Ant Farm GridWorld Project

Robert Glen Martin
School for the Talented and Gifted, Dallas, TX

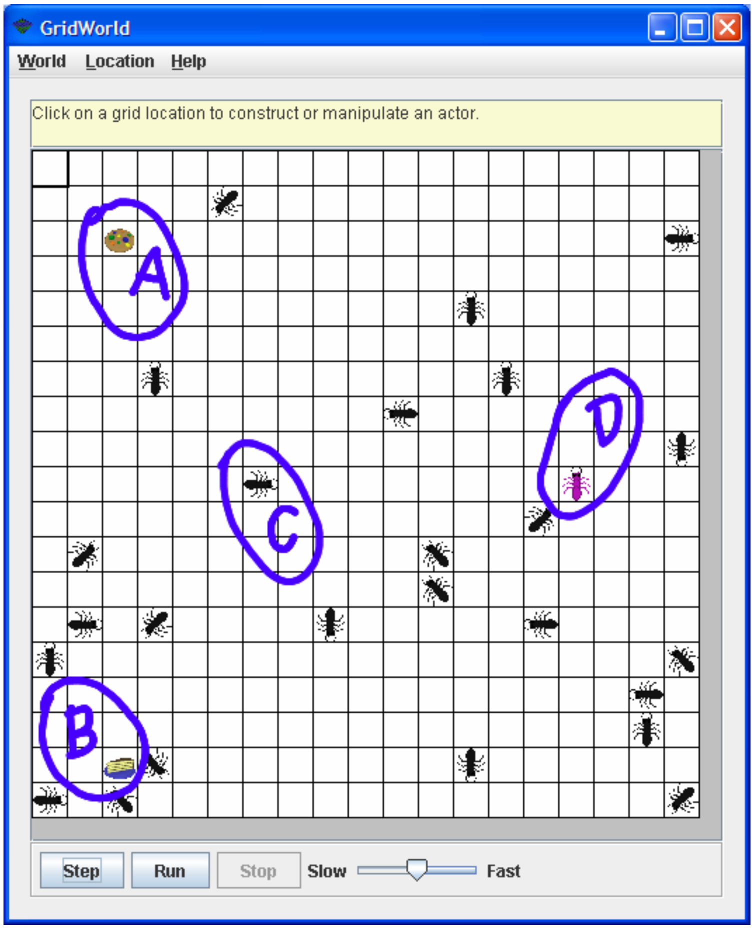
Introduction

This is the assignment for Ant Farm, a GridWorld programming project. You have a JCreator starter project contained in the folder named **AntFarm Student** and Javadoc documentation in the folder named **AntFarm Javadocs (index.html)**. As you work through this project, you will complete an interface and both concrete and abstract classes. Your solution will demonstrate inheritance, encapsulation, and polymorphism. Prior to beginning this project, you must read and understand the first four chapters of the GridWorld Student Manual.

Overview

The project utilizes four new types of objects (see **Figure 1**), two kinds of food (**A** - `Cookie` and **B** - `Cake`) and two kinds of ants (**C** - `WorkerAnt` and **D** - `QueenAnt`). Initially, the worker ants walk around in search of food. When they find food, they take a bite. Ants with food turn red. Then the worker ants go in search of a queen ant to give food. Once they give their food to a queen, they turn black and go back to get more food.

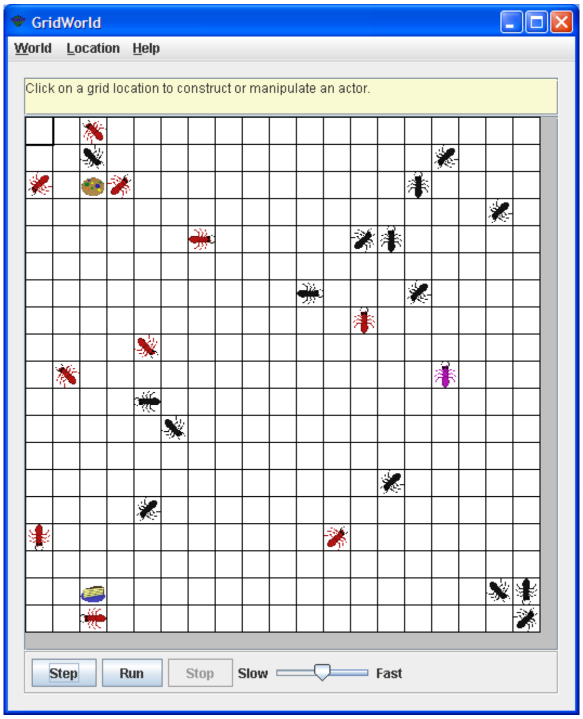
Food and queens remain stationary. Worker ants remember the locations of the food and queen. Additionally, they share those locations with other worker ants they meet.



Click on a grid location to construct or manipulate an actor.

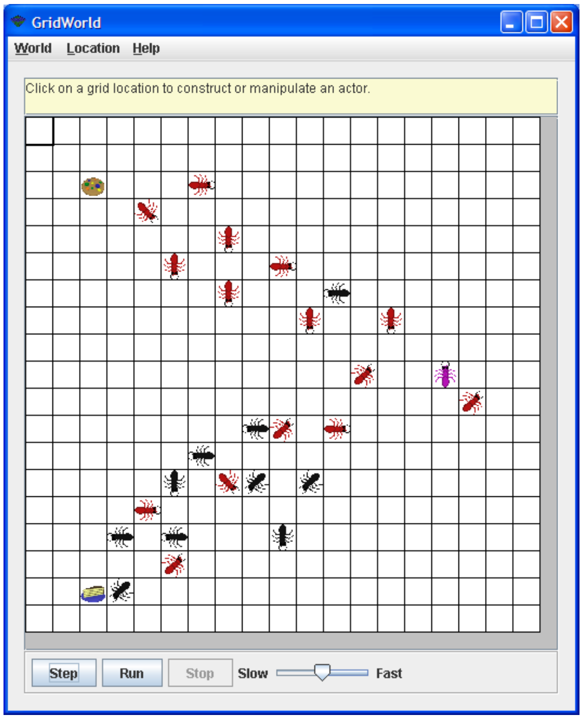
Figure 1 – Ant Farm (Initial State) – Worker ants hunt for food.

When the Ant Farm program starts, the worker ants are spread around the grid in random locations. Initially, they are disorganized as they search for food. As the worker ants start to find food and the queen, they get more organized (see **Figure 2** below). After all the ants learn the locations, they exhibit an emergent behavior that is very organized (see **Figure 3** below).



Click on a grid location to construct or manipulate an actor.

Figure 2 – Intermediate State – worker ants start learning locations of food & queen.



Click on a grid location to construct or manipulate an actor.

Figure 3 – Final State – worker ants know locations of food & queen.

Program Organization

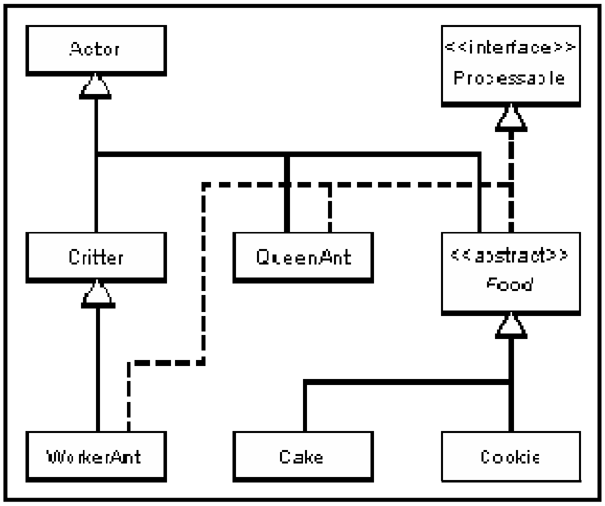


Figure 4 – Ant Farm Classes

Figure 4 shows the organization of the Ant Farm classes and interface.

GridWorld has a "built-in" `Actor` class for objects that "live" in the grid. Actors that have minimal interaction with other objects in the grid normally inherit from Actor. This is appropriate for `QueenAnt` and `Food.` `Cake` and `Cookie` inherit indirectly from `Actor`.

The other "built-in" actor is `Critter,` which inherits from `Actor`. `Critters` have additional methods that are useful for interacting with other actors. `WorkerAnts` need to "communicate" with the `QueenAnt`, `Cake`, `Cookie`, and other `WorkerAnt` objects. So inheriting from `Critter` is appropriate for them.

Ant Farm also has a new `Processable` interface. This interface has a single `process` method that is the key to communication between worker ants and the other actors.

Starter Code

Folder **AntFarm Student** contains a JCreator project with starter code for the Ant Farm interface and classes. `AntFarmRunner` (the application) is complete and requires no changes. The needed imports, class headings, method headings, block comments, and image (gif) files are provided for the remaining classes. The interface heading and comments are provided for `Processable`.

To Do Compiling early and often is a good programming practice. It helps identify errors when they are easiest to fix. Compile and execute the project. You should see all the actors on the screen. All the actors are blue at this point. Why?

Note: Clicking the **Step** or **Run** button at this point will cause a `NullPointerException.`

Processable Interface

To Do It is critical that you understand the `Processable` interface and how it is used in Ant Farm. Examine the `Processable.java` file. The `Processable` interface, contains a single `void process` method. This `process` method takes one parameter of type `WorkerAnt.` Add the method heading for `process`. **Note**: All interface methods are automatically public and abstract. Compile the project and correct any errors.

When implemented in `QueenAnt`, `Food`, and `WorkerAnt`, the `process` method processes (communicates with) a single `WorkerAnt` object (the one passed as a parameter). This interface allows worker ants to invoke the other actor's `process` methods polymorphically. The individual `process` methods in each class will do the following:

- `QueenAnt`
  - Get food from the worker ant.
  - Give the queen's location to the worker ant.

- `Food`
  - Give food to the worker ant.
  - Give the food's location to the worker ant.

- `WorkerAnt`
  - Give the saved food location to the other worker ant.
  - Give the save queen location to the other worker ant.

Note that Ant Farm uses the `Processable` interface to implement an interface variant of the Template Design Pattern. The Template Design Pattern normally uses an abstract class to contain the abstract method(s). Then concrete classes (which inherit from the abstract class) implement the method(s) as appropriate. In Ant Farm, we use the `Processable` interface to hold the abstract `process` method. `process` methods are written in the `QueenAnt`, `Food`, and `WorkerAnt` classes, each of which implement `Processable`.

QueenAnt Class

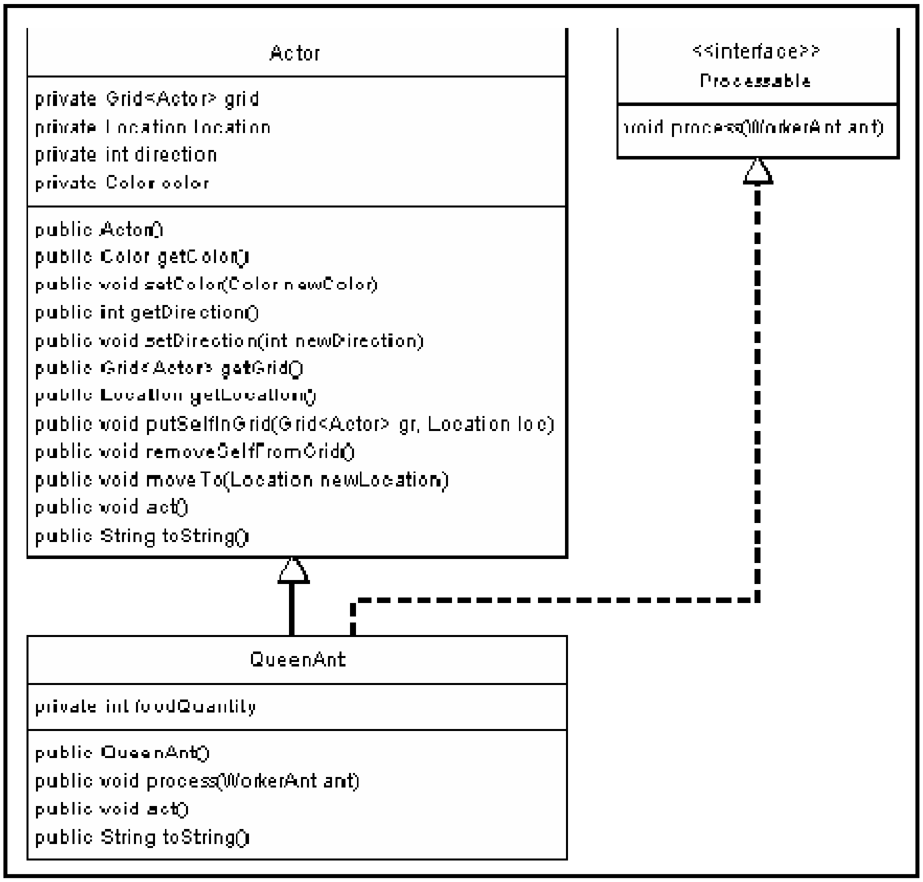

Actor
private Grid<Actor> grid
private Location location
private int direction
private Color color
public Actor()
public Color getColor()
public void setColor(Color newColor)
public int getDirection()
public void setDirection(int newDirection)
public Grid<Actor> getGrid()
public Location getLocation()
public void putSelfInGrid(Grid<Actor> gr, Location loc)
public void removeSelfFromGrid()
public void moveTo(Location newLocation)
public void act()
public String toString()
<<interface>>
Processable
void process(WorkerAnt ant)
QueenAnt
private int foodQuantity
public QueenAnt()
public void process(WorkerAnt ant)
public void act()
public String toString()

Figure 5 – QueenAnt Class

Figure 5 shows the `QueenAnt` class. Queen ants are the simplest of the new Ant Farm objects.

To Do Start updating `QueenAnt` by adding the new `foodQuantity` instance field. It is used to contain the total amount of food that has been given to the queen by the worker ants. You will make all instance fields `private` to preserve encapsulation.

To Do Write the constructor body. It needs to initialize `foodQuantity` to `0` and use the inherited `setColor` method to set the queen's color to `Color.MAGENTA`.

To Do Since `QueenAnt` implements `Processable`, you need to write the `process` method. `process` needs to get food from the passed worker ant using the `WorkerAnt giveFood` method. This method, which is shown in **Figure 7** below, returns an `int` amount which should be added to `foodQuantity`. `process` also needs to provide the worker ant with queen's location by calling the `WorkerAnt shareQueenLocation` method. Write the `process` method.

To Do The `Actor act` method needs to be overridden with an empty "do nothing" method (`QueenAnts` don't act). Look at `QueenAnt` to see how this was accomplished. Note the use of the `@Override` annotation (on the line preceding the `act` method heading). Although annotations are not included in the AP Java subset, `@Override` is very helpful. If you accidentally misspell the method name, `@Override` will cause a compile error telling you about this mistake. This error can be very difficult to find otherwise. The compiler is your friend.

To Do The Actor toString method also needs to be overridden to add additional information to the string returned by Actor's toString. This provides a good example of using super to call a super class method. Replace the toString body with the following:

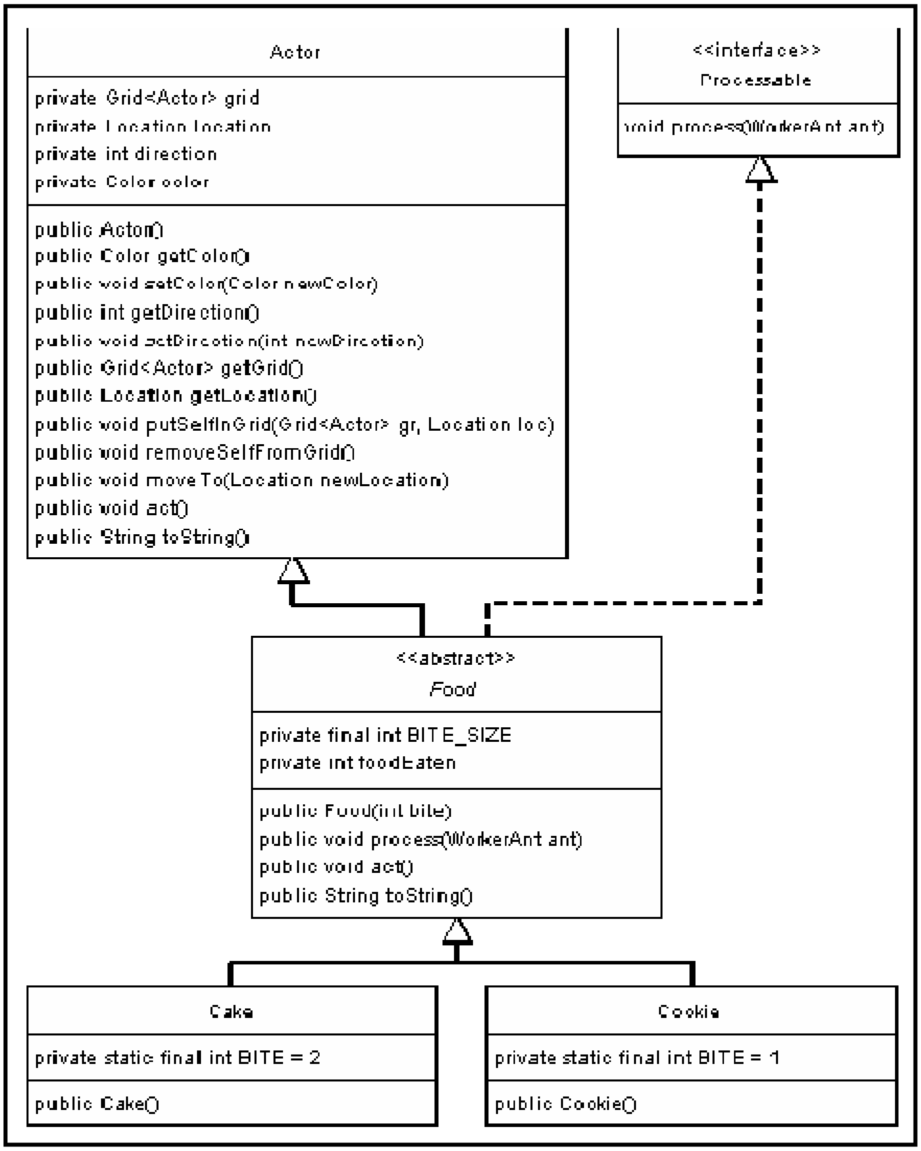
```
return super.toString() +
    ", FQty=" + foodQuantity;
```

To Do Compile and execute your project. The queen ant at location (15, 9) should now be magenta. Right-click the queen ant and execute its toString method. You should get "QueenAnt[location=(9, 15),direction=0,color=java.awt.Color[r=255,g=0,b=255]], FQty=0". Observe that this toString information is also shown when you place your mouse over the queen ant.

Foods - Food, Cake, and Cookie Classes

Figure 6 shows the Food, Cake, and Cookie classes. Foods act like queens, but they give food instead of getting it.

Different kinds of food are very similar. They differ only by the size of a bite and the displayed image. To take advantage of this similarity, the common instance fields and methods are placed in a Food super class. This class contains no abstract methods, but it is declared abstract so that it can not be instantiated. All foods have both bite sizes and keep track of the total amount that has been eaten. So, instead of repeating this information in both Cake and Cookie, it is stored in Food instance fields:



Actor
private Grid<Actor> grid
private Location location
private int direction
private Color color
public Actor()
public Color getColor()
public void setColor(Color newColor)
public int getDirection()
public void setDirection(int newDirection)
public Grid<Actor> getGrid()
public Location getLocation()
public void putSelfInGrid(Grid<Actor> gr, Location loc)
public void removeSelfFromGrid()
public void moveTo(Location newLocation)
public void act()
public String toString()
<<interface>>
Processable
void process(WorkerAnt ant)
<<abstract>>
Food
private final int BITE_SIZE
private int foodEaten
public Food(int bite)
public void process(WorkerAnt ant)
public void act()
public String toString()
Cake
private static final int BITE = 2
public Cake()
Cookie
private static final int BITE = 1
public Cookie()

Figure 6 – Food, Cake, and Cookie Classes

- `BITE_SIZE` – a constant that determines how much food is given to a worker ant when it gets food.
- `foodEaten` – keeps track of the total amount of food “given” to worker ants.

The constructor initializes `BITE_SIZE` to the `bite` value passed in the parameter, initializes `foodEaten` to 0, and calls `setColor(null)` so that the Cake.gif and Cookie.gif images display with their original coloring.

To Do Update `Food` to include the two new instance fields and complete the constructor as discussed above. You will need to uncomment the constructor heading and brackets. Note that Java allows constant (`final`) instance fields to be initialized in a constructor.

To Do Compile your Ant Farm project again. `Food` should have no errors, but `Cake` and `Cookie` now have compile errors. Why? What change caused these errors? We will fix these errors later.

All foods implement the `process` method (from `Processable`) to give food to the passed worker ant and to provide it the food’s location. Foods need to override the `Actor act` method with an empty “do nothing” method (foods don’t act). Foods also need to override the `toString` method to include the `BITE_SIZE` and `foodEaten` information. Since all three of these methods are the same for all foods, they are placed in `Food`. Otherwise they would have to be written in both `Cake` and `Cookie`.

To Do Write the `process` method (use `WorkerAnt`’s `takeFood` and `shareFoodLocation` methods). Also replace the body of the `toString` method as discussed above. Don’t forget to include the `Actor` super class `toString` information like you did with `QueenAnt`. Make sure that `Food` compiles without error.

Because of the `Food` class, the `Cake` and `Cookie` classes are very simple. They contain a single class constant `BITE` which contains the size of a bite. They each have a one statement constructor which passes the value of `BITE` to the `Food` constructor.

To Do Complete the Cake and Cookie classes by adding the `BITE` class constants (see **Figure 6** for the appropriate values).

To Do Complete the Cake and Cookie constructors by adding a single statement - `super(BITE);` This causes the one parameter `Food` constructor to be used when a cake or cookie is created.

To Do Compile and execute the project. The cake and cookie should now display properly. They should not be tinted. Hover your mouse above the cake and cookie images to make sure that the `toString` methods are working properly. For example, the cookie should display “(2, 2) contains Cookie[location=(2, 2),direction=0,color=null], BSize=1, FEaten=0”

WorkerAnt Class

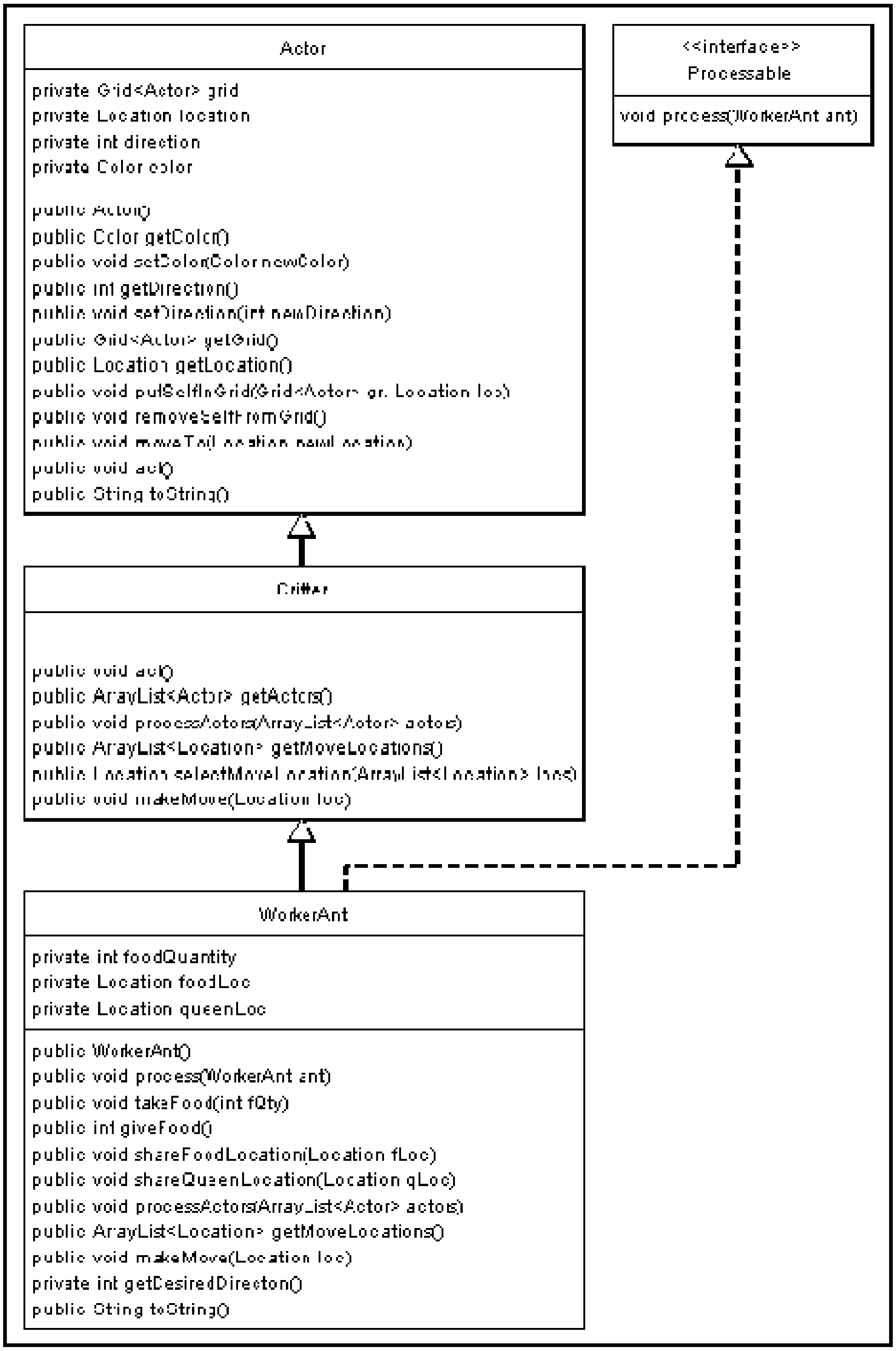

Actor
private Grid<Actor> grid
private Location location
private int direction
private Color color
public Actor()
public Color getColor()
public void setColor(Color newColor)
public int getDirection()
public void setDirection(int newDirection)
public Grid<Actor> getGrid()
public Location getLocation()
public void putSelfInGrid(Grid<Actor> gr, Location loc)
public void removeSelfFromGrid()
public void moveTo(Location newLocation)
public void act()
public String toString()
<<interface>>
Processable
void process(WorkerAnt ant)
Critter
public void act()
public ArrayList<Actor> getActors()
public void processActors(ArrayList<Actor> actors)
public ArrayList<Location> getMoveLocations()
public Location selectMoveLocation(ArrayList<Location> locs)
public void makeMove(Location loc)
WorkerAnt
private int foodQuantity
private Location foodLoc
private Location queenLoc
public WorkerAnt()
public void process(WorkerAnt ant)
public void takeFood(int fQty)
public int giveFood()
public void shareFoodLocation(Location fLoc)
public void shareQueenLocation(Location qLoc)
public void processActors(ArrayList<Actor> actors)
public ArrayList<Location> getMoveLocations()
public void makeMove(Location loc)
private int getDesiredDirecton()
public String toString()

Figure 7 – WorkerAnt Class

`WorkerAnt` (**Figure 7**) is the most complex Ant Farm class. This is to be expected, since `Critters` interact with other actors in the grid.

Worker ants have instance fields to keep track of the amount of food they currently have as well as known locations of the food and the queen.

The constructor initializes these instance fields (to `0`, `null`, and `null`), makes the ant black, and uses `Math.random` to randomly point the ant in one of the eight valid compass directions.

To Do Add the `WorkerAnt` instance fields and complete the constructor as discussed above and in **Figure 7**. Use the `Location` constants `HALF_RIGHT` and `FULL_CIRCLE` when writing your constructor. Do not "hard code" constants like 45 and 8. Compile and execute the project to check your code.

`WorkerAnt` implements the `process` method to share queen and food locations with other worker ants. `WorkerAnt` has four methods that do the actual “processing.” They are `takeFood`, `giveFood`, `shareFoodLocation`, and `shareQueenLocation`. These methods are called from the `process` methods of the `QueenAnt`, `Food`, and `WorkerAnt` classes.

To Do Complete the five processing methods as follows. Make sure that the project compiles after every change.

1. `process` – call the passed worker ant’s `shareFoodLocation` and `shareQueenLocation` methods to share the food and queen locations with the other ant.
2. `takeFood` – add the amount of food passed in `fQty` to the food quantity instance field.
3. `giveFood` – replace the method body to return the current food quantity to the caller (queen). Before `giveFood` exits, the food quantity needs to be reset to zero (all the food is being given to the queen).
4. `shareFoodLocation` – Foods and worker ants call `shareFoodLocation` to share the food location. If the current saved food location is `null`, then set it to the value of `qLoc`.
5. `shareQueenLocation` – Queens and worker ants call `shareQueenLocation` to share the queen location. If the current saved queen location is `null`, then set it to the value of `fLoc`.

You are about ready to override several of the `Critter act` methods, but first you will need to complete the useful `getDesiredDirecton` private helper method. This method returns the general direction that the ant wants to go.

To Do Replace the `getDesiredDirecton` body to return one of three directions:

1. If the queen location is not null and the food quantitiy is not zero, then return the direction from this ant toward the known location of the queen (use `Location`’s `getDirectionToward` method).
2. Otherwise, if the food location is not null and the food quantity is zero, then return the direction from this ant toward the known location of the food.
3. Otherwise, return the current direction of this ant.

The `Critter act` method calls the following methods in this order:

1. `getActors` – gets a list of actors for interaction.
2. `processActors` – interacts with each of the actors in the list from `getActors`.
3. `getMoveLocations` – gets a list of possible locations for moving this critter.
4. `selectMoveLocation` – chooses one of the possible move locations for this critter.
5. `makeMove` – moves this critter.

`WorkerAnt` inherits the `Critter act` method which does the following:

1. Uses the inherited `getActors` to get all the adjacent neighboring actors.

2. `processActors` processes each of the neighboring ant farm actors. This method should be very short. It needs a loop to traverse (loop through) the `actors ArrayList`. An **enhanced for loop** works well for this. Each `actor` in `actors` needs to call its `process` method. The parameter for each call will be `this`, the reference to the worker ant executing

the `processActors` method.

An `actor` could be a `QueenAnt`, a `Cake`, a `Cookie`, or a `WorkerAnt`. Without the `Processable` interface, `processActors` would need to determine the type of `actor` and then downcast the `actor` reference before making the call to `process`. But, since each of these classes implements `Processable`, `processActors` only needs to cast the `actor` to `Processable` before the call. This polymorphic processing is allowed because `Processable` contains the `process` abstract method. The Java Run Time Environment (JRE) determines the actual type of object at runtime and calls the appropriate `process` method.

To Do Complete the `processActors` method as discussed in the preceeding paragraphs.

3. `getMoveLocations` does the following:

   a. Calls the private `getDesiredDirecton` method to get the general direction the ant wants to move.

   b. Creates a list with up to three adjacent locations that are in the general direction of the one returned by `getDesiredDirection.` Locations are included if they meet all of the following criteria. They must be:
      i. Adjacent to the current location.
      ii. In the desired direction, or 45 degrees to the left of the desired direction, or 45 degrees to the right of the desired direction.
      iii. Valid (in the grid).
      iv. Empty.

   c. Returns the list of locations.

To Do Replace the `getMoveLocations` body as discussed above. For part b, use `Location`'s `HALF_LEFT` and `HALF_RIGHT` constants and the `getAdjacentLocation` method. You will want to use `Grid`'s `isValid` to see if a given location is valid (is in the grid) and `get` to help see if the location is empty (`get` returns `null`).

4. Uses the inherited `selectMoveLocation` to randomly select one of the possible locations. If the list of possible locations is empty, it returns the current location.

5. If the selected move location is different from the current location, `makeMove` moves to the selected location and changes its direction to match the direction it moved. Otherwise it stays put and changes its direction by randomly choosing between the two directions 45 degrees to the left or right (use `Location.HALF_LEFT` and `Location.HALF_RIGHT`). Then, in either case, it sets its color based on if it has food (red) or not (black).

To Do Write the body of the `makeMove` method as discussed above.

To Do Complete the `toString` body to include `Critter`’s `toString` result as well as the values of the `WorkerAnt` instance fields.

To Do Compile and thoroughly test your project. Make sure that your actors behave properly as described in the **Overview** section at the beginning of this assignment. You can learn a lot about the the state of your actors by viewing their `toString` information (hover over the object). Make sure this information changes appropriately as your actors interact with each other.