ORACLE Academy

Working with Source Code and Documentation

What Will I Learn?

Objectives

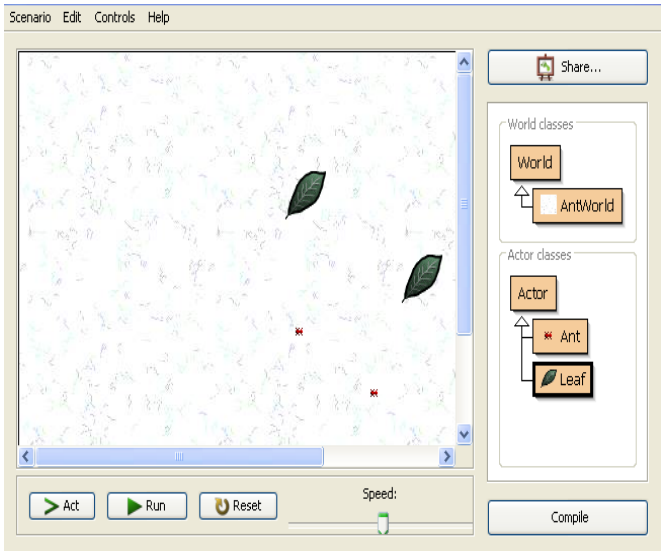
- Demonstrate source code changes to invoke methods programmatically
- Demonstrate source code changes to write an IF decision statement
- Describe a procedure to display object documentation

Why Learn It?

Purpose

In gaming, objects move on-command, either at the command of the person playing the game, or through code that tells the object how to move.

In Greenfoot, you need to write source code to make objects act and move in your game.



Source Code

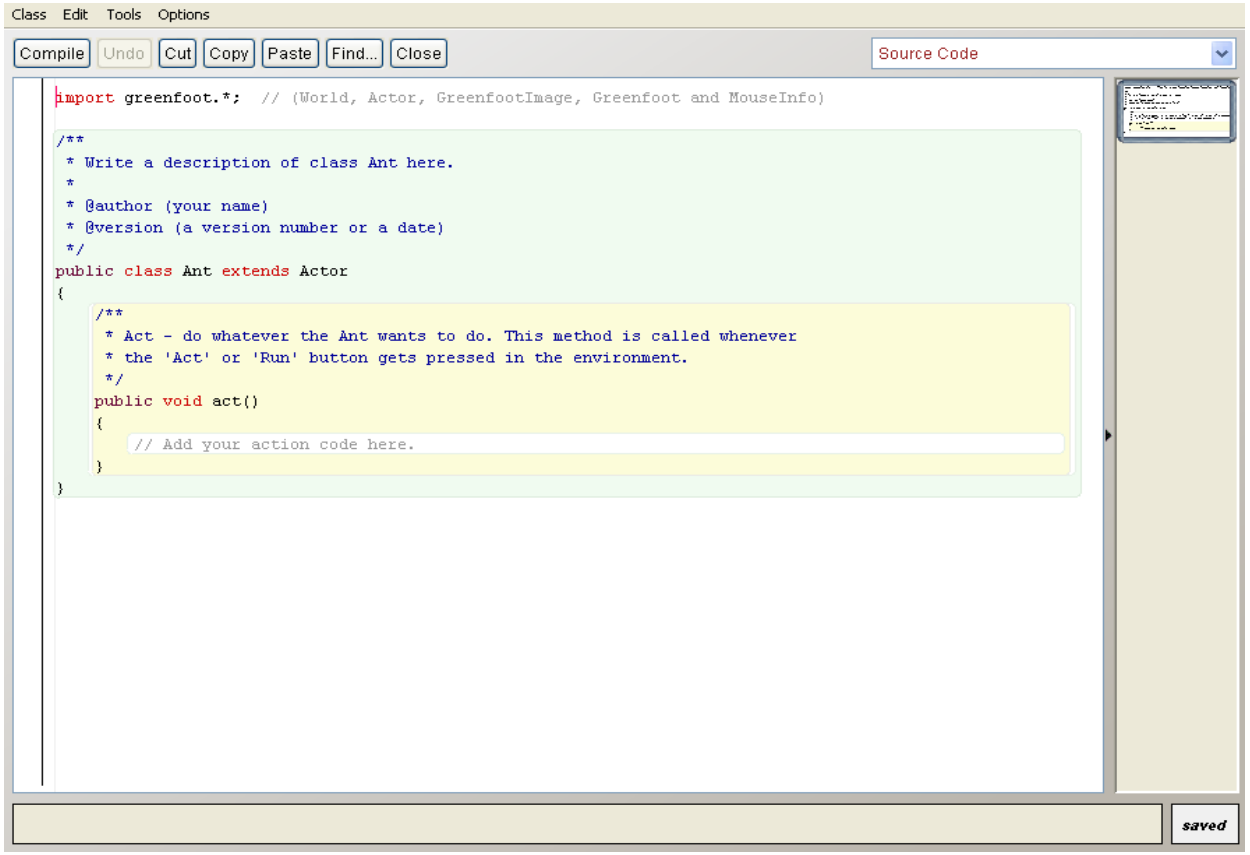
Source code is the blueprint or map that defines how your program functions. Without it, your objects could not move or interact.

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

/**
 * Write a description of class Alligator here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class Alligator extends Actor
{
    /**
     * Act - do whatever the Alligator wants to do. This method is called whenever
     * the 'Act' or 'Run' button gets pressed in the environment.
     */
    public void act()
    {
        // Add your action code here.
    }
}
```

Code Editor

Source code is managed in the code editor. To view the code editor, right click on any class in the environment, then select Open Editor from the menu.

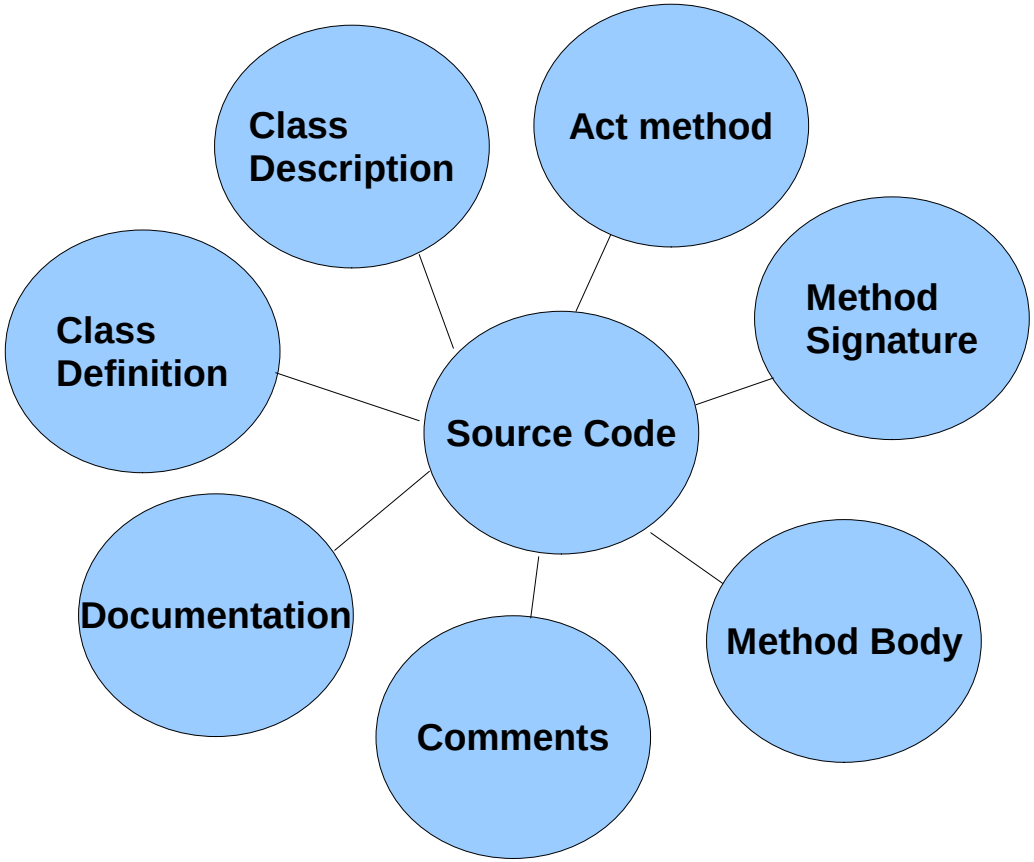


Functions of the Code Editor

In the code editor, you can:

- Write source code to program instances of the class to act.
- Modify existing source code to change an instance's behavior.
- Review a class's inherited methods and properties.
- Review methods created specifically for the class by the programmer who wrote the source code.

Components of Source Code



Class Description

The class description is a set of comments the programmer can modify to describe:

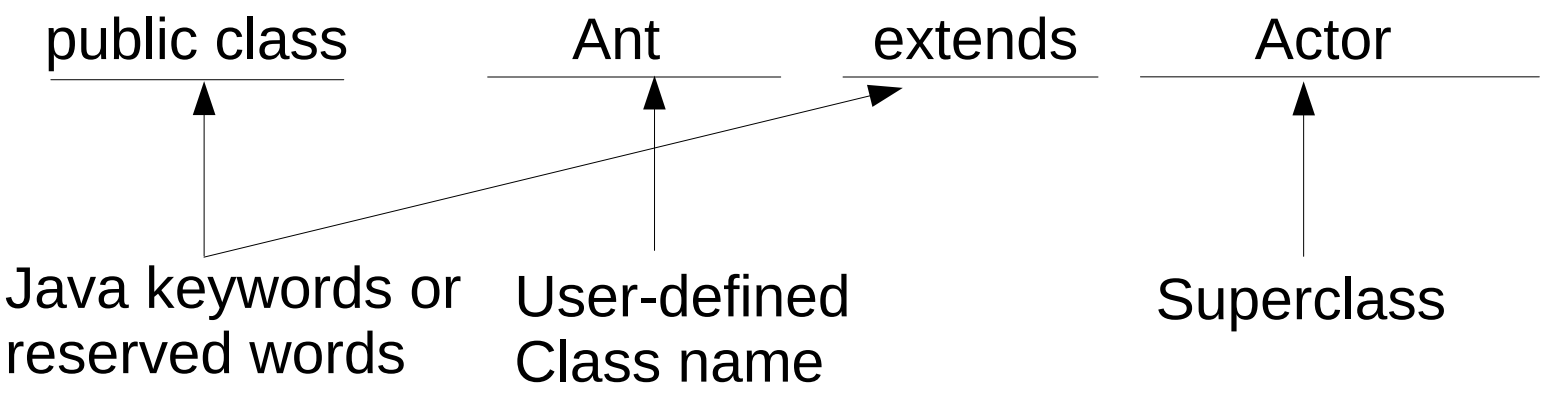
- What the class is and what it does
- Name of the person who authored the code
- The date it was last modified

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

/**
 * Write a description of class Ant here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class Ant extends Actor
{
    /**
     * Act - do whatever the Ant wants to do. This method is called whenever
     * the 'Act' or 'Run' button gets pressed in the environment.
     */
```

Class Definition

The class definition defines the class as follows:




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

/**
 * Write a description of class Ant here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class Ant extends Actor
{
    /**
     * Act - do whatever the Ant wants to do. This method is called whenever
     * the 'Act' or 'Run' button gets pressed in the environment.
     */
```

Act Method

The Act method is the part of the class definition that tells objects which methods to perform whenever the Act or Run execution controls are clicked in the environment.

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

/**
 * Write a description of class Ant here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class Ant extends Actor
{
    /**
     * Act - do whatever the Ant wants to do. This method is called whenever
     * the 'Act' or 'Run' button gets pressed in the environment.
     */
    public void act()
    {
        // Add your action code here.
    }
}
```

Defining Classes

The class definition is used to define:

- Fields and/or variables that store data persistently within an instance
- Constructors that initially set up an instance
- Methods that provide the behavior of an instance

Develop a strategy to be consistent when you define a class. For example:

- Define variables first
- Constructors second
- Methods third

Method Signature

The method signature:

- Describes what the method does
- Contains a return type, method name, and parameter list

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

/**
 * Write a description of class Ant here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class Ant extends Actor
{
    /**
     * Act - do whatev                    do. This method is called whenever
     * the 'Act' or 'Ru                   utton gets pressed in the environment.
     */
    public void act()
    {
        // Add y                r action code here.
    }
}
```

Method Name

Parameter List ()

Return Type

Comments

Comments:

- Describe what the source code does
- Do not impact functionality of the program
- Start with a forward slash and two asterisks `/**`
- Are in blue font (in Greenfoot)

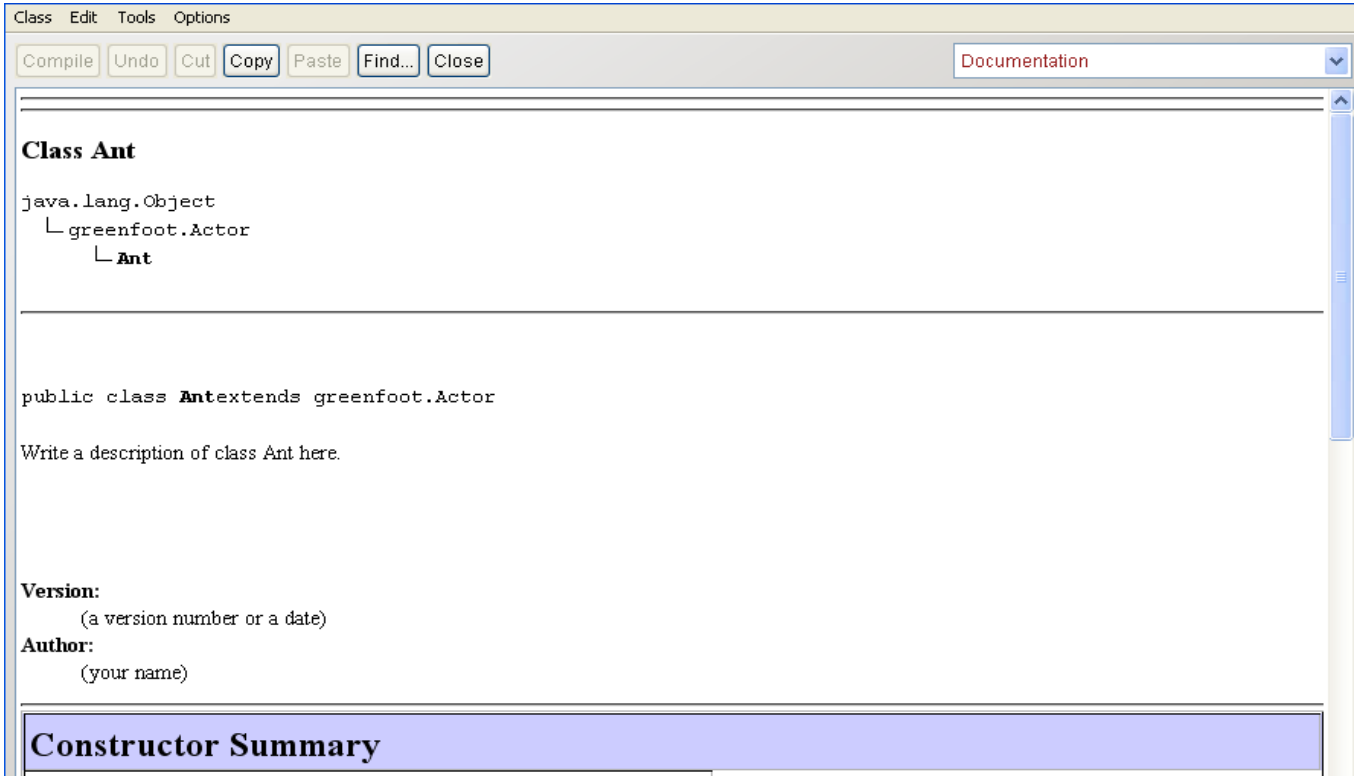
```
import greenfoot.*;  // (World, Actor, GreenfootImage, Greenfoot and MouseInfo)

/**
 * Write a description of class Ant here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class Ant extends Actor
{
    /**
     * Act - do whatever the Ant wants to do. This method is called whenever
     * the 'Act' or 'Run' button gets pressed in the environment.
     */
    public void act()
    {
        // Add your action code here.
    }
}
```

Documentation

Documentation:

- Describes the properties of the class
- Is accessible from the Documentation option in the source code's drop-down list



Invoke Methods Programmatically

In Greenfoot, instances do not move or react to keyboard commands automatically. One or more methods must be invoked to command instances to act in your game.

Invoke methods programmatically by writing each programming statement in the body of the Act method.

```
lic class Ant extends Actor

/**
 * Act - do whatever the Ant wants to do. This method is called whenever
 * the 'Act' or 'Run' button gets pressed in the environment.
 */
public void act()
{
    // Add your action code here.
}
```

Invoking Methods in Act Method

To invoke a method in the Act method:

- Enter each method call in sequential order in the space between the curly brackets.
- A method call's components include:
  - Return type: Variable to hold data returned by the method call (methods with void return types do not require variables and do not return data)
  - Method name
  - Parameter list: Indicates the type of arguments to invoke, if required
  - Semicolon: marks the end of the method call

```
public void act()
{
      move(10);
      turn(50);
}
```

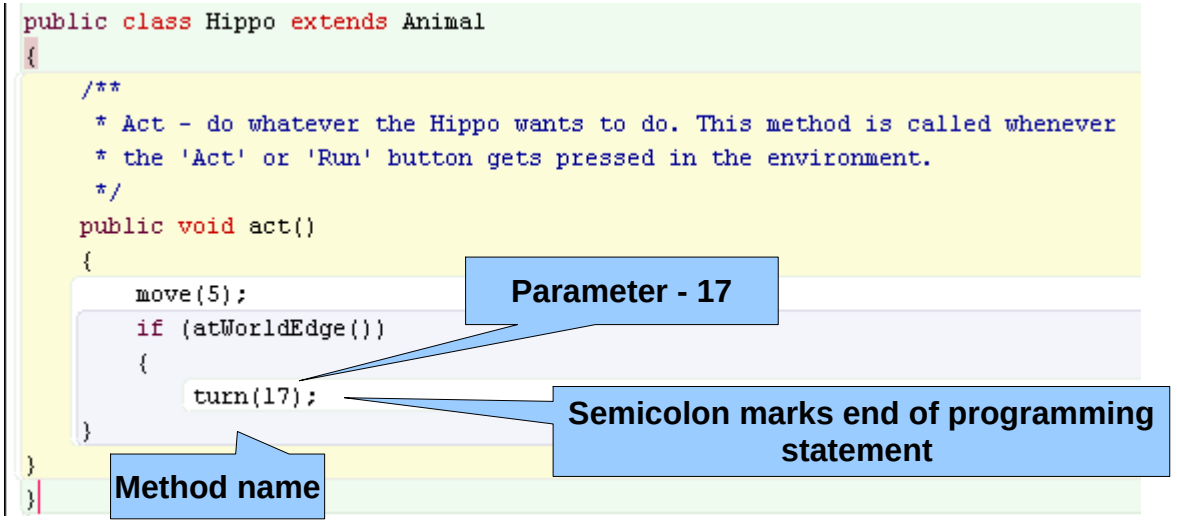
Semicolon

Method name

Parameters

Invoking Methods Example

The method call is written into the body of the Act method, ending with a semicolon. Each additional method call is typed directly underneath, until all methods are entered in the space between the curly brackets.



Method Examples

Here are examples of methods that instruct objects to perform actions:

Method Name	Description
void move(int distance)	Assign the object a number of steps to move, or the command to simply move when the Act or Run buttons are clicked.
void turn(int amount)	Assign the object a number of degrees to turn.
void act()	Act method that gives objects the opportunity to perform an action in the scenario. Method calls are inserted into this method.
void setLocation(int x, int y)	Assign a new location for this object.
void setRotation(int rotation)	Set a new rotation for this object.

View Available Methods

To view a list of all available methods to use in your game:

1. To view a list of all inherited methods, open a class's code editor, then view its documentation.
2. View the Greenfoot Class Documentation.
    1. Open Greenfoot.
    2. Select Help.
    3. Select Greenfoot Class Documentation.
3. View the Java Library Documentation.
    1. Open Greenfoot.
    2. Select Help.
    3. Select Java Library Documentation.

Sequential Tasks

Within a single task, multiple smaller, sub-tasks may be performed. For example, getting ready for school could require several tasks:

- Wake up
- Take a shower
- Brush your teeth
- Get dressed
- Eat breakfast
- ...

Even within those sub-tasks, there may be even more sub-tasks that take place (e.g., walking to school requires the left leg and right legs to move forward, in order).

Sequential Methods

Sequential methods make it possible for an object to perform tasks in sequence, such as run and then jump, or play a sound after something explodes.

Objects can be programmed in Greenfoot to perform sequential methods whenever the act button is clicked.

Sequential methods are multiple methods executed by Greenfoot in the order in which they are written in the program.

If-Then Relationships

Many things around us have a cause and effect relationship, or “if-then” relationship. Examples:

- If your cellphone rings, then you answer it. If it doesn't ring, then you do not answer it.
- If a flower starts to wilt, then you give it water. If the flower looks healthy, then you don't give it water.

If-decision Statements

An if-statement is written to tell your program to execute a set of programming statements only if and when a certain condition is true.

General form of an if statement:

```
if (condition)
{
    instruction;
    instruction;
    …
}
```

If-decision Statement Components

The If-statement contains a condition, which is a true or false expression, and one or more method calls that are executed if the condition is met.

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

/**
 * Write a description of class Duke here.
 *
 * @author Oracle Academy
 * @version S03_L02_S01
 */
public class Duke extends Actor
{
    /**
     * Act - do whatever the Duke wants to do. This method is called whenever
     * the 'Act' or '   s pressed in the environment.
     */
    public void act()
    {
        move(1);
        if (Greenfoot.isKeyDown("left"))
        {
            turn(-2);
        }
        if (Greenfoot.isKeyDown("right"))
        {
            turn(2);
        }
    }
}
```

Condition

Method Calls

If-decision Statement Example

In the following code example:

- If the left arrow key is pressed on the keyboard, the object turns left. If the right arrow key is pressed, the object turns right.
- If the condition is false, the method calls defined in the if-statement are not executed.
- The move method is executed regardless of the if-statement.

```
public class Duke extends Actor
{
    /**
     * 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(1);
        if (Greenfoot.isKeyDown("left"))
        {
            turn(-2);
        }
        if (Greenfoot.isKeyDown("right"))
        {
            turn(2);
        }
    }
}
```

IsKeyDown Method

The `isKeyDown` method:

- Is a pre-existing Greenfoot method that listens to determine if a keyboard key is pressed during program execution.
- Is called in a class using dot notation

When a method is not in the class or inherited by the class you are programming, specify the class or object that has the method before the method name, then a dot, then the method name. This technique is called dot notation.

Object Orientation in the Real World

As we move about the world we live in, it's important for us to know our orientation, or sense of direction.

- When you drive a car, you always need to know if your car is in the correct lane of the road.
- When a plane flies through the air, it needs to know where it's located relative to other planes, so a collision doesn't occur.
- When you enter your location on a map in a cellphone, you receive coordinates that tell you where you are, and the address.

Display an Object's Orientation

Methods can tell us how an object is positioned in the world, relative to itself and other objects. You can invoke a method:

- With a specific data type (e.g., boolean method) to ask the object a question about it's orientation.
- In the environment to learn how the object is oriented in the scenario.

Methods to Verify an Object's Orientation

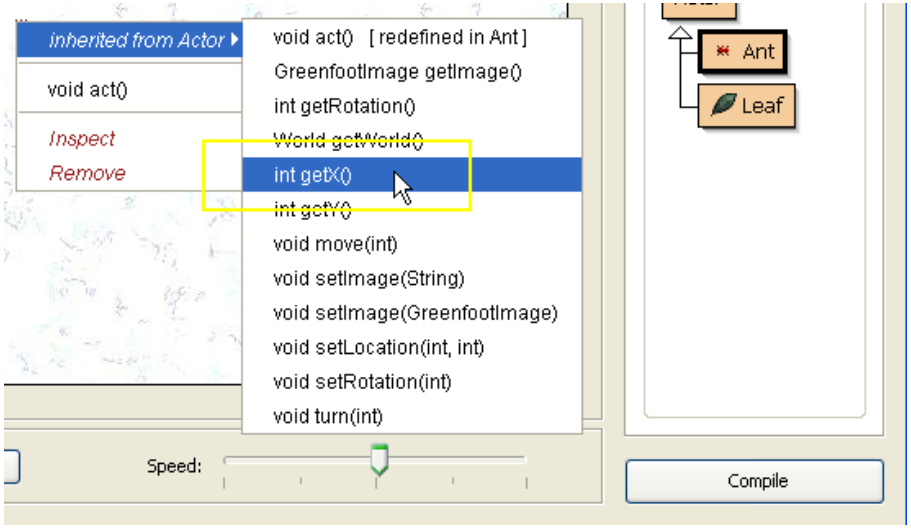
Here are some methods that return information about an object's orientation.

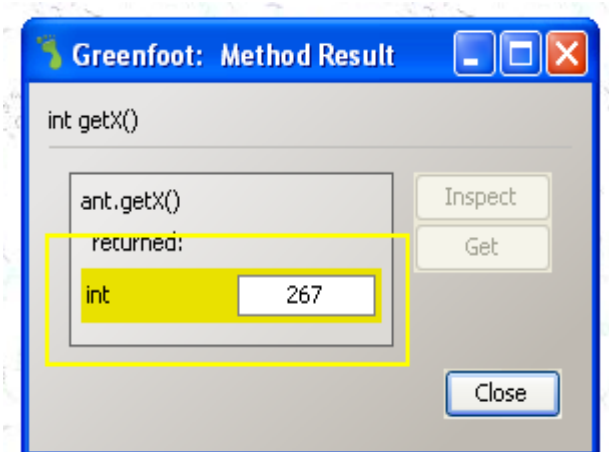
Method Name	Description
int getRotation()	Returns the current rotation of the object.
World getWorld()	Returns the name of the world the object is in.
int getX()	Returns the x-coordinate of the object's current location.
int getY()	Returns the y-coordinate of the object's current location.

Display an Object's Orientation

Follow these steps to display an object's orientation:

1. Right click on the instance in the world.
2. Select *inherited from Actor* to view its methods.
3. Invoke a method with a specific data type to ask the object a question about its orientation.
4. The method result will display. Note the value returned, then click Close.





Terminology

Key terms used in this lesson included:

Class description

Comments

If decision statements

Invoking a method

Object oriented analysis

Sequential methods

Summary

In this lesson, you learned how to:

- Demonstrate source code changes to invoke methods programmatically
- Demonstrate source code changes to write an IF decision statement
- Describe a procedure to display object documentation

Practice

The exercises for this lesson cover the following topics:

- Modifying source code to turn actors
- Modifying source code to include a decision