# Ant vs. SomeBees

Projet de programmation

## 1 Introduction

In this project, you will implement a Tower Defense called Ants Vs. SomeBees. As the ant queen, you populate your colony with the bravest ants you can muster. Your ants must protect their queen from the evil bees that invade your territory. Irritate the bees enough by throwing leaves at them, and they will be vanquished. Fail to pester the airborne intruders adequately, and your queen will succumb to the bees' wrath.

This project is highly inspired by an existing assignment from John DeNero, Tom Magrino and Eric Tzeng (UC Berkeley), which was inspirated by the PopCap Games' Plants Vs. Zombie ®. The current version was adapted from the Python programming language to Scala by myself. It it provides you with less pre-written code, but gives you much more freedom about how to organize your code. Freedom may be disturbing, at first, but you'll get used to it ;)

### 1.1 Core Concepts

A game of Ants Vs. SomeBees consists of a series of turns. In each turn, new bees may enter the ant colony. Then, new ants are placed. Finally, all insects (ants, then bees) take individual actions: bees sting ants, and ants throw leaves at bees. The game ends either when a bee reaches the ant queen (you lose), or the entire bee flotilla has been vanquished (you win).

**The Colony**. The colony consists of several places that are chained together. The *exit* of each *Place* leads to another *Place*.

**Placing Ants**. There are two constraints that limit ant production. Placing an ant uses up some amount of the colony's food, a different amount for each type of ant. Also, only one ant can occupy each *Place*.

**Bees**. When it is time to act, a bee either moves to the exit of its current Place if no ant blocks its path, or stings an ant that blocks its path.

**Ants**. Each type of ant takes a different action and requires a different amount of food to place. The two most basic ant types are the *HarvesterAnt*, which adds one food to the colony during each turn, and the *ThrowerAnt*, which throws a leaf at a bee each turn.

### 1.2 Due date and grading

You are requested form teams for two people for this project, with (only) one group of three people. Nobody is allowed to work alone. You have turn your code before Sunday December 6. at noon. The defenses will be organized on Tuesday December 8. instead of the usual practical. You will have to turn in your slides on that day. Finally, you will have to send your report **and a 60-seconds long screencast** of your application before Sunday December 13. No late submission will be accepted. Kazam is a nice application to take screencasts.

## 2 Technical setup

### 2.1 scala-swing

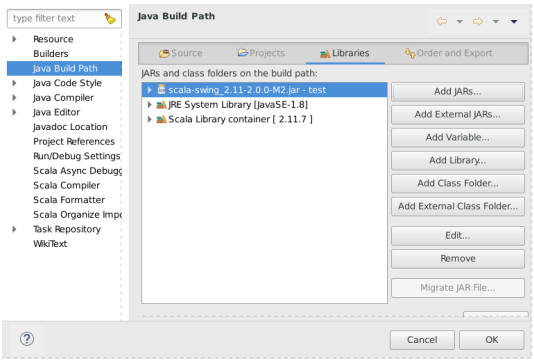
This project must be written in Scala, using the Swing library. You really should use the scala-swing library, which provides Scala wrappers to the main Java Swing classes. You can download the mandatory library from this site. If you feel brave, you can try to use ScalaFX, which will be a much better solution to build GUI in Scala. But unfortunately, ScalaFX is still under construction.

### 2.2 Editing with Eclipse

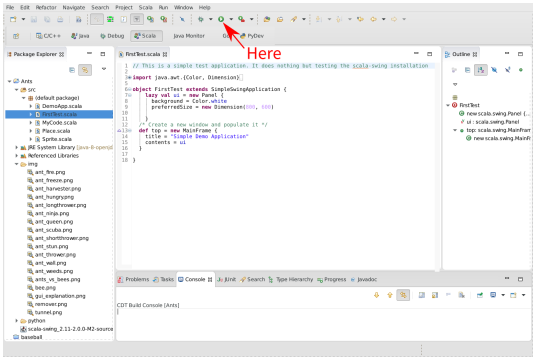
It is advised (although not mandatory) that you use the Eclipse source editor for this assignment. That's a huge machinery, but it's really worth learning. When you know it, this tool can greatly increase your code productivity. Don't miss this introductive video. Its major drawback however is its size. Eclipse is really the programming caterpilar, and you have to have a good laptop with at least 4 Gb of RAM to use it comfortably.

First, download the Scala IDE and install it on your machine, or add the its update site to your existing Eclipse installation.

Create a new project, and copy the scala-swing.jar files into its main directory. Open the project properties (from the right-click menu), and select to the "Java Build Path" entry. In the "Libraries" tab, add the scala-swing jarfile as a JAR. This should look as follows before you click OK:




Once done, the first provided code should open an empty window. To test it, download this file, and copy it in the src/ folder of your eclipse project. Then in eclipse, press F5 to see it appear in the project explorer, right click on the file and select "Run As" / "Scala application". Once you've done it one time, you can directly run the program by clicking on the little button above the code green arrow.




## 2.3 Compiling with sbt

Simple Build Tool (sbt for short) is a very pleasant tool to compile your Scala code. If you (or your computer) are alergic to Eclipse, using sbt with any decent editor (such as Geany) is probably the best solution. Simply edit in Geany and compile in a separate terminal. That's the easiest: use an editor to edit and an builder to build your code...

Download `sbt` from its webpage, unpack it somewhere (e.g. under `/opt/sbt`) and add the `bin` directory to your PATH with the following commands. This approach is better to the one described in the official sbt documentation because it does not install anything with the super-user privileges. You are asked for your root password, but only to create a new directory. No installed file have the root privilege: I never install stuff from untrusted source with the root privilege.

```
# Create the directory (needs root password)
sudo mkdir -p /opt/sbt
# Give the directory ownership to yourself
sudo chown -R ‘logname‘ /opt/sbt
# Enter that directory
cd /opt/sbt
# Unpack the archive
tar xf ~/Téléchargements/sbt-0.13.9.tgz
# Add the binary directory into your PATH
echo ’export PATH=$PATH:/opt/sbt/bin’ >> ~/.bashrc
# Reload the shell configuration
source ~/.bashrc
# Launch your code.
sbt run
# This first run will download some dependencies
```

Please see the sbt documentation for more information.

## 2.4 Compiling manually

If everything else fails, you can always compile your code from the command line as follows. You should however really try to get at least `sbt` working, because such manual compilations are really bothersome on the long term.

```
scalac -cp scala-swing.jar:. FirstTest.scala
scala -cp scala-swing.jar:. FirstTest
```

For that, you need to copy the scala-swing.jar file in your directory, under that exact name.

When your code evolves, you need to replace `FirstTest.scala` on the first line by the name of your source files. On the second line, give the name of your main class instead of `FirstTest`.

## 2.5 Code Versioning

As you will eventually realize, it is very difficult to share code with your teammates without the appropriate tools. You are perfectly free to use anything you see fit, such as USB key, email attachments or pigeons, but you probably want to use a SVN or Git.

# 3 Warm up

You are provided with an initial application called DemoApp It exemplifies everything you need to know about the Scala Swing library. Download it, and read its code. You see that the code falls in three big parts: (1) a *state* object that contains the whole game state, (2) an *ui* object: a graphical panel on which the game is drawn and which listens to the keyboard and mouse events, (3) a *timer* object for the game animation, which calls *state.update* and *ui.repaint* 50 times per second. When the application starts, a window is created at the very bottom of this file and populated with the *ui*.

## 3.1 Introducing a class Insect

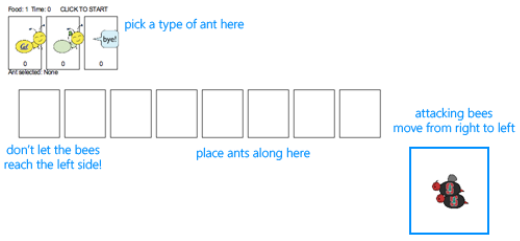
Modify the provided code to introduce a proper class named *Insect* encapsulating all of the sprite logic. It should encapsulate the position and speed of a Sprite as well as the image displayed image. It should provide the necessary methods to make the code of your *state* and *ui* clean and pleasant to read.

## 3.2 Introducing a class Place

A place is a location of the game, depicted as a cell on the board. It is defined by its name and position (its constructor takes two parameters). It can contain one or several bees and at most one ant. Each place is connected to an entrance and an exit (the entrance of its exit is itself). Define the corresponding class, and populate your game with 8 places that form a tunnel.

# 4 First game

We will first write the main game elements according to the following screenshot.




The colony is on top. This is where you select the kind of ant that you want to place on your board. The colony cell containing the word "Bye" is particular: when you select this cell and click on one of your existing ant, it is removed from the game. Under the colony, you can see one tunnel of eight places, in which you can place ants. Later in the game, we will create several tunnels forming several lines one under another. On the right, you can see the hive, containing 2 bees that will soon enter your tunnel.

## 4.1 The basics

In this first game, food is not an issue: you can create as many ants as you wish until you fill all existing Places. For that, click on the one cell of the Colony to select it, and then click on the tunnel place in which you want to add an ant of the selected type.

The game is decomposed in several turns (of three seconds each). The player can add new ants at any time during the game. At each turn, the following actions occur:

- A bee enters in the right-most place of the tunnel, unless all bees entered the game already.
- Each ant moves, depending on its type. The `HarversterAnt` adds one Food to the colony (useless for now, but soon crucial). If a bee is in the same Place, the `ThrowerAnt` removes one Armor to the bee. An insect with 0 Armor is killed.
- Each bee moves.
  - If there is no ant in its place, it moves to the next place.
  - If there is an ant in its place, it reduce its Armor by one.

If all bees are defeated, the player wins. If one bee passes the left-most place, the player loses.

**Harvester (Cost: 2; Armor: 1)**

**Long Thrower (Cost: 3; Armor: 1)**

**Thrower (Cost: 2; Armor: 1)**

**Fire Ant (Cost: 5; Armor: 1)**

### 4.2 Making a real game

Implement the food logic, where you cannot place a new ant if you don't have enough food.

Now, modify your HaversterAnt so that it shots at the nearest bee on its right. Well, it makes the game a bit too easy, so we will introduce two new kind of ants. The short-range thrower can only shot bees that are at most two entrances away while the long-range thrower can only shot bees that are at least three entrances away.

**Short Thrower (Cost: 3; Armor: 1)**

### 4.3 Graphical Refinements

The insects should move smoothly from one place to another at the end of each turn, and the leaves shot by throwers should be animated too. The player should still be able to place ants at any point of time (provided that there is enough food).

## 5 Water and Fire

### 5.1 Adding a Fire Ant

Implement the *FireAnt*. A *FireAnt* has a special *reduceArmor* method: when the *FireAnt*'s armor reaches zero or lower, it will reduce the armor of all *Bee/s in the same /Place* as the FireAnt by its damage attribute (defaults to 3).

### 5.2 Adding water to the game

Our tunnels are a bit boring as is, because all places are the same (but the hive). To make things more interesting, we're going to create a new type of Place called Water.

Only an ant that is *watersafe* can be deployed to a *Water* place. In order to determine whether an Insect is *watersafe*, add a new attribute to the Insect class named *watersafe* that is *false* by default. Since bees can fly, make their watersafe attribute true, overriding the default.

Now, implement the *addInsect* method for *Water*. First call *Place.addInsect* to add the insect, regardless of whether it is *watersafe*. Then, if the insect is not *watersafe*, reduce the insect's armor to 0 by invoking *reduceArmor*. Do not copy and paste code. Try to use methods that have already been defined and make use of inheritance to reuse the functionality of the *Place* class.

### 5.3 Adding water to the board

Change the code that create the tunnel to now create 3 tunnels of 8 places each in which every third place is water. When a bee enters the board, it picks a tunnel randomly.

You may also change your code so that bees arrive in waves at pre-determined turns (to leave some time to the player to rebuild the base).

### 5.4 Adding a Scuba ant

Currently there are no ants that can be placed on Water. Implement the *ScubaThrower*, which

is a subclass of *ThrowerAnt* that is more costly and *watersafe*, but otherwise identical to its base class.

**Scuba Thrower (Cost: 5; Armor: 1)**

# 6 More units

## 6.1 Wall Ant

We are going to add some protection to our glorious *AntColony* by implementing the *WallAnt*, which is an ant that does nothing each turn (already the default action of the *Ant* class), but presents a large armor value.

**Wall Ant (Cost: 4; Armor: 4)**

## 6.2 Ninja Ant

Implement the *NinjaAnt*, which damages all Bees that pass by, but is never seen. It cannot be attacked by a *Bee* because it is hidden, nor does it block the path of a *Bee* that flies by. To implement this behavior, add a new attribute *blocksPath* to *Ant* that is *true* by default and *false* for *NinjaAnt*.

**Ninja Ant (Cost: 6; Armor: 1)**

## 6.3 HungryAnt

We will now implement the new offensive unit called the *HungryAnt*, which will eat a random *Bee* from its place, instantly killing the Bee. After eating a Bee, it must spend 3 turns digesting before eating again.

**Hungry Ant (Cost: 4; Armor: 1)**

# 7 God saves the Queen

## 7.1 Bodyguard Ant

A *BodyguardAnt* differs from a normal *Ant* because it can occupy the same *Place* as another ant. When a *BodyguardAnt* is added to the same *Place* as another ant, it shields the other ant and protects it from damage. Attacks should damage the *BodyguardAnt* first and only hurt the protected ant after the *BodyguardAnt* has perished.

A *BodyguardAnt* has a field *ant* that stores the ant contained within the bodyguard. It should start off as *null*, indicating that no ant is currently being protected. Give *BodyguardAnt* a *containAnt* method that adds the Ant passed as a parameter inti that field.

**Bodyguard (Cost: 4; Armor: 2)**

Now, change your program so that a BodyguardAnt and another Ant can simultaneously occupy the same Place:

- Add an attribute to *Ant* indicating whether it is a container. Only *BodyguardAnt* are containers.
- Add a method to *ant* indicating whether the receiver can contain the parameter: If and only if the receiver is an empty container and the paramter is not a container.
- Change the *addInsect* method accordingly.

## 7.2 QueenAnt

The queen is a waterproof *ScubaThrower* that inspires her fellow ants through her bravery.

Whenever the queen throws a leaf, she also doubles the damage of all other ants in the same tunnel with her, including any ants protected by a bodyguard. Once any ant's damage has doubled, it cannot be doubled again.

**Queen ant (Cost: 6; Armor: 2)**

However, with great power comes great responsibility. The Queen is governed by three special rules:

- If a bee ever enters the place occupied by the queen, then the bees immediately win the game. The game ends even if the queen is protected by a bodyguard. The bees also win if any bee reaches the end of a tunnel where the queen normally would reside.
- There can be only one true queen. Any queen beyond the first one is an impostor and should die immediately (its armor reduced to 0) upon taking its first action, without doubling any ant's damage or throwing anything
- The true (first) queen cannot be removed. Attempts to remove the queen should have no effect (but should not cause an error).

# 8 Extensions

Feel free to implement any extensions that you may see fit. Some gameplay elements may be interesting, such as new units or new enemies. You can also change your game to a more classical tower defense where the units can move freely over a given path and/or provide ant updates that come at some price. But such work is more centered on the game design. While interesting, this is not the topic of this assignment. Please check that page for more pointers.

## 8.1 Functional Programming

But it is much more interesting to experiment with Functional Programming once your project is fully working in OOP. This topic is covered in a short set of blog posts discussing the programming of purely functional retrogames, focused on a simple Pac-Man. This blog is particularly interesting to programming learners. I am an absolute fan of its subtitle: "It's not about technology for its own sake. It's about being able to implement your ideas". Yeah! Exactly!

This other presentation presents the organization of a rather large game using the functional approach with Scala. This is probably larger than what you will need for your project, but that's still very interesting.

Remember, you should finish your project following the OOP approach before even thinking of the FP approach.

## 8.2 Complex Simulation

Another over-simplification lays in the handling of the time. Our game is currently turn-based while time is rarely discrete. But making a full-featured simulation engine has a lot of implications. You should read this paper on the game aspects. From a Computer Science perspective, you will end up writing a Discrete-Events Simulator (DES). If you want to make your game efficient even with huge amounts of entities (turning it into a real ant simulator), you may need to use advanced future events list structures and cleaver routing algorithm. You will also need to not animate the entities that are not currently on screen, but this will require to untangle the time of graphical animations from the time of logical updates. The former must occur at least 50 times per second for the entities on screen while the latter is well adapted to DES, with irregular updates. This brain teaser is mandatory to have a large but resource efficient simulation game...