Creating a World, Animating Actors, and Ending a Game

What Will I Learn?

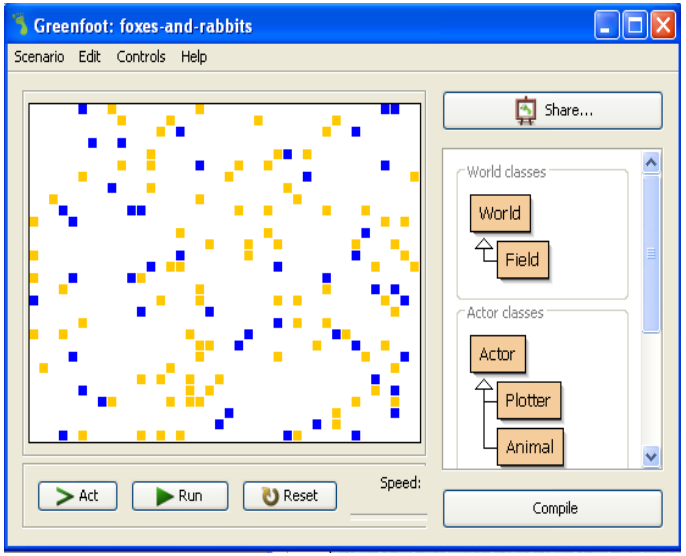
Objectives

- Construct a world object using a constructor method
- Create an object using a constructor
- Write programming statements to use the new keyword
- Define the purpose and syntax of a variable
- Recognize the syntax to define and test variables
- Write programming statements to switch between two images
- Write programming statements to end a game

Why Learn It?

Purpose

Most people enjoy playing games, but they like to know that there is an ending, or a final goal to achieve. Once the world is created, and actors are animated, focus can be placed on programming statements that will allow a game to end.



Constructors

When a new World subclass is created and compiled, Greenfoot executes a constructor that creates an instance of it and displays it in the scenario.

Constructors:

- Set up the instance and establish an initial state (size, resolution).
- Have no return type.
- Have the same name as the name of the class in which they are defined.
- Use syntax to have the constructor name immediately follow the word “public”.

Constructors are special methods that are executed automatically whenever a new instance of the class is created.

Constructor Parameters

The parameters of a constructor:

- Allow initial values for an instance to be passed into the constructor.
- Are only available to the instance created by the constructor.
- Have a restricted scope that is limited to where a construcctor is declared.
- Have a restricted lifetime that is limited to the single constructor call.
- Immediately disappear once a constructor is finished executing.
- Are valid field/variables as long as the instance exists exists.

Constructor Example

In this constructor, the world's height, width and resolution are defined in the World superclass of DukeWorld. To pass these values to the World superclass, use `super( )`.

```
import greenfoot.*;  // (World, Actor, GreenfootImage, Greenfoot and MouseInfo)

/**
 * Write a description of class DukeWorld here.
 *
 * @author Oracle Academy
 * @version JF_S03_L04
 */
public class DukeWorld extends World
{
    /**
     * This constructor creates a new world and the objects that start the
     * game.
     */
    public DukeWorld()
    {
        super(600, 400, 1);
    }
}
```

Parameters Example

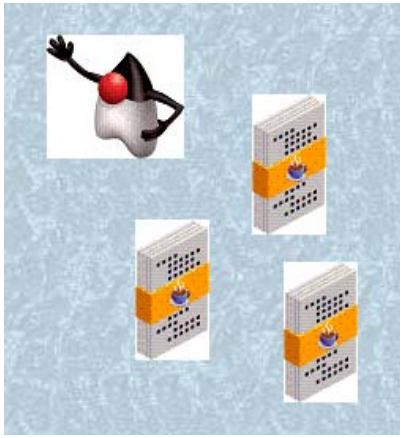
To create a different-sized game board, change the size and resolution in the constructor. This example makes the world square instead of rectangular by changing the x coordinate limit to 400 instead of 600.

```
/**
 * Write a description of class DukeWorld here.
 *
 * @author Oracle Academy
 * @version JF_S03_L04
 */
public class DukeWorld extends World
{
    /**
     * This constructor creates a new world and the objects that start the
     * game.
     */
    public DukeWorld()
    {
        super(400, 400, 1);
    }
}
```

Automatically Create Actor Instances

You can write code in the World constructor that will add the Actor instances to the game when the scenario is initialized. This eliminiates the need for the player to have to manually add instances before the game starts.

For example, in a matching game, the cards should automatically display in the scenario when the game starts.



Code to Automatically Create Instances

The code in the World constructor includes the following components:

- `super()` statement
- Size of the World as arguments in `super()`
- `addObject` method
- `new` keyword for the actor being added
- Location of the object in the `addObject` method

```
public DukeWorld()
{
     super(560, 560, 1);
     addObject (new Duke(), 150, 100);
}
```

Greenfoot Actor Instances

One way to give instances the appearance of movement is by manipulating their images. Alternating between two images that look slightly different give an instance the appearance of movement.

Greenfoot Actor instances:

- Receive and hold an image from their class (the image was assigned to the class when the class was created).
- Have the ability to hold multiple images.
- Can be programmed to change the image they display at any time.

GreenfootImage Class

The `GreenfootImage` class:

- Enables Greenfoot actors to maintain their visible image by holding an object of type `GreenfootImage`.
- Is used to help a class obtain and manipulate different types of images.
- Requires images used with this class to pre-exist in the scenario's Images folder.

Constructor to Obtain New Image Object

Create a constructor that gets a new image object from a file when creating an instance of a class. This constructor would include the:

- `SetImage` method
- `new` keyword
- `GreenfootImage` class
- Image file name as arguments in parameter list

The constructor below creates the new image, and attaches it to the Actor class.

```
setImage (new GreenfootImage(“duke5.png”));
```

Assigning a New Image to a Class

The statement below creates the new image object from the named image file. When inserted into the class's source code, this image object is ready for the class to use.

The statement is executed as follows:

- The `GreenfootImage` object is created first.
- The `setImage` method call is executed, passing the newly-created image object as an argument to the parameter list.

```
setImage(new GreenfootImage (“duke5.png”));
```

Assigning an Image Example

The `setImage` method assigns the image in the file "duke5.png" to the Actor class. Each time an instance of this class is added to the scenario, it displays the "duke5.png" image.

```
setImage (new GreenfootImage("duke5.png"));
```

Allow image object to be used by Actor class; Expects image as parameter

Create new image

Image from GreenfootImage class

Image file name as argument

Reasons Why Instances Hold Multiple Images

You may want an instance to hold and access multiple images:

- To appear to change color.
- To appear to change from one type of object to another (such as magically change from a rabbit to a tortoise).
- To give the appearance of movement:
  - Change from an object with left leg extended, to one with right leg extended, to give the appearance of walking.
  - Give the appearance that a card is flipped over in a matching game or other card game.

Accessing Multiple Images

We may want an instance to access two images, each with a slightly different arm position, so the instance waves its arm as it moves.

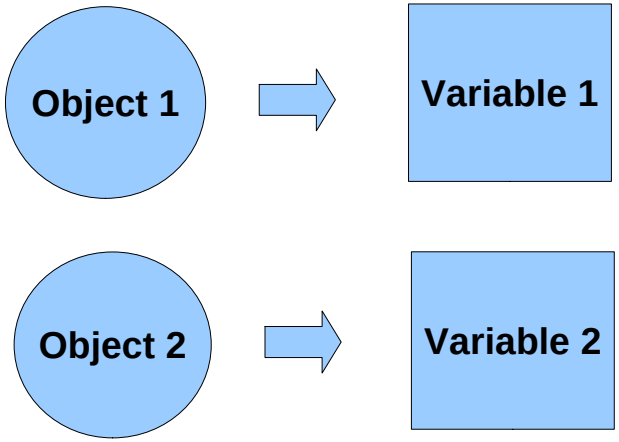
To achieve this:

- Create two images of the instance, each with slightly different arm positions.
- Store the two images in the instance, so they can be accessed repeatedly as the object moves.
- Code the class to alternate between the two images that are displayed.

Variables

Use a variable to store the two image objects in the class, so the class can easily access them for use with the instances.

A variable is declared in a class. It is used to store information for later use, or to pass information. It can store objects or values.



Variable Format

Variable format includes:

- Data type: What type of data to store in the variable.
- Variable name: Describes what the variable is used for so it can be referred to later.

```
private variable-type  variable-name;
```

Example

The variable name is image1 and the variable type is GreenfootImage.

```
private GreenfootImage image1
```

Declaring Variables

Declare variables before the constructors and methods.
The format for declaring a variable includes the:

- Keyword `private` to indicate that the variable is only available within the Actor class.
- Class to which the image belongs.
- Placeholder for the variable into which the image will be stored.

```
/**
 * Write a description of class Duke here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class Duke extends Animal
{
private GreenfootImage image1;
private GreenfootImage image2;
```

Variables

Assignment Statements

An assignment is needed to store objects in a variable. When an object is assigned to a variable, the variable contains a reference to that object.

An assignment statement:

- Stores the object or value into a variable
- Is written with an equals symbol

Format

```
Variable = expression;
```

Assignment Statement Components

An assignment statement includes:

- Variable: Name of variable to store object or value
- Equals symbol (the assignment symbol)
- Expression:
  - Name of object or value to assign
  - An instruction that the object or value is new
  - The class to which the image belongs

Example

```
image1 = new GreenfootImage(“duke.png”);
image2 = new GreenfootImage(“duke2.png”);
```

Initializing Images or Values

"Initializing" is the process of establishing the instance and establishing its initial values.

When the class creates new instances, each instance contains a reference to the images or values contained in the variables. They follow these guidelines:

- Signature does not include a return type
- Name of constructor is the same as name of the class
- Constructor is automatically executed (to pass the image or value on to the instance) when an instance of the class is created

Actor Constructors Example

The following actor constructor tells Greenfoot to automatically create a new Duke instance and initialize, or assign, two variables to the instance.

```
public class Duke extends Animal
{
    private GreenfootImage image1;
    private GreenfootImage image2;
    /**
     * Create a Duke and initialize his two images.
     */
    public Duke()
    {
        image1 = new GreenfootImage("duke.png");
        image2 = new GreenfootImage("duke2.png");
        setImage(image1);
    }
```

The last line of the constructor, `setImage(image1)`, indicates that the first variable should display when the instance is added to the scenario.

Test Values of Variables

Once the class has initialized the two variables with the images, program the instance to automatically switch the image it displays as it moves.

As these images alternate with each movement, it makes the instance appear more animated.

It is possible to program the switch between images without having to write many lines of code that associates each image to every single movement.

Write Actions in Pseudocode

Initially identify the actions by writing them in pseudocode.

Pseudocode expresses the tasks or operations for the instances to perform in a mix of Java language and plain English words. This helps us better understand what behaviors we want the instances to perform before we write the real code.

Example

image1 is displayed when the instance is created. When Duke makes his next movement, image2 should be displayed, and vice versa. This is expressed in an if-else statement.

```
if (current image displayed is image1) then
     use image2 now
else
     use image1 now
```

== Operator

The programming statements that instruct the instance to alternate between images contains an:

- if-else statement
- == operator (two equals signs)

The == operator:

- Is used in an if-statement to test whether two values are equal.
- Compares one value with another.
- Returns a boolean (true or false) result.

Remember that = is the assignment symbol, not the equals symbol.

Components of If-else Statement

Components of the if-else statement:

- Method `getImage` receives the instance's current image.
- The == operator checks that the value the instance displays is equal to image1.
  - If equal, then display image2
  - Else, display image1

If-else Statement Example

The if-else statement below is included in the act method to make the instance alternate display of two images as it moves forward.

```
/**
 * Act - do whatever the Duke wants to do. This method is called whenever
 * the 'Act' or 'Run' button gets pressed in the environment.
 */
public void act()
{
    if (getImage() == image1)
    {
        setImage(image2);
    }
    else
    {
        setImage(image1);
    }
    move(2);

}
```

End the Game

The Greenfoot class has a Stop method that you can use to end a game at a point that you designate.

You may want to end the game when:

- The player achieves a milestone.
- Time runs out on the clock.
- The instance touches a certain coordinate or object.

Example Duke Game

Example game:

- Let the player decide how many Code objects must be eaten by a keyboard-controlled Duke instance to end the game.
- When Duke eats that number of objects, the game ends with a sound that says, “Game Over”.

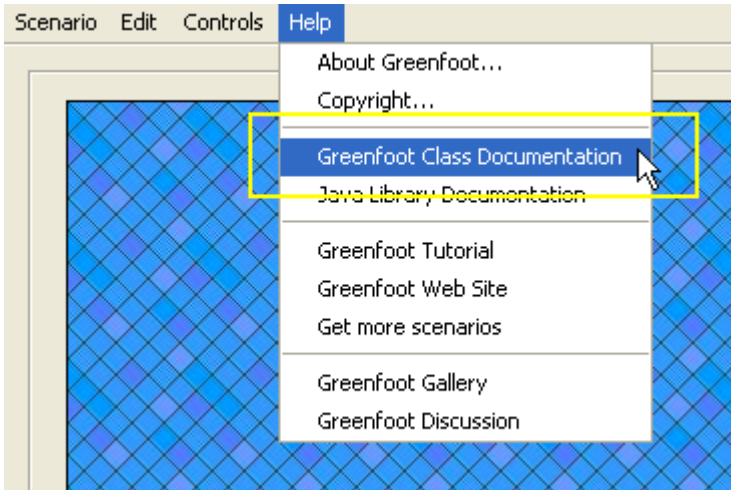
This programming includes:

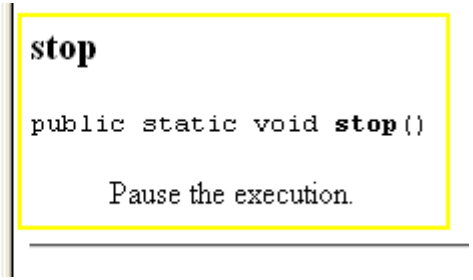
- Create and initialize variables to eat objects.
- Provide a count of the total objects eaten.
- Allow the player to enter the number of objects the instance should eat to win the game.
- Enter the `stop` method to stop the game when the required number of objects are eaten by the instance.

Find Stop Method in Greenfoot API

Follow these steps to view the Greenfoot class to find the `stop` method, which will be used to stop the execution of your game:

1. In the environment, select the Help menu.
2. Select Greenfoot Class Documentation.
3. In the Greenfoot API, scroll down and find the `stop` method.





Write Stop Method in Source Code

At the point that the game should end, write the method as follows into the source code. Take note that you are using dot notation to call the method.

```
Greenfoot.stop();
```

Assign Variables to Instances Example

In this example, Duke must eat a number of Code objects defined by the player to win the game. Notice the variables are defined before the constructors and methods. Also note that the Duke constructor assigns the variables to the instances produced by constructor.

```
public class Duke extends Animal
{
// counts how many code Duke has eaten.
private int codeEaten;
// the total amount of code that Duke needs to eat to win.
private int codeToWin;
// provides the balance of code Duke has eaten.
private int codeBalance;
/**
     * Create Duke and initialize codeEaten to zero.
     */
    public Duke(int codeEaten)
    {
        codeToWin = codeEaten;
        codeBalance = 0;
    }
```

lookForCode Defined Method Example

In this example there is a defined method below the Act method that tells the Duke object to look for Code objects. If the amount of code eaten equals the code to win, then the game is won and a sound is played.

```
/**
 * Look for code. If Duke finds code, he eats it. Otherwise, he does nothing.
 * If Duke wins, play a sound.
 */
public void lookForCode()
{
    if (canSee(Code.class))
    {
        eat(Code.class);
        codeEaten = codeEaten + 1;
    }
    if (codeEaten == codeToWin)
    {
        Greenfoot.playSound("GameOver.wav");
        Greenfoot.stop();
    }
}
```

Keyboard Controls in Act Method Example

In the Act method, note the keyboard controls, as well as the `lookForCode` defined method. If Duke finds code as he moves, he should eat it.

```
public class Duke extends Animal
{
    /**
     * Act - do whatever the Duke wants to do. This method is called whenever
     * the 'Act' or 'Run' button gets pressed in the environment.
     */
    public void act()
    {
        move(3);
        if (Greenfoot.isKeyDown("left"))
        {
            turn(-2);
        }
        if (Greenfoot.isKeyDown("right"))
        {
            turn(2);
        }
        lookForCode();
    }
```

Terminology

Key terms used in this lesson included:

Constructor

Defined variable

Pseudocode

Summary

In this lesson, you learned how to:

- Construct a world object using a constructor method
- Create an object using a constructor
- Write programming statements to use the new keyword
- Define the purpose and syntax of a variable
- Recognize the syntax to define and test variables
- Write programming statements to switch between two images
- Write programming statements to end a game

Practice

The exercises for this lesson cover the following topics:

- Creating a world constructor method
- Image animation
- Ending a game
- Concept and terminology review
- Journaling world creation and variable uses