

Hunger Game

1 Introduction

This project will give you practice with Swing.

2 Problem Description

Hunger Game is a video game where the player is lost in a forest and is extremely hungry. In order to not die of starvation, the player must hunt Animals. While Edible Animals are very nutritious, EndangeredSpecies are not very nutritious and are illegal to hunt. If the player kills too many EndangeredSpecies, the game will end with the player being arrested. As time goes on, the player's hunger increases. The player will die of starvation when the Hunger bar fills up.

3 Solution Description

Due to how involved this assignment is, you have been provided with a lot of skeleton code. This is to simulate how coding is often done in the real world, as you are rarely able to start a brand new project from scratch. Often, there is a substantial code base already written which you must interface with. Using the provided Javadocs to understand the code that has been given to you, finish writing the game as described above. A lot of the logic is provided, so you will mostly be dealing with Swing.

3.1 `Animal` `Class`

The abstract `Animal` class represents animals. All concrete `Animal` classes should extend this class. All `Animals` have a `Home` and most of the time they tend to hide there. Every once in a while `Animals` get brave and go out for a walk. Each `Animal` has a maximum distance that it is comfortable being from its home. This distance is a randomly generated int from 150 to 500 meters (inclusive).

3.2 `Boar`

The `Boar` class represents a wild boar. It is a concrete `Edible Animal`.

3.3 `Panda`

The `Panda` class represents a `Panda`. It is a concrete `EndangeredSpecies Animal`.

3.4 `EndangeredSpecies`

The `EndangeredSpecies` interface is implemented by all `EndangeredSpecies`.

3.5 `Edible`

The `Edible` interface is implemented by all other `Animals`. `Edibles` are much more nutritious than `EndangeredSpecies` and they don't get you arrested for eating them.

3.6 `Your` `Classes`

The only complete classes that you must write are:

1. At LEAST ONE of your very own concrete `Edible Animal`.
2. At LEAST ONE of your very own concrete `EndangeredSpecies Animal`

The game will be more effective if you use icons without backgrounds.

3.7 `Home`

The `Home` class is where `Animals` are born and live. `Homes` have a location.

3.8 `Direction`

The `Direction` enum represents two directions: out and in. An `Animal` has a `Direction` in which it is moving.

3.9 Nature

The `Nature` class is a singleton class. This means that there is only ever one instance of `Nature`. You have used singletons before (`NumberFormat` has a `getCurrencyInstance()` that returns its sole instance). `Nature` has a `getInstance()` method that returns its sole instance. `Nature` uses `Vectors` instead of `ArrayLists` because `Vectors` are thread safe. For the purpose of this project, just assume they are `ArrayLists`. The `Vectors` are as follows:

1. **existingAnimals**: all of the `Animals` in the forest.
2. **homes**: all of the homes in the forest.
3. **chosenOnes**: all of the `Animals` that have become brave enough to wander off into the forest.

Every time the player shoots, `Nature` is in charge of checking to see if the wandering `Animals` were within the scope of the shot. If they are, they are removed from all vectors and return the food points or wanted level of the animals. `Nature` uses factory methods to populate the forest.

1. **createAnimal(Class, Home)**: creates an `Animal` of a specific class and assigns it to a specific `Home`.
2. **createAnimal(Home)**: creates a random `Animal` and assigns it a specific `Home`.
3. **createEdible(Home)**: creates a random `Edible` and assigns it a specific `Home`.
4. **createEndangered(Home)**: creates a random `EndangeredSpecies` and assigns it a specific `Home`.

`Nature` has 3 Class arrays:

1. **animals**:
2. **edibles**
3. **endangered**

Don't worry too much about what these do, just make sure that you add your classes to the appropriate arrays.

The `motivate()` method is what motivates the chosen animals to move.

3.10 Drawable

The `Drawable` interface is implemented by all classes that can be drawn (have a graphical representation). Classes that must implement `Drawable`:

1. **Animal**
2. **Home**
3. **Nature**

You must make sure all of these classes actually implement `Drawable`.

3.11 GamePanel

The `GamePanel` class is in charge of all the game logic. The game runs on a `Timer`. Every tick of the timer, the `TimerListener` will execute the code in its `actionPerformed` method. This is what must be done every tick:

1. The hunger must be increased by -1.
2. The appropriate `Animals` should move.
3. The `Panel` must be painted using the `paintComponent(Graphics)` method.
4. You must check to see if the player has been arrested. If so, you must stop the timer and show a message dialog saying that you've been arrested.
5. You must check to see if the player has died. If so, you must stop the timer and show a message dialog saying that you have died of starvation.

The `GamePanel` also has a `MouseListener`. This has three methods: `mouseMoved(MouseEvent m)`, `mouseClicked(MouseEvent m)`, and `mouseExited(MouseEvent m)`. `mouseClicked(MouseEvent m)` and `mouseExited(MouseEvent m)` have been provided. You must fill in the `mouseMoved(MouseEvent m)` method.

Every time the mouse is moved you must ensure that:

1. The `Panel` is painted using the `paintComponent(Graphics)` method.
2. The `lastMouse` member variable is updated with the new mouse `Point`.
3. The `crossHairs` should be drawn on the `GamePanel`'s graphics. In order to draw the image at the right location you must use the `upperLeft()` static method in `GamePanel`.

You must complete the `shoot(Rectangle)`. It must perform the following operations:

1. Increase the hunger with whatever is returned by `Nature`'s `shootFood(Rectangle)` method.
2. Increase the wanted level with whatever is returned by `Nature`'s `shootEndangered(Rectangle)` method.
3. Increase the hunger with the same wanted points that was returned by the `shootEndangered(Rectangle)` method.

Make sure that `Nature`'s `shootFood(Rectangle)` and `shootEndangered(Rectangle)` methods are called only once in this method. You must also fill in a few lines of code in `GamePanel`'s constructor.

3.12 ControlPanel

`ControlPanel` should have a height of 75, and use a `BorderLayout`. `ControlPanel` has two `ProgressBars`: `hungerBar` and `wantedBar`. The `hungerBar` must be added to the left side of the panel and the `wantedBar` must be added to the right side of the panel. The `addButtonCenter(JButton)` is used to add a button to the middle of the `ControlPanel`. This method is called by the `GamePanel` to add the play/pause button to the middle of its `ControlPanel`. You must complete this method.

3.13 HungerGame

The `HungerGame` class is provided. It is the `Driver` class of the game.

4 Checkstyle

Review the [CS 1331 Code Conventions](#) and download the [Checkstyle jar](#) file and the [configuration](#) file to the directory that contains your Java source files. Run Checkstyle on your code like this (in the directory containing all your Java source files):

```
$ java -jar checkstyle-5.6-all.jar -c cs1331-checkstyle.xml *.java
Starting audit...
Audit done.
```

The message above means there were no Checkstyle errors. You can easily count the errors by piping the output of Checkstyle through `grep`.

```
$ java -jar checkstyle-5.6-all.jar -c cs1331-checkstyle.xml *.java | grep -cEv "(Starting
audit...|Audit done)"
0
$
```

The `-c` option tells `grep` to count matching lines instead of printing them, `E` means use `egrep` (extended `grep`) syntax, and `v` means invert the match. Here we use an inverted match to discard the two non-error lines of Checkstyle's output. If you have `egrep` you could leave off the `-E` and just use `egrep`.

```
$ java -jar checkstyle-5.6-all.jar -c cs1331-checkstyle.xml *.java | egrep -cv "(Starting
audit...|Audit done)"
0
$
```

Alternatively, if you are Windows, you can use this command to count the number of style errors by piping output through the `findstr` program.

```
C:\textbackslash > java -jar checkstyle-5.6-all.jar -c cs1331-checkstyle.xml *.java | findstr /v
"Starting audit..." | findstr /v "Audit done" | find /c /v "!!!!!"
```

The Java source files we provide contain no [Checkstyle](#) errors. You are responsible for any [Checkstyle](#) errors you introduce when modifying these files. Be warned — the Checkstyle cap from this homework onward is 100. Yes, this means you can receive 0s for perfectly functional code if it is messy enough. Don't make this mistake! Run Checkstyle early and often.

5 Tips

- Before you even start coding, read through all of the provided documentation and files. ALL OF IT.
- Draw out a diagram of how the provided files interact with each other on paper.
- Think about how the classes you'll be filling out need to interact with one another. Much of what you'll be writing will be calling other methods that are already written or methods which need to be filled out as well.
- The Java API will help you out significantly on this homework, particularly regarding the intricacies of various Layout managers and Swing components.

6 Turn-in Procedure

Submit all of the Java source files your project uses to T-Square in addition to any images it requires. Yes, this includes the images provided to you. Do not submit any compiled bytecode (`.class` files), the Checkstyle jar file, or the `cs1331-checkstyle.xml` file. When you're ready, double-check that you have submitted and not just saved a draft. These files should include (but are not limited to):

1. `Animal.java`
2. `Boar.java`
3. `ControlPanel.java`
4. `Direction.java`
5. `Drawable.java`
6. `Edible.java`
7. `EndangeredSpecies.java`
8. `GamePanel.java`
9. `Home.java`
10. `HungerGame.java`
11. `Nature.java`
12. `Pand.java`
13. `YourEdible.java` (any class name)
14. `YourEndangeredSpecies.java` (any class name)
15. all of the image files provided as well as any additional image files needed in your own classes.

7 Verify the Success of Your Submission to T-Square

Practice safe submission! Verify that your HW files were truly submitted correctly, the upload was successful, and that the files compile and run. It is solely your responsibility to turn in your homework and practice this safe submission safeguard.

1. After uploading the files to T-Square you should receive an email from T-Square listing the names of the files that were uploaded and received. If you do not get the confirmation email almost immediately, something is wrong with your HW submission and/or your email. Even receiving the email does not guarantee that you turned in exactly what you intended.
2. After submitting the files to T-Square, return to the Assignment menu option and this homework. It should show the submitted files.
3. Download copies of your submitted files from the T-Square Assignment page placing them in a new folder.
4. Recompile and test those exact files.
5. This helps guard against a few things.
 - (a) It helps insure that you turn in the correct files.
 - (b) It helps you realize if you omit a file or files.¹ (If you do discover that you omitted a file, submit all of your files again, not just the missing one.)
 - (c) Helps find last minute causes of files not compiling and/or running.

¹Missing files will not be given any credit, and non-compiling homework solutions will receive few to zero points. Also recall that late homework will not be accepted regardless of excuse. Treat the due date with respect. The real due date is midnight Wednesday. Do not wait until the last minute!