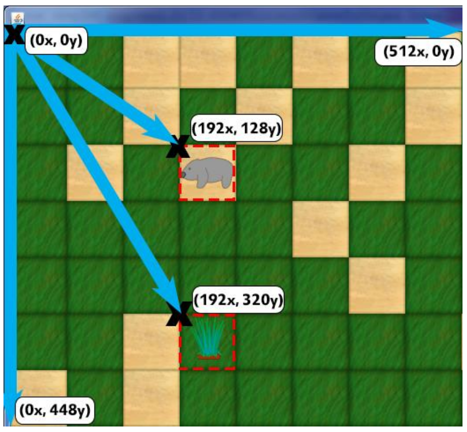


Figure 1: Position of Wombat and GrassTuft relative to (0, 0).

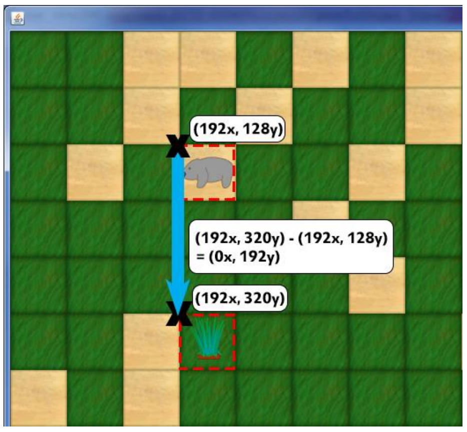
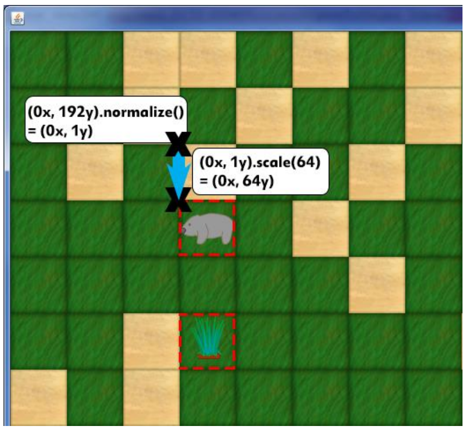


Figure 2: Use subtract to get position of GrassTuft relative to Wombat.



(0x, 192y).normalize() = (0x, 1y)
(0x, 1y).scale(64) = (0x, 64y)

Figure 3: Use normalize() to scale the Vector to length 1. Use scale() to multiply the length by speed. Move the Wombat by the result.

ecosim.entity.Wombat

Wombat will extend from Animal and have an instance variable, called targetGrass, containing a GrassTuft target. By default this GrassTuft should be null. The Wombat constructor will have to send its position and its default image, energy, breed steps and movement speed to the super constructor. You can set your energy to 10, breedSteps to 18 and speed to 1. The Wombat will need to implement the breed method and selectTarget method. It will also need to override the update method.

The breed method should return a Wombat at the same position as the calling object. The update method should call super.update(). Provided the Wombat is not dead, it should then find the distance between its position and its target vector. If it is less than half the width of a tile, then the Wombat has reached its destination. It should set target to null. If the Wombat has a target GrassTuft, and it is within a half Tile width of that object, the Wombat should add 10 to its energy, call the target grass' die method and set its target grass variable to null. It should then call the selectTarget method.

The selectTarget method will check to see if the Wombat's energy is lower than 10 and if it is, the Wombat should find the closest GrassTuft object and set a reference to that GrassTuft's position as the Wombat's target. The GrassTuft should also be saved as the Wombat's targetGrass. If the Wombat does not have a targetGrass and it does not have a target, you should generate a new target at a random position between 0 and 150 pixels away from the Wombat's current position. If the new target's x or y coordinate is less than zero or its x coordinate is greater than the Game's pixel width or its y coordinate is greater than the Game's pixel height, you should throw an OutOfBoundsException. If an OutOfBoundsException is caught, the target vector should be constrained to between 0 and the Game's width along the x and between 0 and the Game's height along the y.

ecosim.entity.Snake

Snake will extend from Animal and have an instance variable for moveSpeed, attackSpeed and a targetWombat containing a reference to a Wombat. The Snake constructor will have to send its position and its default image, energy, breed steps and movement speed to the super constructor. You can set the energy to 16, breedSteps to 32 and speed to 0.6. Set the attack speed to 1.1 and the move speed to 0.6. You will need to implement the breed method and selectTarget method and override the update method.

The breed method should return a Snake at the same position as the calling object. The update method should call super.update(). Provided the Snake is not dead, it should set its speed to its moveSpeed. If its targetWombat is not null and the targetWombat is alive, if the distance to the Wombat is less than half a Tile width, then the Snake should gain the energy of the Wombat, the Wombat should die and the targetWombat should be set to null. Otherwise, if the distance from the Snake to the Wombat is less than 3 Tile widths, set the Snake's speed to its attackSpeed, so it can chase down the Wombat. It should then call the selectTarget method.

The selectTarget method will check to see if the Snake's energy is lower than 8 and if it is, the Snake should find the closest Wombat object and set a reference to that Wombat's position as the Snake's target. The Wombat should also be saved as the Snake's targetWombat. If the Snake does not have a targetWombat and it does not have a target, you should generate a new target at a random position between 0 and 150 pixels away from the Snake's current position. If the new target's x or y coordinate is less than zero or its x coordinate is greater than the Game's pixel width or its y coordinate is greater than the Game's pixel height, you should throw an OutOfBoundsException. If an OutOfBoundsException is caught, the target vector should be constrained to between 0 and the Game's width along the x and between 0 and the Game's height along the y.

ecosim.exception.OutOfBoundsException

The OutOfBoundsException class should extend from Exception. It should have a private final instance variable called badPosition that contains a Vector2D. The class will need a constructor that takes a Vector2D parameter and saves it in the instance variable. It should also call super with the default error message, "Out of bounds: " + badPosition. Provide a getter method to return the badPosition.

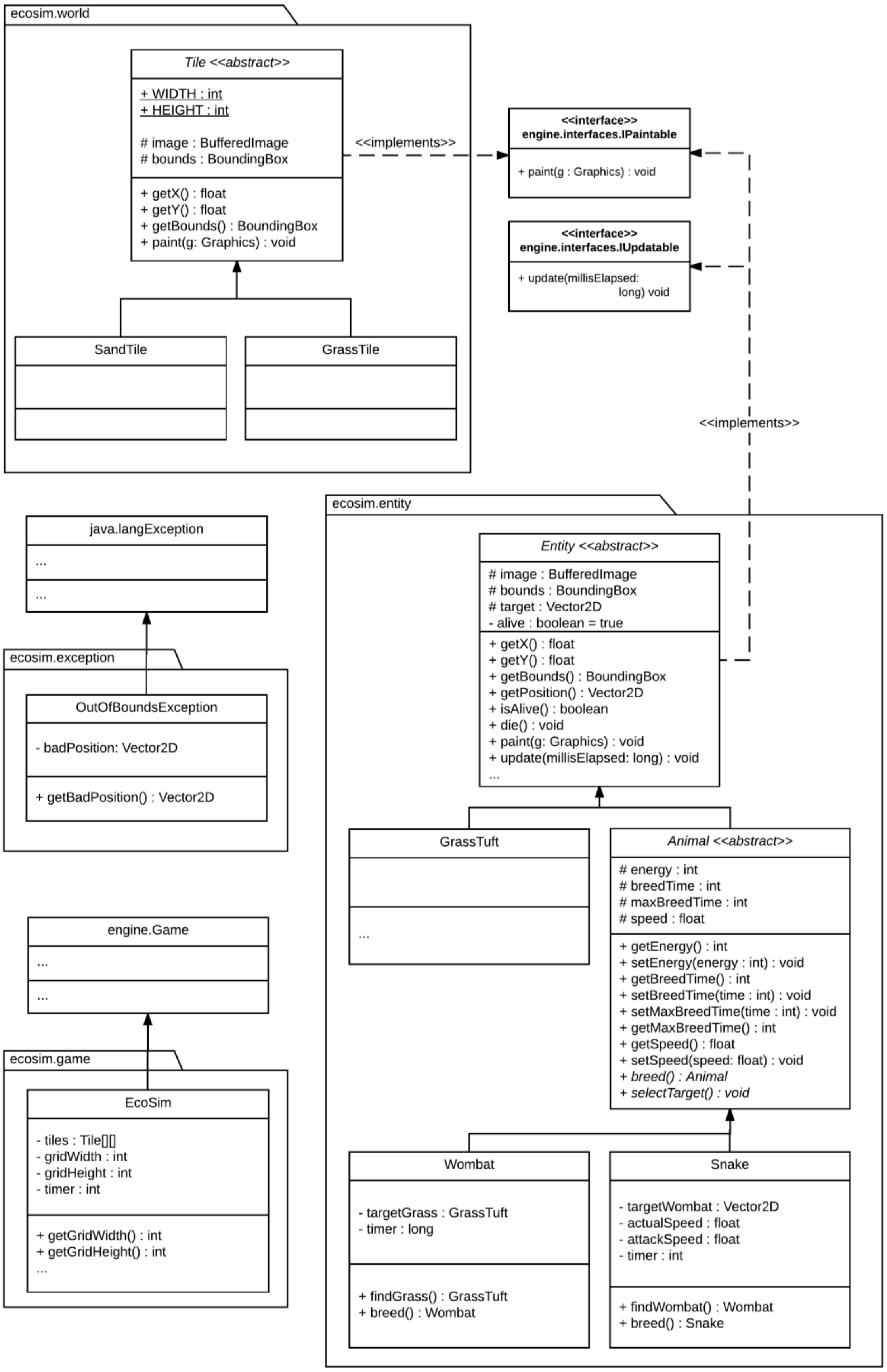
Source Code Documentation

You will be required to include source code documentation for each of the methods you write. You must place a comment between /** and */ on the line above each of the methods you write. Describe in detail what the method does, what parameters are required and what value is returned.
An example of a doc string for the EcoSim getWinWidth method is as follows:

```java
/**
 * Return the total screen width of the Game, which should be
 * the number of horizontal Tiles multiplied by the tile width.
 * @return gridWidth multiplied by default Tile width.
 */
public int getWinWidth() {
     return this.gridWidth * Tile.WIDTH;
}
```

UML Diagram

The following UML diagram is a summary of all classes in the ecosim package. Your goal in this assignment is to implement the UML Diagram in code. Note that any underlined methods are static and any methods in italics are abstract.



HOW TO START

The following are a few things you can do to test that the GameEngine works:

- Implement any unimplemented methods to remove all the compile errors.
- In the EcoSim load method, under super.load() call GameEngine.add(new Wombat(200, 300)). This should put a Wombat on the screen. Change the numbers to change the Wombat's position.
- If your window is too small, return a number from the EcoSim getWinHeight() and getWinWidth() method. Maybe return 600 from the getWinWidth() and return 400 from getWinHeight().
- In the Wombat update method call this.getBounds().move(1, 0). This should cause the Wombat to move 1 pixel to the right 60 times per second. Try changing the numbers to see what causes it to move in different directions.
- Press the tilde button (~). This should set the GameEngine debugging state to true. You should be able to see a BoundingBox drawn around the Wombat. Normally this would show the movement vector as an arrow, but you will have to follow the specifications above to get that to work. You can also try pressing the space key to pause the program.
- You may like to open the GameEngine, BoundingBox and Vector2D classes to read their code. You may also like to read the documentation in the doc folder. This should give you further understanding of how the GameEngine works.
- Once you are feeling confident return to the EcoSim Required Classes section and follow the instructions to complete the assignment.

ADDITIONS

The EcoSim and GameEngine are extendable. For full marks you must implement at least **one** of the following additions. Please only attempt these once you have completed all other requirements. If you submit an unfinished addition, please ensure that your code still compiles without errors.

- **Another Animal**: Implement a predator of Snakes or another predator of GrassTufts. The new Animal must have a new image.
- **Animation**: Implement sprite animations. In the assets/images folder, there is a second image for both Wombat and Snake. Cycle between the first and second image every 500 milliseconds to create a walk cycle.
- **Obstacle**: Implement a new Tile that Animals cannot stand on or move through. Water or Bush tiles might be appropriate. These Tiles will need new images.
- **Wombat Artificial Intelligence**: Wombats should try to move away from any nearby Snakes, unless they are hungry and the Snake has gotten in the way of their food.
- **Collisions**: Implement the ICollidable interface for Snakes and Wombats so that Snakes do not pass through other Snakes and Wombats do not pass through other Wombats.
- **Realistic Breeding**: Implement the ICollidable interface for Animals so that Animals can only breed when their breedTimer is less than or equal to zero and they have collided with another object of the same class.
- **Sound**: Add a sound system and sound files to the EcoSim, so that Snakes and Wombats make sounds whenever they eat, breed or die.
- **Paused Label**: Display a label saying PAUSED at the center of the screen whenever the GameEngine is paused.
- **Other**: If you would like to make an addition not listed above, please discuss it with your lecturer.

SUBMISSION DETAILS

You are required to submit an electronic copy of your program via Moodle. Make sure your eclipse project is included in a zip file (WinZip). The zip file should be called `yourId.zip`. For example: `bonjy1207.zip`. Ensure that your files are named correctly (as per instructions outlined in this document).

All Java files that you submit must include the following comments.

```
/**
/ File: fileName.java
/ Author: your name
/ Id: your id
/ Version: 1.0 today’s date
/ Description: Assignment 2 - ….
/ This is my own work as defined by the SAIBT
/ Academic Misconduct policy.
*/
```

This is to acknowledge this is your own individual work.

Assignments that do not contain these details may not be marked.

Work that has not been correctly submitted to Moodle will not be marked.

It is expected that students will make copies of all assignments and be able to provide these if required.

EXTENSIONS AND LATE SUBMISSIONS

There will be **no** extensions/late submissions for this course without one of the following exceptions:

1. A medical certificate is provided that has the timing and duration of the illness and an opinion on how much the student’s ability to perform has been compromised by the illness. **Please note** if this information is not provided the medical certificate WILL NOT BE ACCEPTED. Late assessment items will not be accepted unless a medical certificate is presented to the Course Coordinator. The certificate must be produced as soon as possible and must cover the dates during which the assessment was to be attempted. In the case where you have a valid medical certificate, the due date will be extended by the number of days stated on the certificate up to five working days.

2. A SAIBT councillor contacts the Course Coordinator on your behalf requesting an extension. Normally you would use this if you have events outside your control adversely affecting your course work.

3. Unexpected work commitments. In this case, you will need to attach a letter from your work supervisor with your application stating the impact on your ability to complete your assessment.

4. Military obligations with proof.

Applications for extensions must be lodged with the Course Coordinator before the due date of the assignment.

Note: Equipment failure, loss of data, ‘Heavy work commitments’ or late starting of the course are not sufficient grounds for an extension.

ACADEMIC MISCONDUCT

Students are reminded that they should be aware of the academic misconduct guidelines available from the SAIBT website.

Deliberate academic misconduct such as plagiarism is subject to penalties. Information about Academic integrity can be found in the "Policies and Procedures" section of the SAIBT website.

MARKING CRITERIA

Assessment feedback			
Programming Fundamentals (COMP1040) **Assignment 2 – Weighting: 20%**			
NAME:	**MAX MARK**	**MARK**	**COMMENT**
ECOSIM: Constructor(3), load(7), update(10)	**20**		
ANIMAL: Constructor(2), update(10), getters and setters(8)	**20**		
WOMBAT: Constructor(3), update(5), selectTarget(5), findGrass(7)	**20**		
SNAKE: Constructor(3), update(5), selectTarget(5), findWombat(7)	**20**		
OUTOFBOUNDSEXCEPTION: Constructor(3), getter(2), Exception Handling(5)	**10**		
STYLE: Doc comments (5)	**5**		
ADDITIONS*: Animation (10), extra animal(10), obstacle(10), Wombat A.I.(10), collisions(10), breeding(10), sound(10), paused label(10), other…	**5 ~ 20**		
TOTAL	**100 MARKS**		

**Note that the maximum marks without additions is 95%. An addition must be made for full marks. Additions may recover lost marks up to a maximum of 20.*

Possible deductions:

- *Programming style:* -10 marks for poor or no commenting, poor variable names, poor indentation, use of break and continue etc.
- *Submitted incorrectly:* -20 marks if assignment is submitted incorrectly.