

**Pocket Adventure: Cultural Immersion**  
**A Greenfoot Java Game**  
Christine Yang

**Planning**

**Description of scenario:**

One problem that has been a challenge to perfect in elementary school education is the method of teaching cultural awareness- that is, the ability to empathize and understand complexly the differences between one's own culture and another's foreign culture. This particular issue is challenging because children typically do not have the opportunity to immerse themselves in foreign cultures at a young age. In Forsyth County, students do not begin formal foreign language training until the eighth grade, when many have already developed barriers of perceptions that would predispose one to a close minded attitude in the classroom. I have interviewed my high school German teacher, who thinks positively of the concept of a computer game which would allow children to learn about different cultures, thus normalizing the idea of "differences" between cultures, and facilitating an appreciation for such differences. His ideal solution would have dialogue features that allow the player to interact with ingame characters to learn facts about German culture relevant to daily life. In addition, there should be some sort of reward system that has cultural relevance as well, as an incentive for the player to interact with all characters.

**Rationale for the proposed project:**

I propose designing this solution within Greenfoot's object oriented programming environment, as a game which will center around a sprite the player will manipulate. Greenfoot's graphical interface is ideal for an interactive game because it is easy display text and see a Sprite's onscreen movement with keyboard triggers. In addition, Greenfoot's "checking for intersecting objects" feature will allow me to use event-based triggers to cause text or images to pop up. With the characters I plan on implementing, most will generally share a similar behavior except for the differences in dialogue and appearance. Because of this, it would be ideal to use the features of Object Oriented Programming to create a Human class that encompasses all similarly behaving humans. I could then define objects within the class, which would allow each object I create of that class to have its own image, name, and file. Furthermore, Greenfoot possesses a file reading capability that can be used through importing packages of java classes such as `BufferedReader` and `InputStream`, which will assist in storing (at the initialization of the program) and displaying text as needed. It would be very simple for even an inexperienced programmer to access and edit text files needed for the program to run, in case any additions needed to be made in the wake of a culture's change.


## Solution Overview


### Main Functions of Each Class:


- **World Class:**

- Germany
  - Contains all methods that add objects to world
  - Orders the alphabetized Food Array that makes up the Market's inventory
  - Displays You Won! message

- **Actor Classes:**

- Human 
  - Has a name, image, and file name
  - Reads in text files upon Initialization, saves each line as a String in an Array
  - Returns the next String (dialogue line) from array
  - Keeps track of how far in conversation player is

- Sprite 
  - Manipulated by user with keyboard controls, moves left, right, up, down
  - Check for Intersection with Human Objects or Market
  - Gets lines from Human object's lineArray and displays them
  - Adds interaction points to counter

- Food 
  - Has a name, image, and file name.
  - Displayed as a small icon in shop
  - When hovered over with mouse, will display larger image and description
  - Reads in description from text file sharing its name using BufferedReader
  - Controls counters to "buy" food

- Counter
  - Holds Interaction Points score

- Holds Foods Eaten score
- Market
  - Displayed when Sprite is in range of its hotspot
- Instruction
  - Displayed upon Initialization of world
  - Disappears when spacebar is pressed
- TextActor
  - Takes form of GreenfootImage to draw Strings and images temporarily

## Drafting Human line reading with Pseudocode

### In Human class:

```

    String name;
    Human (String name)
        this.name = name;
        filename = name + "txt";
        readText() into array for lines

    String getName()
        return name;
  
```

### In Germany class:

```

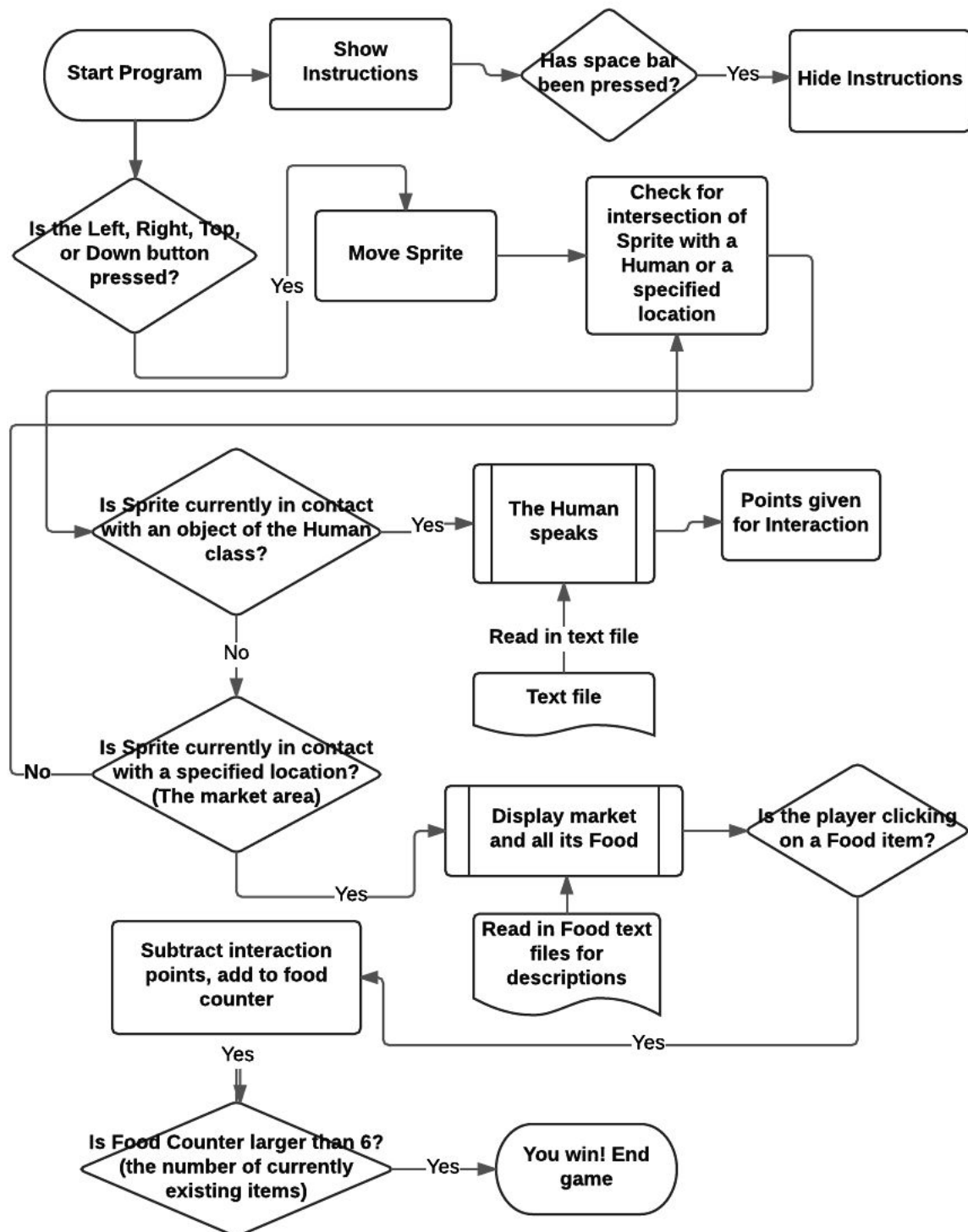
    Human j = new Human ("Jannik"); //creating object
  
```

### In Sprite class:

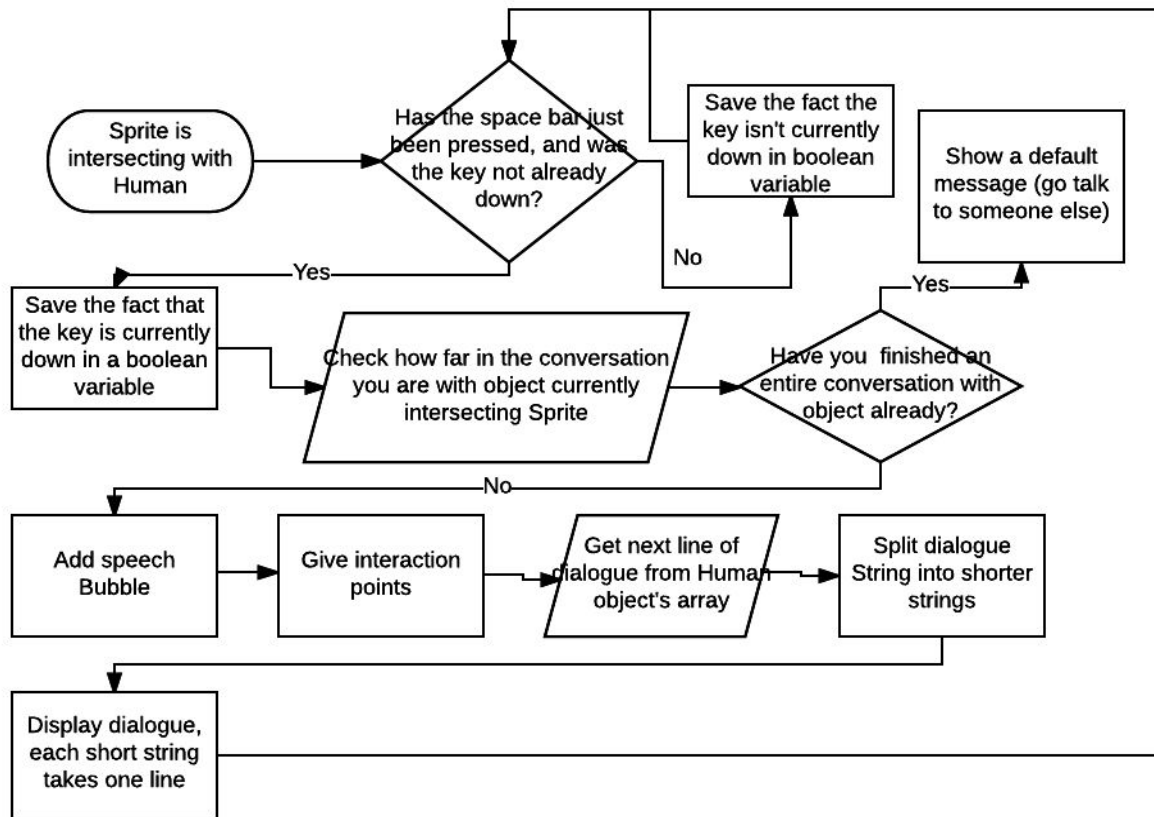
```

    Human h = getOneIntersectingObject
    If (h.getName().equals("...")) //check which Human intersecting with
    {
        //Human specific actions (triggers, extra images)
    }
    h.speak(); // call speak method, defined in Human class
  
```

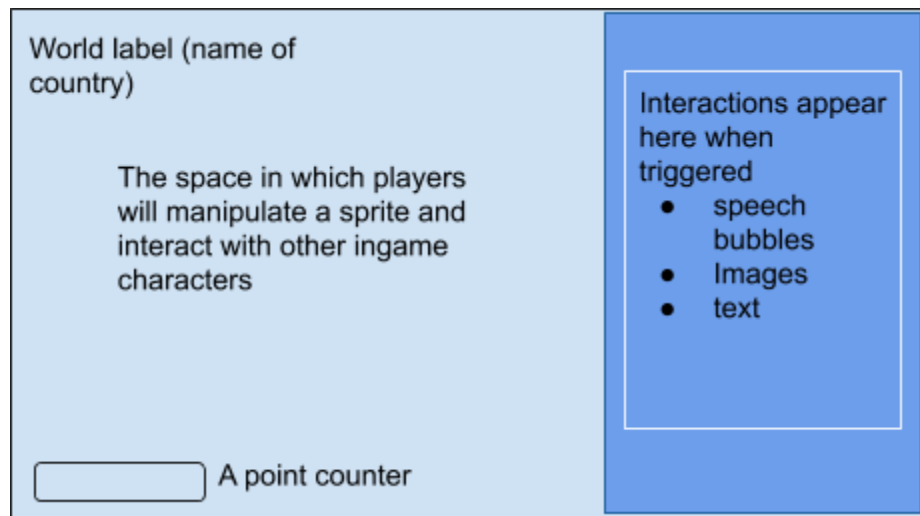
## General Program flow:



## Flowchart for “the Human Speaks” subprocess



## Graphic Interface Design



- Interactions occur when the player's sprite is touching an object of another class.

Diagram of what an interaction with an object of the Human Class will produce.

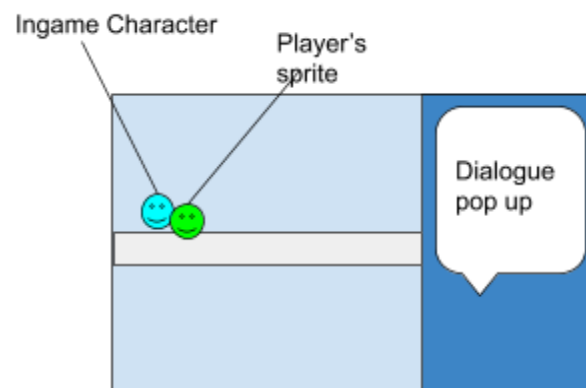
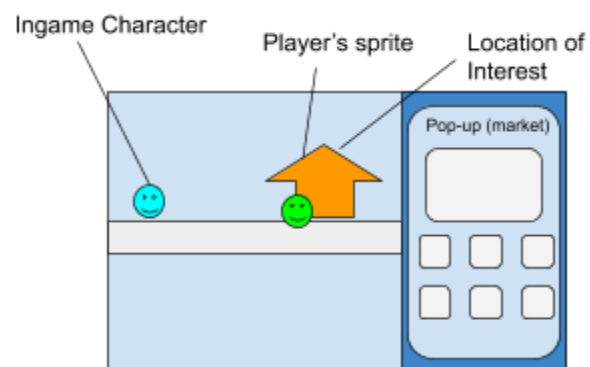


Diagram of what a pop up upon interacting with a location will produce. The Market is used as an example.



**Success Criteria:**

- Players will be able to read Instructions before playing
- All humans will function as separate objects of the same class.
- The player will be able to manipulate a Sprite to interact with ingame Characters.
- My program will read each Human's dialogue from text files.
- Players will be able to use the keyboard to cycle through an array of lines a character has, with a "press spacebar to continue" type interaction.
- Players will be rewarded for engaging with these other characters with points that they can then spend on food at a food market.
- The food market will display an ordered inventory of items.
- Players can buy food by clicking on them in the store inventory.
- Players win by buying all available food items.

## Development

This program was created with the Java coding language in Greenfoot.  
(See appendix for all code).

### Techniques Used:

- Creating objects of a class (ex- Human, Food)
- Local and global variables
- Checking for intersecting objects
- Reading text files
- Using line-split to display a long String
- Using Lists to alphabetize an array
- Getting references to object of other class

### Imported Packages

- greenfoot.\*
- java.awt.Color
- java.io.\*
- java.util.\*

### Creating Objects of a Class

In the method addHumans, which is called in the constructor of the Germany World class, objects j and o are created of the Human class. Humans as a Class act similarly (read files, talk, show image), but as individual objects they show distinct conversations and pictures.

```
public void addHumans()
{
    j = new Human ("jannik");
    o = new Human ("jonas");
    addObject (j , 150, 210);
    addObject (o, 300, 200);
}
```



As shown in the constructor of the Human class, Humans are created with the property of having a name, which is used to set the name of its image and text file. Originally Humans were created with properties of having a separately defined image name and file name, but I found organizationally, that could be simplified by adding file extensions to the defined name.

```
public Human (String humanName)
{
    this.name = humanName;
    himage = name + ".png";
    setImage (himage);
    file = name + ".txt";
    try {
        lineArray = readText();
    }
    catch (IOException ioe) {
        GreenfootImage lineImage = new GreenfootImage("Error: Can't Read Text", 16, Color.WHITE, Color.BLACK);
        Actor text = new TextActor();
        text.setImage(lineImage);
        getWorld().addObject(text, 720, 40);
    }
}
```

### Using the InputStream and BufferedReader classes

Each object of the Human class, upon creation, will create and save its own lineArray comprised of long Strings read in from its own text file. A loop will read in a new line from a text file until the next line returns “null.” At first, lines were read in from a file at the time the text was to be shown, but I then changed to reading files upon initialization and saving to an array to make the program more efficient. Then, the index of the line array could be incremented to display each new line. I decided to read dialogue lines from text files rather than write the entire Strings within the code to make future debugging and corrections easier. It is much easier to rewrite a text file than to scroll through code to change lines.

```
public String[] readText() throws IOException
{
    InputStream input = getClass().getClassLoader().getResourceAsStream(file);
    BufferedReader reader = new BufferedReader(new InputStreamReader(input));

    String[] lineArray = new String[10];

    int i = 0;
    String l = null;
    while ((l = reader.readLine()) != null){
        lineArray[i] = l;
        i = i + 1;
    }

    return lineArray;
}
```

## Global and Local Variables

checkKeyPress is called in the Sprite class's act method to check for keyboard input that determines movement. Local variable speed determines how fast the Sprite moves, and it is only used in this method.

```
public void checkKeyPress()
{
    int speed = 3;
    if (Greenfoot.isKeyDown("left") && getX() > 20 )
    {
        move(-speed);
    }
    if (Greenfoot.isKeyDown("right") && getX() < 560)
    {
        move(speed);
    }
    if (Greenfoot.isKeyDown("down") && getY() < 220)
    {
        setLocation(getX(), getY()+speed);
    }
    if (Greenfoot.isKeyDown("up") && getY() > 200)
    {
        setLocation(getX(), getY()-speed);
    }
}
```

Global variable convNumber tells how many lines a Human has spoken already and is called in multiple methods (getConvNumber, speak, defaultMessage) in the Human class. It will become the index of lineArray in the speak method, but if it is equal to 0 (have not yet conversed), the default “hello” message will show.

```
public int getConvNumber()
{
    return convNumber;
}

public String speak()
{
    convNumber = convNumber + 1;
    if (convNumber >= 10) {
        return lineArray[0];
        // after you've cycled through a character's conversation, will urge you to converse with others.
    }
    else {
        return lineArray[convNumber];
    }
}

public void defaultMessage()
{
    if (convNumber == 0 ){
        Actor speech = new TextActor();
        speech.setImage("nontspeech.png");
        getWorld().addObject(speech, 715, 150);
        GreenfootImage lineImage = new GreenfootImage( "Hey! I'm " + name + ". Press 'space' \n to talk to me.", 16, Color.BLACK, new Color (0,0,0,0));
        Actor text = new TextActor();
        text.setImage(lineImage);
        getWorld().addObject(text, 710, 60);
    }
}
```

## Checking for an Intersecting Object

```
public void checkForPerson()
{
    isHuman = (Human) getOneIntersectingObject(Human.class);
    if (isHuman != null) { //Sprite is intersecting with a human
        if (isIntersect == false) { //and you were not already intersecting with them
            isHuman.defaultMessage(); //show the Human's default message
            hSpeaking = true; //there is a human speaking
        }
        isIntersect = true; //sprite is intersecting with a human
    }

    else if (getWorld().getObjects(TextActor.class) != null) {
        //if at least one textactor currently exists and you're not intersecting with anything
        getWorld().removeObjects(getWorld().getObjects(TextActor.class)); //remove all text
        isIntersect = false; // sprite is not intersecting with a human
        hSpeaking = false; //no human is currently speaking
    }
}
```

Method `checkForPerson` is called in the `Sprite` class's `act` method to check whether the `Sprite` is currently intersecting with a `Human` object and returns `isHuman`, a `Human` object. The method to show this particular object's default message is then called.

## Reading and saving text in Human Class

```
public void humanSpeak()
{
    if (hSpeaking == true){ //there is a human currently speaking
        if ( isDown == false && (Greenfoot.isKeyDown("space") == true )) {
            //space key wasn't already down and the space key is being pressed (not held)
            isDown = true; //note the key is now being held down
            int cn = isHuman.getConvNumber(); //get how far in conversation you are

            if (cn<10){ //if you haven't already finished the conversation
                givePoints();
            }
            Actor speech = new TextActor();
            speech.setImage("nontspeech.png");
            getWorld().addObject(speech, 715, 150); //add speech bubble image
            showSpeak(); //display text
        }
        if ( isDown == true && (Greenfoot.isKeyDown("space") == false )) {
            //space key was being held down but now it is not
            isDown = false; //note the key is not being held down
        }
    }
}
```

If in `checkForPerson` boolean `hSpeaking` is set to true, `humanSpeak` will give points and call `showSpeak`. The if statement checks for Boolean value `isDown` in addition to if the space key is being pressed because showing a Message and giving points should only activate once every key press. This was implemented after issues originally when the space bar was held, as `givePoints` and `showSpeak` were accidentally called multiple times in succession. Similarly functioning boolean variables can also be seen throughout the program, for example boolean `isShowing` in the Germany class's `showFood` method.

Method showSpeak will display text by calling the intersecting Human object's speak method to retrieve a long String (containing several sentences). Originally, displaying multiple lines of text at once was a challenge because GreenfootImage only provides for drawing a single line of text. This was solved by splitting the longer String into several smaller strings where there was a hyphen in the text file and storing each smaller string into an array. The for loop in the code below will display each small string on its own line, and it will run while local variable i is less than the number of Strings in array words.

```
public void showSpeak()
{
    String line = isHuman.speak();
    String[] words = line.split("-"); //split the String where there's a -, save each shorter string in array words
    int height = 50;
    int count = words.length; //count length of words array

    for (int i = 0; i < count; i++) {
        GreenfootImage lineImage = new GreenfootImage( words[i] , 16, Color.BLACK, new Color (0,0,0,0));
        Actor text = new TextActor();
        text.setImage(lineImage);
        getWorld().addObject(text, 710, height); //place each shorter string on a line
        height = height + 20; //go to new line
    }
}
```

The speak method in the Human class increments convNumber by one each time it is called to track how many lines Human has spoken in a conversation. It will return the next long String saved in the array. A default line will be returned if you've already seen all of a Human's unique lines.

```
public String speak()
{
    convNumber = convNumber + 1;
    if (convNumber >=10) {
        return lineArray[0];
        // after you've cycled through a character's conversation, will urge you to converse with others.
    }
    else {
        return lineArray[convNumber];
    }
}
```

## Initializing the Alphabetized Food Array:

This feature would become useful if the market inventory were to contain many more items in the future so that adding new items to the list would not be very difficult.

First the foodNames array is initialized with Strings of names, in no particular order: (new items could be appended to the end of the array)

```
String[] foodNames =  
    {"haribo", "apfelschorle", "spaetzle", "kinder", "schnitzel", "currywurst"};  
String[] alphaFood;
```

The initializeArray method is called in the Constructor, creating a new ArrayList of Strings, named Food.

```
public void initializeArray()  
{  
    List<String> food = new ArrayList();  
    int count = foodNames.length; //number of food names entered  
    for (int i = 0; i < count; i++) {  
        food.add(foodNames[i]); //load Strings in name array into list  
    }  
    java.util.Collections.sort(food); // alphabetically sort list  
    alphaFood = food.toArray(new String[food.size()]);  
    //convert alphabetized list back to array for ease of use  
}
```

A new ArrayList is created, and the food names from the foodNames array are added to the List food. The array is converted to a List because in Greenfoot, Lists have the ability to sort themselves. After sorting, the list items are loaded back into an array.

Then, showFood is called in the act method and displays all Food items if the Market is being displayed. The if loop nested within the while loop assists in formatting rows in the market, where after placing every three items it will automatically start placing on a new row.

```
public void showFood()
{
    boolean isMarket = girl.returnIsMarket(); //get info if obj of market class exists in world
    int i = 0;
    if (isShowing == false && isMarket == true){
        //is food isn't already being displayed and the market is showing now
        int count = foodNames.length; //get length of foodNames array
        int xcoord = 630;
        int ycoord = 280;
        while (i < count) //for all objects in foodNames array
        {
            addObject(new Food(alphaFood[i]), xcoord + 85*i, ycoord);
            //add each Food Object to world in alphabetical order
            i++;
            if (i % 3 == 0) { //every three items, format automatically to a new row
                ycoord = ycoord + 80;
                xcoord = xcoord - (85*3);
            }
        }
        isShowing = true; //note the food is now showing
    }
    else if (isShowing == true && isMarket == false) //if the food is currently showing but the market isn't
    {
        removeObject(getObjects(Food.class)); //remove all the food!
        isShowing = false; //note no food is showing now
    }
}
```



## References to Object of Other Class

Code for the Counter Class was imported from Neil Brown and Michael Kölling's Counter design.

```
Counter interactionCount; //instance variables defined in World class
Counter foodEaten;
```

```
public void addCounters()
{
    interactionCount = new Counter();
    addObject (interactionCount, 50, 370);

    foodEaten = new Counter();
    addObject (foodEaten, 50, 340);
}
```

Two counters are defined as instance variables and then added to the world in the Germany class.

```
public Counter getIntCounter()
{
    return interactionCount;
}
```

```
public Counter getFoodCounter()
{
    return foodEaten;
}
```

Methods getIntCounter and getFoodCounter in the Germany class retrieve counter values so they can be accessed by the Food class.

```
private void buyAnItem()
{
    Germany germany = (Germany) getWorld(); //get reference to the world
    Counter ic = germany.getIntCounter(); //get reference to interaction counter
    Counter fc = germany.getFoodCounter(); //get reference to food counter
    int wallet = ic.getValue(); //return number of interaction points
    if (wallet >= 20){ //if you can afford food, buy a food item
        ic.add(-20); //subtract 20 points from interaction counter
        fc.add(1); //add 1 point to food counter
    }
}
```

Method buyAnItem in the Food class is called when a Food object is clicked. References to the world and the counters are created to control point distribution. These were necessary because the triggers to increment counters (buying food) occurred in the actions of other classes, not the Counter.



## Criterion E: Evaluation

### Recommendations for Future Improvements:

Because this game is directed at culture learning, there could be an addition of sound-playing features. As lines of dialogue are being displayed, pre-recorded sounds of the line being read could be played. Because text files are being read into the game, the content of them could be changed to focus on vocabulary learning for foreign languages, a suggestion my teacher and client made. This would enhance the value of adding sounds to the game. Such an addition would not be difficult with Greenfoot's built in support for accessing and playing sound.

In addition, more "hotspots" on screen could be added to display factual information about locations, which trigger similarly to the Market, when the Sprite intersects with their defined location. Because the only action required in adding new Humans is the creation of a text file for their lines and a new image, new Humans that have various occupations and perspectives could very easily be implemented.

Part of the logic in designing a food List that alphabetized itself was the idea that upon expansion of the Market (perhaps to a much greater inventory), adding items would be as simple as creating a text file with the item's name, creating an image with the item's name, and then adding the item's name to the end of the unordered foodNames array. If the shop inventory were to grow to one hundred items, one would not have to read through all the entries in the array to determine the appropriate place to insert a new name. Because the list of foods I have currently added hardly represent the entirety of German culture, I anticipate that more foods could and should be added in the future.

Even though this game was designed to reflect German culture, using the template of the code, the entire game could be tweaked to show another culture, for example, Spanish or French. The file reading and the ability to easily change the items in the food Array would make this shift very efficient. I can foresee that future versions of this game could be a cultural educational tool or language learning tool of any culture or language.