Defining Methods

What Will I Learn?

Objectives

- Describe effective placement of methods in a super or subclass
- Simplify programming by creating and calling defined methods

Why Learn It?

Purpose

Efficiency in programming derives from the concept of using pre-written code to save time and resources.

Programmers simplify their programs by using existing methods rather than re-writing a method every time it is needed.

Efficient Placement of Methods

At times, many lines of code are required to program a behavior, such as turning when the instance reaches the edge of the world, or eating other objects.

To efficiently write methods, saving time and lines of code:

- Define a new method for an action below the code for the act method. Then in the act method issue a call to invoke the new method created.
- Define new methods in a superclass to allow their subclasses to automatically inherit the methods.

Defined Methods

Defined methods are new methods created by the programmer. These methods:

- Can be executed immediately, or stored and called later.
- Do not change the behavior of the class when stored.
- Separate code into shorter methods, making it easier to read.

Defined methods create a new method that a class didn't already possess. These methods are written in the class's source code below the act method.

Define a New Method

Follow these steps to define a new method:

1. Select a name for the method.
2. Open the code editor for the class that will use the method.
3. Add the code for the method definition below the act method.
4. Call this new method from the act method, or store it for use later.

Turn at the Edge of the World

Problem:

- Animal instances are stopping when they reach the edge of the world.
- These objects should turn and keep moving when they reach the edge of the world.

Solution:

- Define a method in the Animal superclass so that all instances of Animal subclasses have the ability to turn when they reach the edge of the world.
- Call the new defined method in an Animal subclass so that its objects will turn when they reach the edge of the world.

Test an Object's Position in the World

To test if an object is near the edge of the world, this requires:

- Multiple boolean expressions to express if one or both conditions are true or false.
- Use logic operators to connect the boolean expressions used to evaluate a condition.

Logic Operators

Logic operators can be used to combine multiple boolean expressions into one boolean expression.

Logic Operator	Means	Definition
Exclamation Mark (!)	NOT	Reverses the value of a boolean expression (if b is true, !b is false. If b is false, !b is true).
Double ampersand (&&)	AND	Combines two boolean values, and returns a boolean value which is true if and only if both of its operands are true.
Two lines (II)	OR	Combines two boolean variables or expressions and returns a result that is true if either or both of its operands are true.

Define atWorldEdge Method in Superclass

Write the code for the `atWorldEdge` method in the Animal superclass, below the act method. Compile the code and then close the code editor.

```
/**
 * Test if we are close to one of the edges of the world. Return true is we are.
 */
public boolean atWorldEdge()
{
    if(getX() < 20 || getX() > getWorld().getWidth() - 20)
        return true;
    if(getY() < 20 || getY() > getWorld().getHeight() - 20)
        return true;
    else
        return false;
}
```

Call atWorldEdge Method in Subclass

Open the code editor for an Animal subclass. Create an if-statement that calls the new defined method as a condition. The condition tells the instance how many degrees to turn when the condition is true. Compile the code and run the scenario to test it.

```
public class Bee extends Animal
{
    /**
     * Act - do whatever the Bee wants to do. This method is called whenever
     * the 'Act' or 'Run' button gets pressed in the environment.
     */
    public void act()
    {
        if (atWorldEdge())
        {
            turn(19);
        }
        move(2);
    }
}
```

Class Documentation

The Animal class documentation shows the new atWorldEdge method after its defined. All subclasses of the animal superclass inherit this method.

Method Summary	
void	**act**() Act - do whatever the Animal wants to do.
boolean	**atWorldEdge**() Test if object is close to one of the edges of the world.

Methods inherited from class
addedToWorld, getImage, getIntersectingObjects, getNeighbours, getObjectsAtOffset, getObjectsInRange, getOneIntersectingObject, getOneObjectAtOffset, getRotation, getWorld, getX, getY, intersects, move, setImage, setImage, setLocation, setRotation, turn

Methods inherited from class
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Defined Method to Eat Objects

In a game, if a predator can see its prey, we can create code so it eats the prey. Create a defined method in the act method of the Animal superclass called `canSee` to enable all Animal subclasses' objects to eat other objects.

To create this defined method:

- Declare a variable for the prey.
- Use an assignment operator to set the value of the variable equal to the return value of the `getOneObjectAtOffset` method.

Define canSee Method

Define a canSee Method in the Animal superclass that returns true if the predator object (in this example, Duke) lands on a prey object (in this example, Code).

```
public class Animal extends Actor
{
    /**
     * Act - do whatever the Animal 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.
    }
    /**
     * Return true if Duke can see an object of the Code class 'clss'. Return false if there is no Code object here.
     */
    public boolean canSee(Class clss)
    {
        Actor actor = getOneObjectAtOffset(0, 0, clss);
        return actor != null;
    }
}
```

Define eat Method

Create a defined method in the Animal superclass called `eat`. This method will instruct Duke to eat the Code object if he lands on it.

```
/**
 * Eat an object of class 'clss' if there is such an object in our current location.
 * Otherwise do nothing.
 */
public void eat(Class clss)
{
    Actor actor = getOneObjectAtOffset(0, 0, clss);
    if(actor != null) {
        getWorld().removeObject(actor);
    }
}
```

Define lookForCode Method

Create a defined method in the animal subclass (in this example, the Duke class) called `lookForCode` that checks if Duke has landed on a Code object. If so, Duke will eat the Code object.

```
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()
    {
        // Add your action code here.
    }
    /**
     * Look for code. If Duke finds code, he eats it. Otherwise, he does nothing.
     */
    public void lookForCode()
    {
        if (canSee(Code.class))
        {
            eat(Code.class);
        }
    }
}
```

Call lookForCode in Act Method

Call the new `lookForCode` method in Duke's act method. Run the animation to test the code.

```
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:

Defined methods

Summary

In this lesson, you learned how to:

- Describe effective placement of methods in a super or subclass
- Simplify programming by creating and calling defined methods

Practice

The exercises for this lesson cover the following topics:

- Examining and simplifying programming