Object Oriented Programming

Week 4 Part 1
Relationships between Classes

Lecture

- What are relationships
- Types of relationships
- UML description of relationships
- Implementing relationships in Java

What are Relationships in OOP?

Classes and Relationships

- Objects let us reason about programs as if they were constructed with things.
- Classes let us define types of objects.
  - Relationships show the way types of objects interaction
- Generalization is only one type of relationship
  - It indicates *is a kind of*
    - A dog is a kind of mammal
    - A mammal is a kind of animal
  - It is represented in Java by inheritance

Other types of relationships

- There are many other types of relationships.
- Three of these are
  - Association: a general type of relationship
    - E.g. a dog chases a cat
  - Aggregation: a group of individual objects forming another object
    - E.g. A class: each student is an individual
  - Composition: a group of objects that exist only to comprise another object
    - E.g. A student: a student has test scores and grades, but these have no existence without the student

Associations

Associations

- Associations capture all of the myriad ways two types of things may relate to each other:
  - For example
    - A Student studies a Subject
    - A Car drives on a Road
    - A Rock lies on the ground
- Association capture relationships that define a model
  - They exists as a representation of the world to be captured

Associations have Properties

- We can categorize associations between classes such as
  - Directionality
  - Cardinality
- The properties a general characteristics of the associate; not properties of the particular relationship
- The properties indicate how they are to be implemented

Properties of Associations

- Directionality
  - *Uni-directional*: One object may access the other, but the other cannot
  - *Bi-directiona*l: both objects can access each other
- Cardinality
  - 1-1: each object is associated with one other object
    - E.g., A Student attempts A Test
  - 1-*: each object is associated with many other objects
    - E.g., A Student receives Grades
  - *-*: many objects are associate with many other objects
    - E.g., Students take Classes

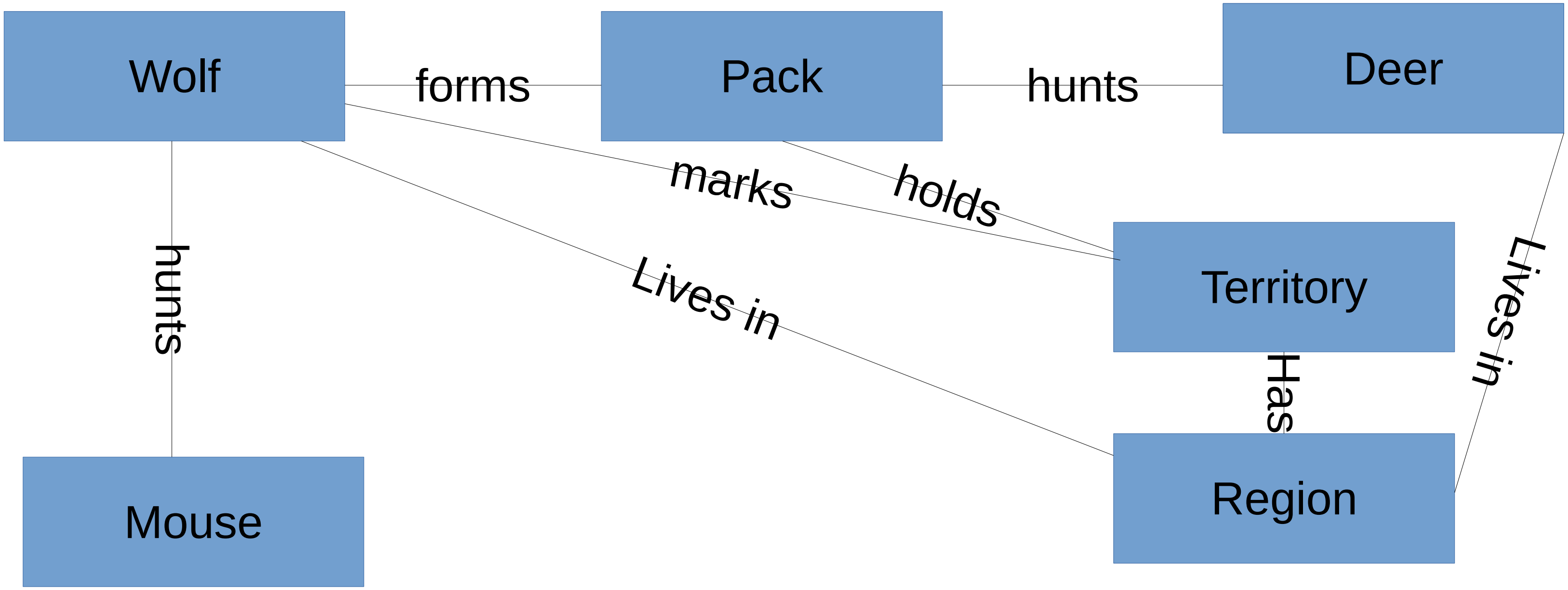
Properties depend on the Model

- Associations model relationships in the real world
- The relationships in our model are simplifications
  - The type of association depends on the aspect of the world we are trying to capture
  - For example
    - A Student takes a Class: is a 1-1 relationship perhaps to capture progress
    - A Student takes Classes: is a 1-* relationship that perhaps captures a schedule
    - Students take Classes: is a *-* relationship that perhaps captures timings

Example: Wolves

- As an example, lets create a model of wolves.
  - Perhaps to create a program to track wild wolves.
- We capture facts about wolves
  - Wolves form packs
  - Wolves mark territory
  - Wolves hunt mice
  - Wolf Packs defend territory
  - Wolf Packs hunt deer
  - Territories are in regions

Wolf Model

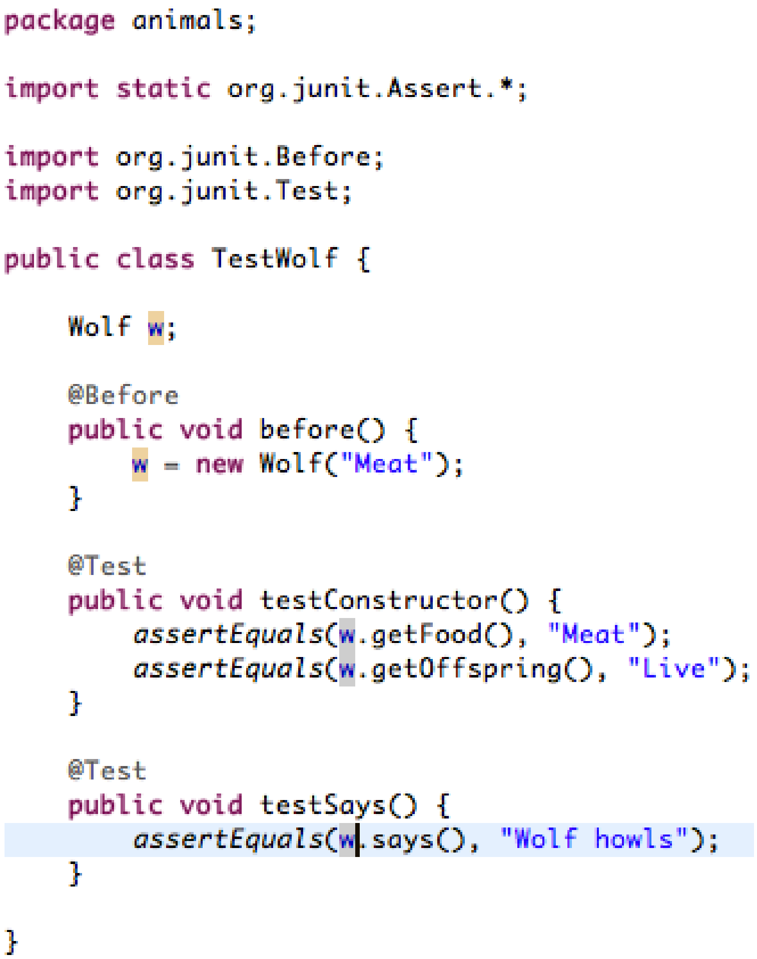

Wolf
forms
Pack
hunts
Deer
marks
holds
Territory
hunts
Lives in
Lives in
Has
Mouse
Region

Wolf hunts Mouse

- Wolves as a class hunt mice as a class, but any particular hunt is a single wolf
  - Wolf hunts mouse is a 1-1 relationship.
- It may be modeled as a uni-directional relationship
  - Wolf hunts Mouse
  - Indicates that in this model the Mouse need know nothing of the wolf
- If may be modeled as a bi-directional relationship
  - Wolf hunts Mouse and Mouse is hunted by Wolf
  - Indicates that the Mouse in the model needs to know of the wolf

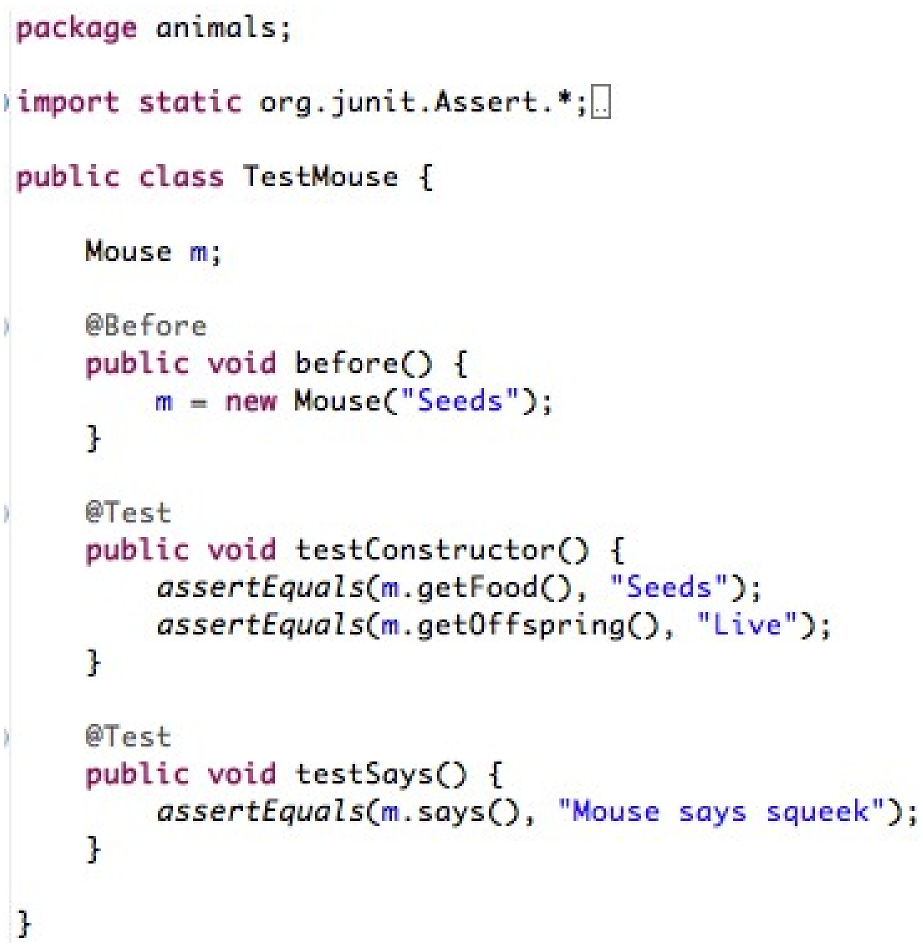
Mouse and Wolf tests

- TestWolf



```java
package animals;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

public class TestWolf {

    Wolf w;

    @Before
    public void before() {
        w = new Wolf("Meat");
    }

    @Test
    public void testConstructor() {
        assertEquals(w.getFood(), "Meat");
        assertEquals(w.getOffspring(), "Live");
    }

    @Test
    public void testSays() {
        assertEquals(w.says(), "Wolf howls");
    }

}
```

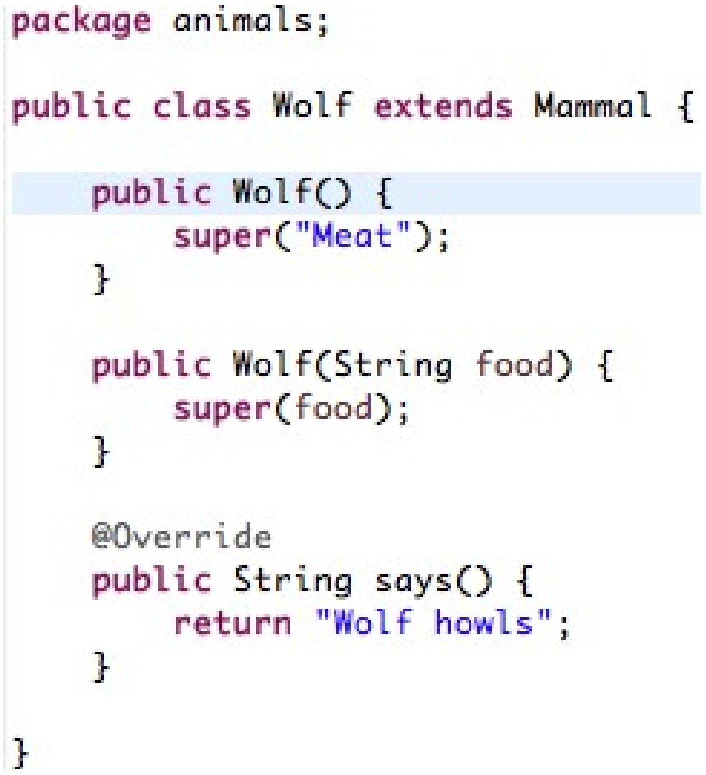
- Test Mouse



```java
package animals;

import static org.junit.Assert.*;

public class TestMouse {

    Mouse m;

    @Before
    public void before() {
        m = new Mouse("Seeds");
    }

    @Test
    public void testConstructor() {
        assertEquals(m.getFood(), "Seeds");
        assertEquals(m.getOffspring(), "Live");
    }

    @Test
    public void testSays() {
        assertEquals(m.says(), "Mouse says squeek");
    }

}
```

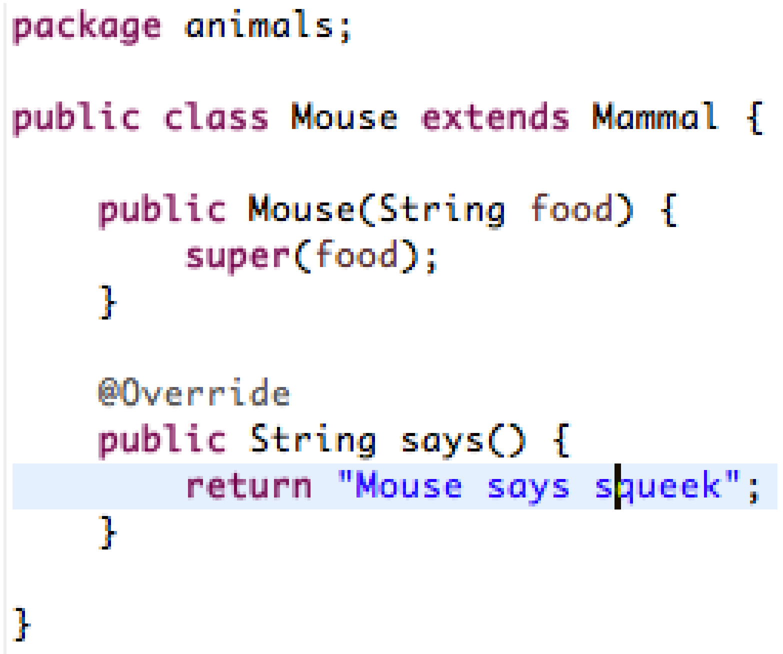
Base Wolf and Mouse Classes

- Wolf



```java
package animals;

public class Wolf extends Mammal {

    public Wolf() {
        super("Meat");
    }

    public Wolf(String food) {
        super(food);
    }

    @Override
    public String says() {
        return "Wolf howls";
    }

}
```

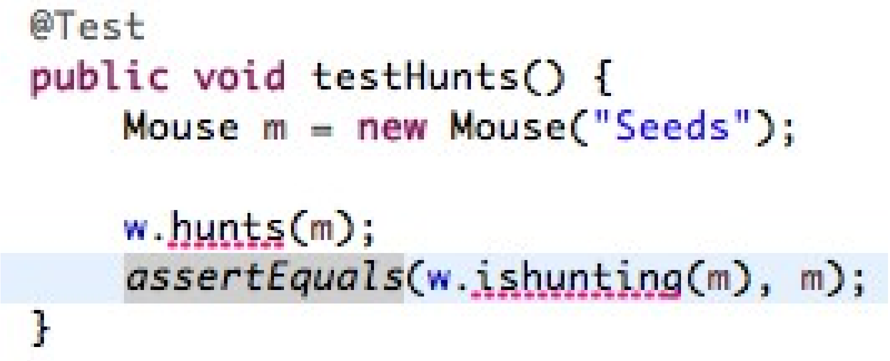
- Mouse



```java
package animals;

public class Mouse extends Mammal {

    public Mouse(String food) {
        super(food);
    }

    @Override
    public String says() {
        return "Mouse says squeek";
    }

}
```

Test for Uni-Direction Association

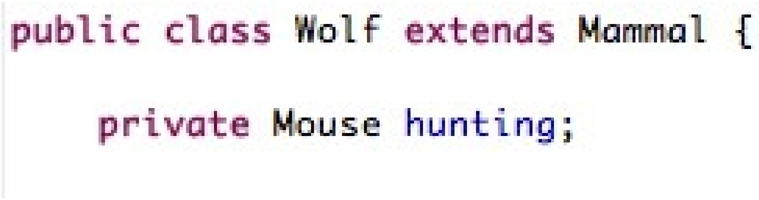
- The Uni-directional association can be implemented as
  - A new field to hold the mouse being hunted
  - Getters and setters for the field
- Add a new test, testHunts(), to TestWolf.java



```
@Test
public void testHunts() {
    Mouse m = new Mouse("Seeds");

    w.hunts(m);
    assertEquals(w.ishunting(m), m);
}
```

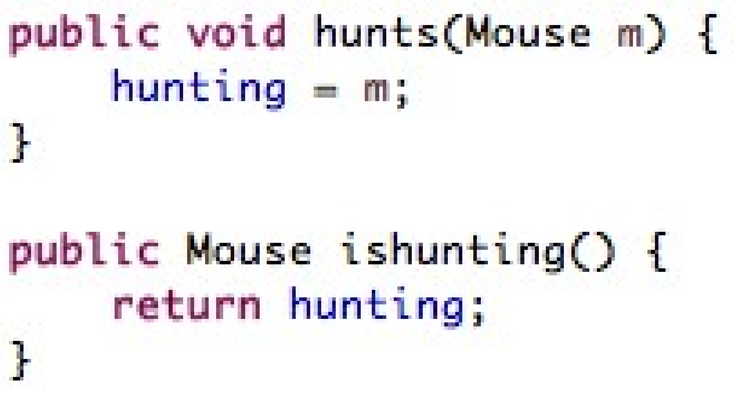
Add Field and Method from Test

- Field



```
public class Wolf extends Mammal {

    private Mouse hunting;
```

- Methods



```
public void hunts(Mouse m) {
    hunting = m;
}

public Mouse ishunting() {
    return hunting;
}
```

- The association is
  - One direction: the Mouse object cannot access the Wolf object that is hunting it
  - 1-*:
    - The Mouse is hunted by only one wolf
    - The Wolf hunts many mice.

Adding Pack

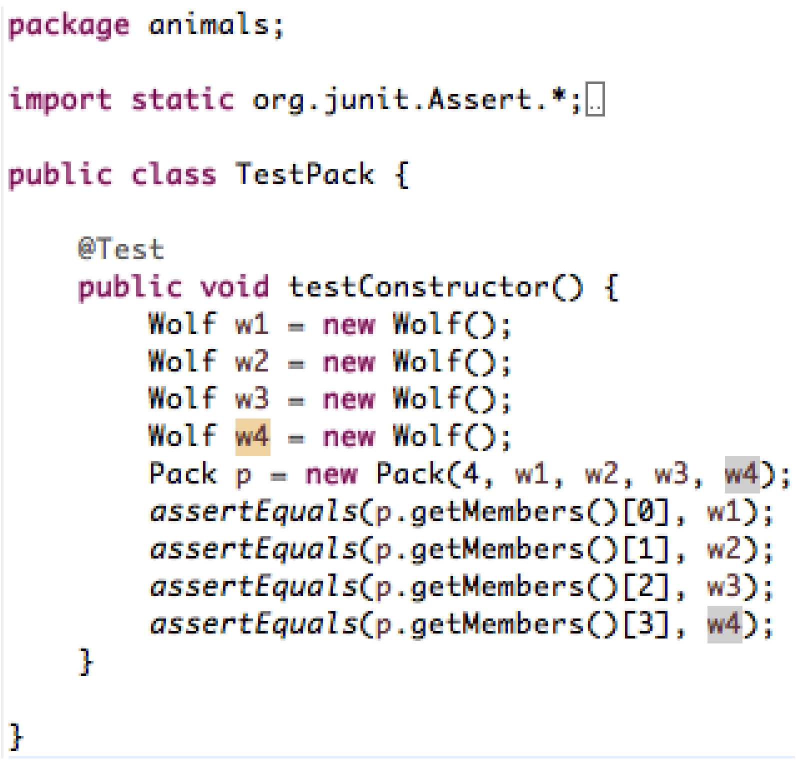
- A Pack is a group of wolves
- We will create a new object called Pack which
  - Contains a number of wolves
  - Each wolf has a dominance position in the Pack
  - Relationship between the Wolves and Packs is Many to one
    - One Wolf is only in one Pack
    - One Pack may have many wolves.
  - Relationship between wolves and packs is bi-directional
    - Each pack knows which wolves are in it
    - Each wolf knows which pack it is a member of

Bi-directional link

- The Wolf will have a public method which return the pack to which it belongs
  - The Wolf will be born into a Pack
    - The constructor will set the Pack
- The Pack will have a public method which returns the members of the Pack
  - The Pack consists of the Wolves
    - The construction will set initial Wolves in the Pack
    - A public method will allow an additional Wolf to be added

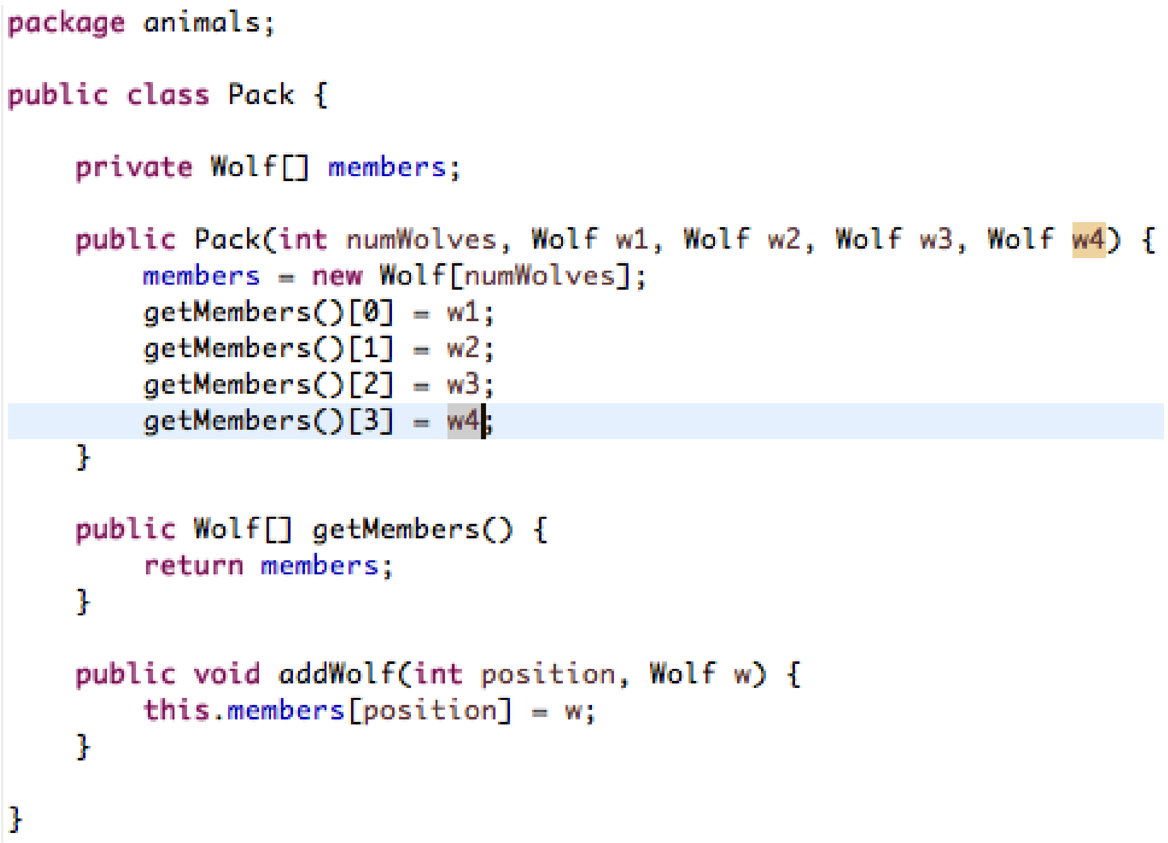
TestPack and Pack

- TestPack



```
package animals;

import static org.junit.Assert.*;

public class TestPack {

    @Test
    public void testConstructor() {
        Wolf w1 = new Wolf();
        Wolf w2 = new Wolf();
        Wolf w3 = new Wolf();
        Wolf w4 = new Wolf();
        Pack p = new Pack(4, w1, w2, w3, w4);
        assertEquals(p.getMembers()[0], w1);
        assertEquals(p.getMembers()[1], w2);
        assertEquals(p.getMembers()[2], w3);
        assertEquals(p.getMembers()[3], w4);
    }

}
```

- Pack



```
package animals;

public class Pack {

    private Wolf[] members;

    public Pack(int numWolves, Wolf w1, Wolf w2, Wolf w3, Wolf w4) {
        members = new Wolf[numWolves];
        getMembers()[0] = w1;
        getMembers()[1] = w2;
        getMembers()[2] = w3;
        getMembers()[3] = w4;
    }

    public Wolf[] getMembers() {
        return members;
    }

    public void addWolf(int position, Wolf w) {
        this.members[position] = w;
    }

}
```

Directionality

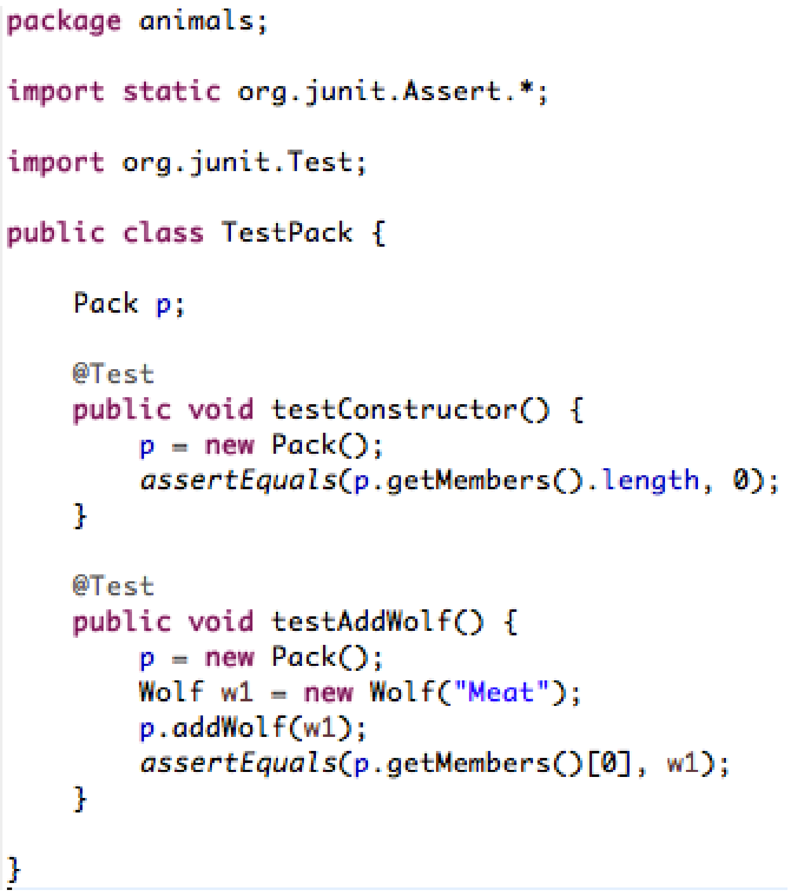
- So far, the implementation is only uni-directional
  - The Pack knows the members, but the Wolf does not know what Pack it is a member of
  - To implement bi-directionality, we need to give the Wolf access to its Pack
- The Wolf is born into a Pack, so we will alter the constructor to insert the Wolf
- A Wolf may change its Pack, so we will need to add a setter

First Refactor Pack

- To create a single Wolf with a new Pack, we need to add a Pack with no members.
- This shows a flaw in our original design
  - We need to create an empty pack, then add wolves
- We will refactor to add a constructor with no wolves.
- Finally, we will need to be able to add individual wolves

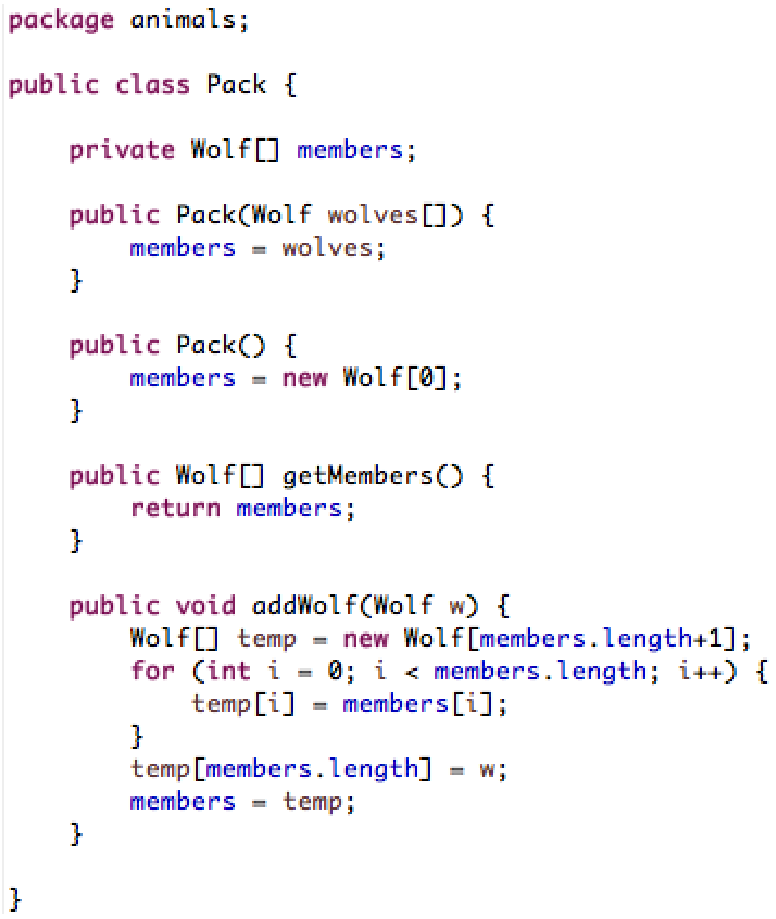
Refactor TestPack and Pack

- TestPack



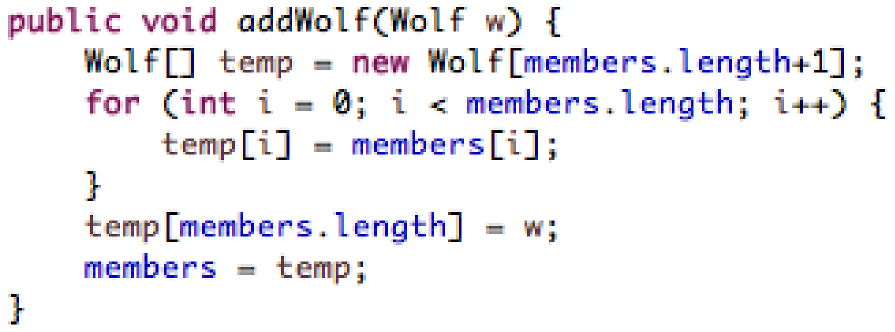
```
package animals;

import static org.junit.Assert.*;

import org.junit.Test;

public class TestPack {

    Pack p;

    @Test
    public void testConstructor() {
        p = new Pack();
        assertEquals(p.getMembers().length, 0);
    }

    @Test
    public void testAddWolf() {
        p = new Pack();
        Wolf w1 = new Wolf("Meat");
        p.addWolf(w1);
        assertEquals(p.getMembers()[0], w1);
    }

}
```

- Pack



```
package animals;

public class Pack {

    private Wolf[] members;

    public Pack(Wolf wolves[]) {
        members = wolves;
    }

    public Pack() {
        members = new Wolf[0];
    }

    public Wolf[] getMembers() {
        return members;
    }

    public void addWolf(Wolf w) {
        Wolf[] temp = new Wolf[members.length+1];
        for (int i = 0; i < members.length; i++) {
            temp[i] = members[i];
        }
        temp[members.length] = w;
        members = temp;
    }

}
```

Java Arrays



```
public void addWolf(Wolf w) {
    Wolf[] temp = new Wolf[members.length+1];
    for (int i = 0; i < members.length; i++) {
        temp[i] = members[i];
    }
    temp[members.length] = w;
    members = temp;
}
```

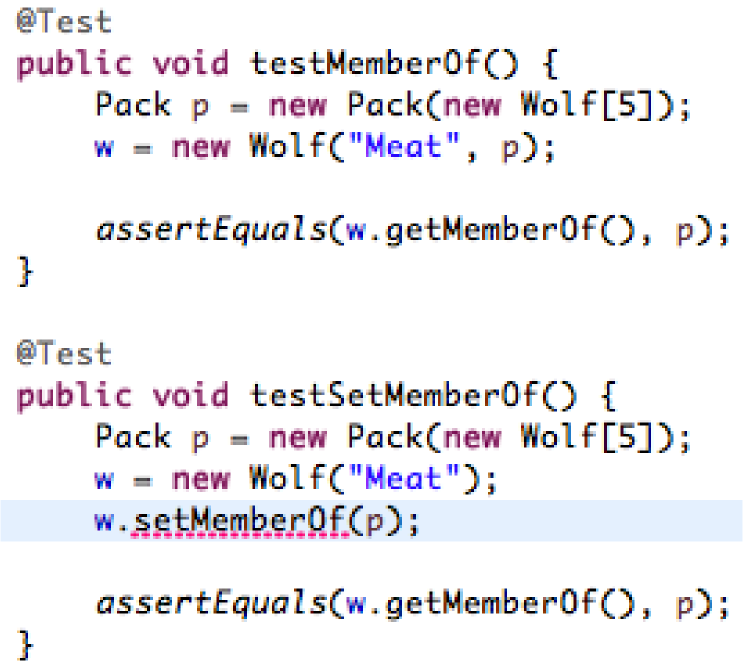
- Java Arrays differ from C
  - They are objects
    - The have a length member
    - For loops can rely on always having the length of the array

Now we can add Bi-directionality

- Add a new field: memberOf
- Add a new constructor: Wolf(String, Pack)
- Add a getter: getMemberOf()
- Add a setter: setMemberOf(Pack)

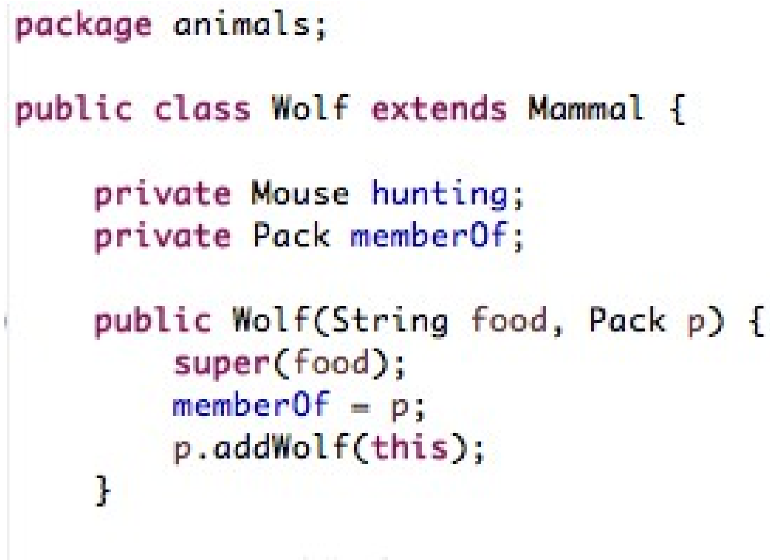
Add Bi-directionality to Wolf

- Add to TestWolf



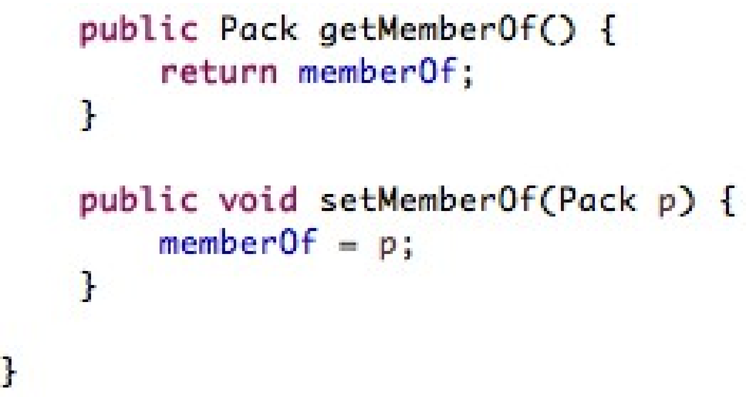
```
@Test
public void testMemberOf() {
    Pack p = new Pack(new Wolf[5]);
    w = new Wolf("Meat", p);

    assertEquals(w.getMemberOf(), p);
}

@Test
public void testSetMemberOf() {
    Pack p = new Pack(new Wolf[5]);
    w = new Wolf("Meat");
    w.setMemberOf(p);

    assertEquals(w.getMemberOf(), p);
}
```

- Add to Wolf



```
package animals;

public class Wolf extends Mammal {

    private Mouse hunting;
    private Pack memberOf;

    public Wolf(String food, Pack p) {
        super(food);
        memberOf = p;
        p.addWolf(this);
    }
```

…



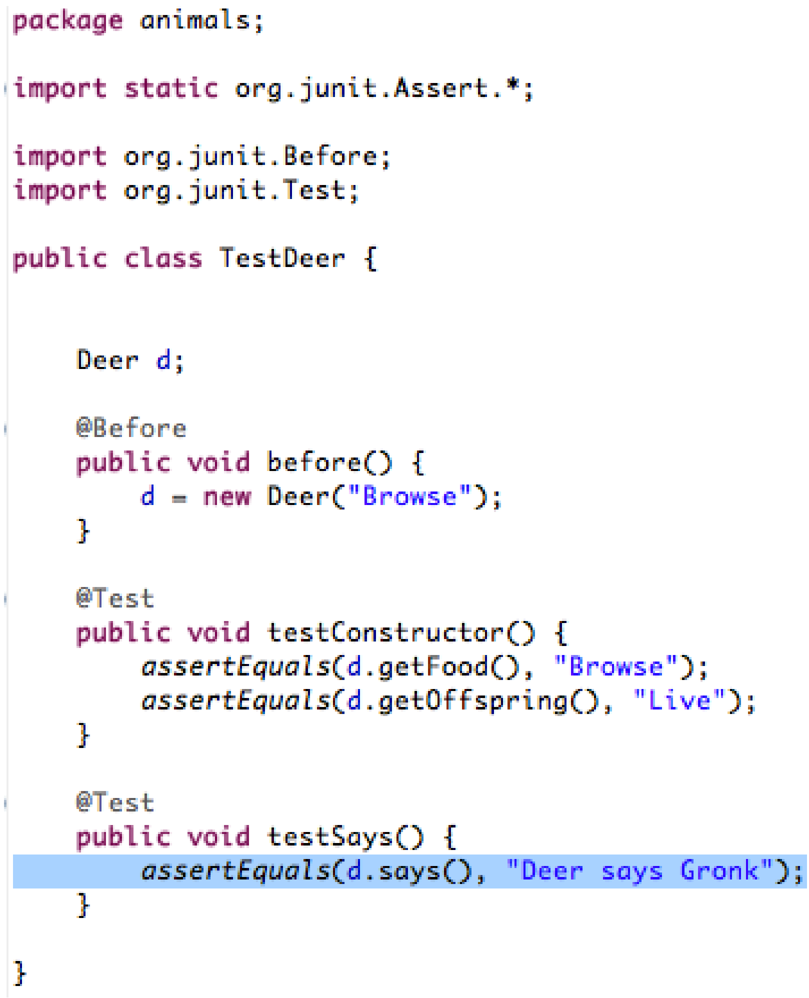
```
    public Pack getMemberOf() {
        return memberOf;
    }

    public void setMemberOf(Pack p) {
        memberOf = p;
    }
}
```

Adding Deer and Dear hunting

- Packs hunt deer, but individuals wolves do not
  - Deer are too hard for an individual to catch
  - They are too big for an individual to eat
- There is a hunts associate between the pack and the deer
- The hunts associate is a 1-* uni-directional associate, just as is the hunts association between Wolf and Mouse
  - A single Pack hunts Deer

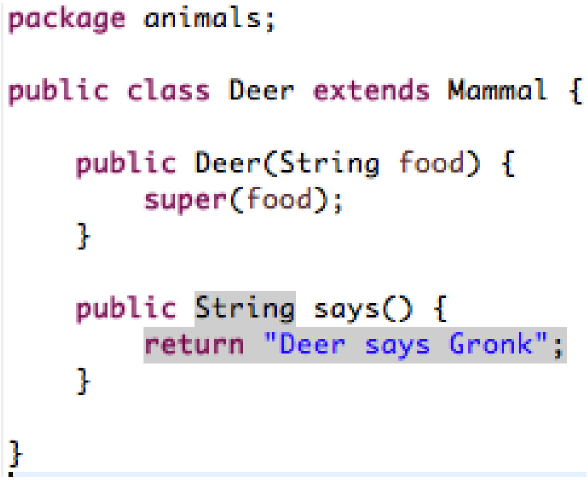
Adding Deer

- TestDeer



```
package animals;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

public class TestDeer {


    Deer d;

    @Before
    public void before() {
        d = new Deer("Browse");
    }

    @Test
    public void testConstructor() {
        assertEquals(d.getFood(), "Browse");
        assertEquals(d.getOffspring(), "Live");
    }

    @Test
    public void testSays() {
        assertEquals(d.says(), "Deer says Gronk");
    }

}
```

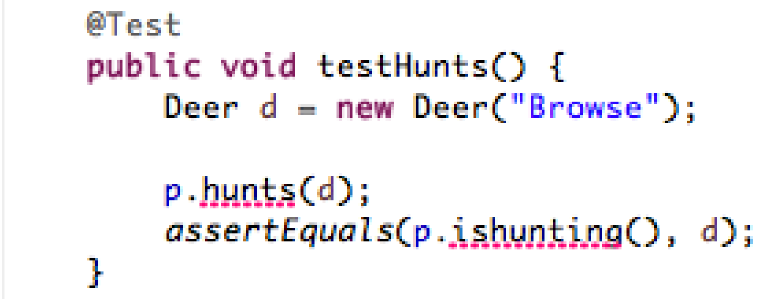
- Deer



```
package animals;

public class Deer extends Mammal {

    public Deer(String food) {
        super(food);
    }

    public String says() {
        return "Deer says Gronk";
    }

}
```

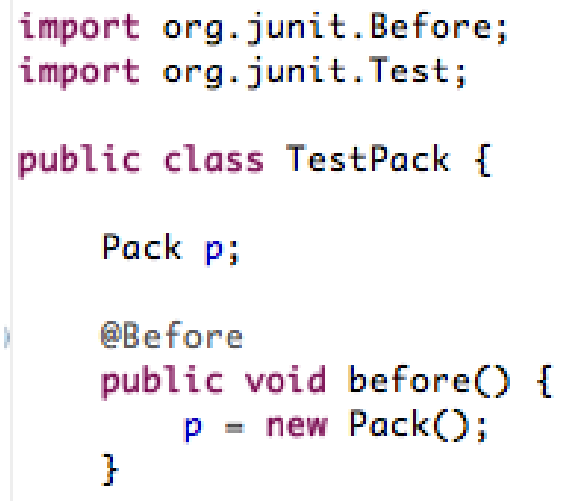
Updating Pack

- Add to TestPack



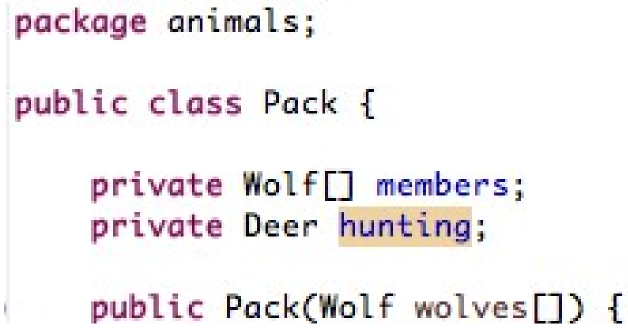
```
    @Test
    public void testHunts() {
        Deer d = new Deer("Browse");

        p.hunts(d);
        assertEquals(p.ishunting(), d);
    }
```

- Refactor to extract Pack constructor



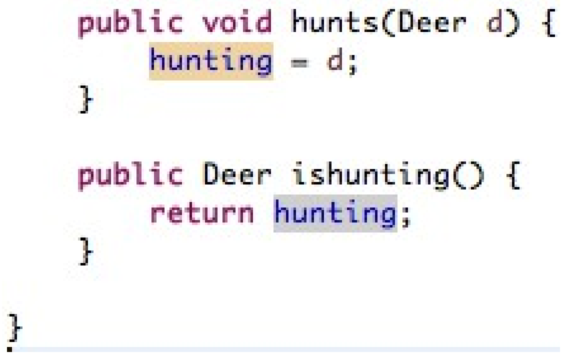
```
import org.junit.Before;
import org.junit.Test;

public class TestPack {

    Pack p;

    @Before
    public void before() {
        p = new Pack();
    }
```

- Add to Pack



```
package animals;

public class Pack {

    private Wolf[] members;
    private Deer hunting;

    public Pack(Wolf wolves[]) {
```

…



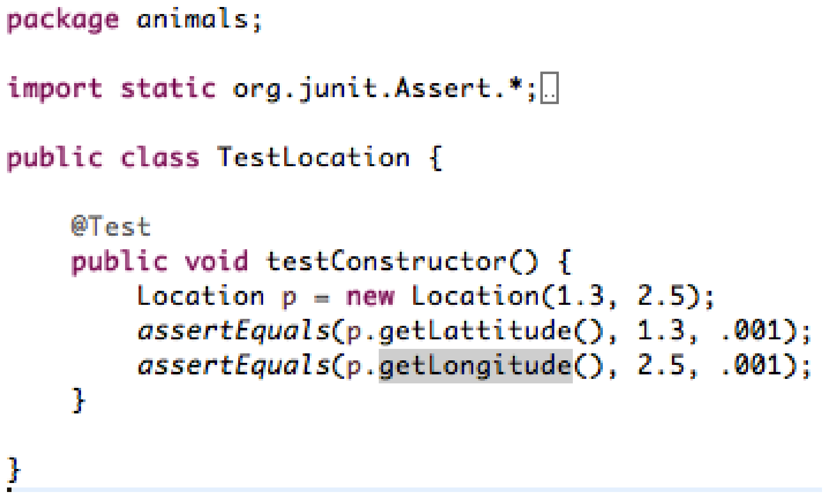
```
    public void hunts(Deer d) {
        hunting = d;
    }

    public Deer ishunting() {
        return hunting;
    }

}
```

Adding Territory and Region

- The intention is that the Territory is the region inhabited by a Pack; a Region is an area under study.
- There can be many territories in a region, and a single territory can be in many regions
  - The relationship between Region and Territory is many to many (*-*)
- Both the Territory and Region are Areas, so we will use a hierarchy to define them
  - The Area is an array of locations defining the boundary of the ares, so we need to define Location also.
- A Location is a longitude and lattitude

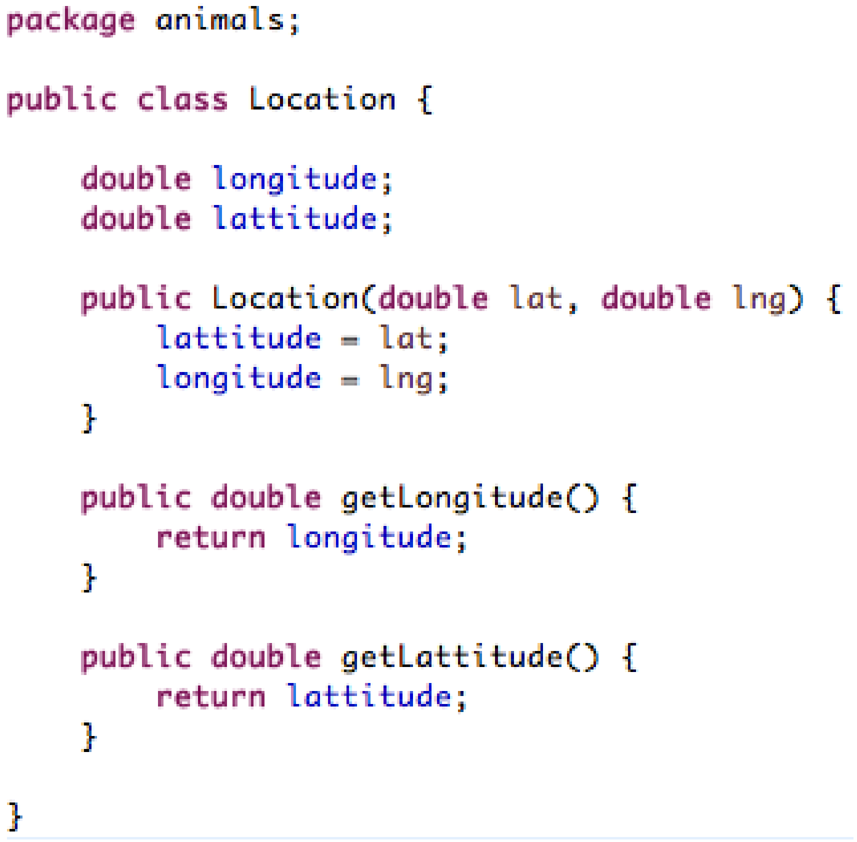
Add Location

- TestLocation



```
package animals;

import static org.junit.Assert.*;

public class TestLocation {

    @Test
    public void testConstructor() {
        Location p = new Location(1.3, 2.5);
        assertEquals(p.getLattitude(), 1.3, .001);
        assertEquals(p.getLongitude(), 2.5, .001);
    }

}
```

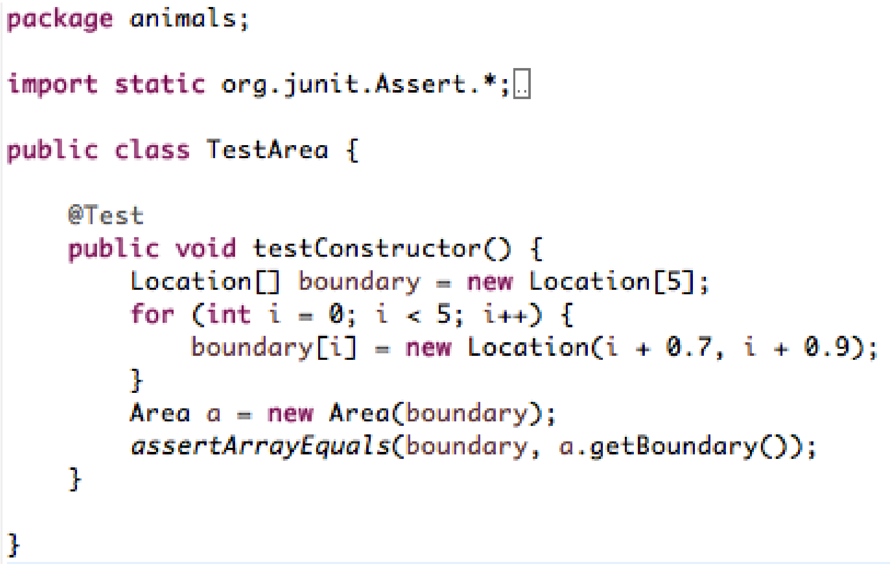
- Location



```
package animals;

public class Location {

    double longitude;
    double lattitude;

    public Location(double lat, double lng) {
        lattitude = lat;
        longitude = lng;
    }

    public double getLongitude() {
        return longitude;
    }

    public double getLattitude() {
        return lattitude;
    }

}
```

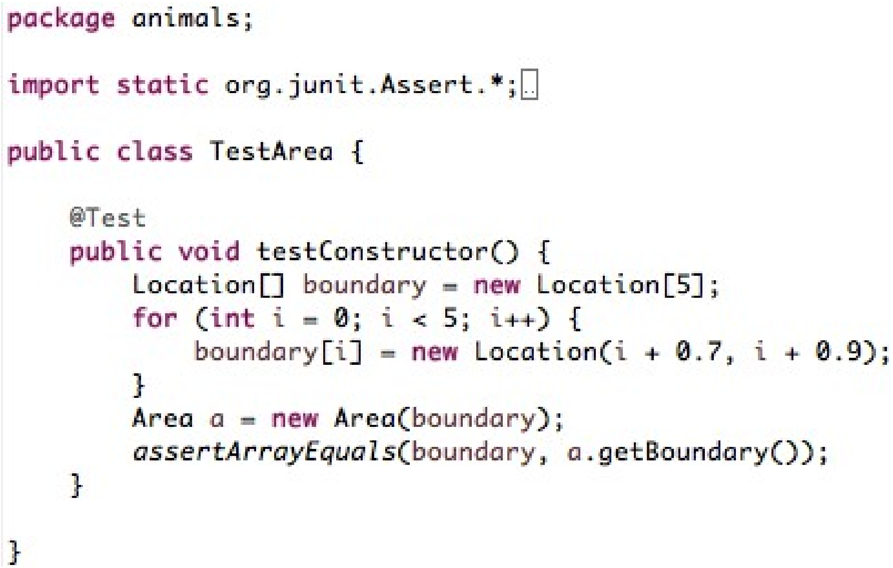
Add Area

- Add TestArea



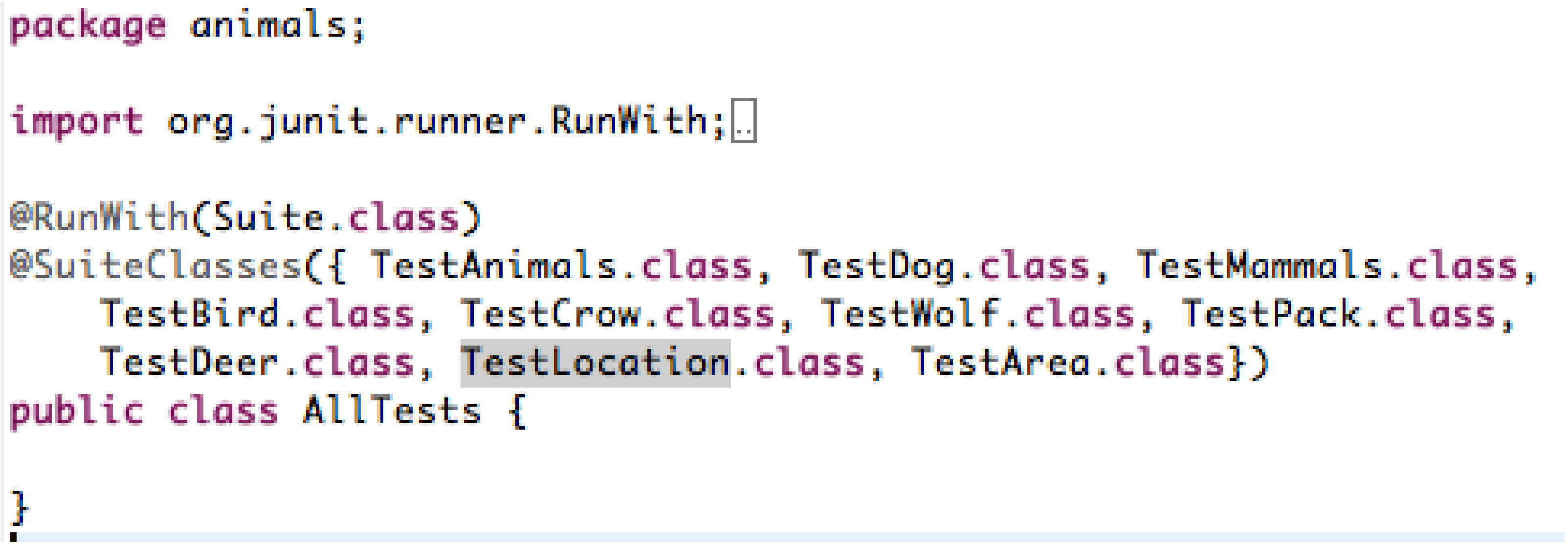
```
package animals;

import static org.junit.Assert.*;

public class TestArea {

    @Test
    public void testConstructor() {
        Location[] boundary = new Location[5];
        for (int i = 0; i < 5; i++) {
            boundary[i] = new Location(i + 0.7, i + 0.9);
        }
        Area a = new Area(boundary);
        assertArrayEquals(boundary, a.getBoundary());
    }

}
```

- Add Area



```
package animals;

import static org.junit.Assert.*;

public class TestArea {

    @Test
    public void testConstructor() {
        Location[] boundary = new Location[5];
        for (int i = 0; i < 5; i++) {
            boundary[i] = new Location(i + 0.7, i + 0.9);
        }
        Area a = new Area(boundary);
        assertArrayEquals(boundary, a.getBoundary());
    }

}
```

Update AllTests



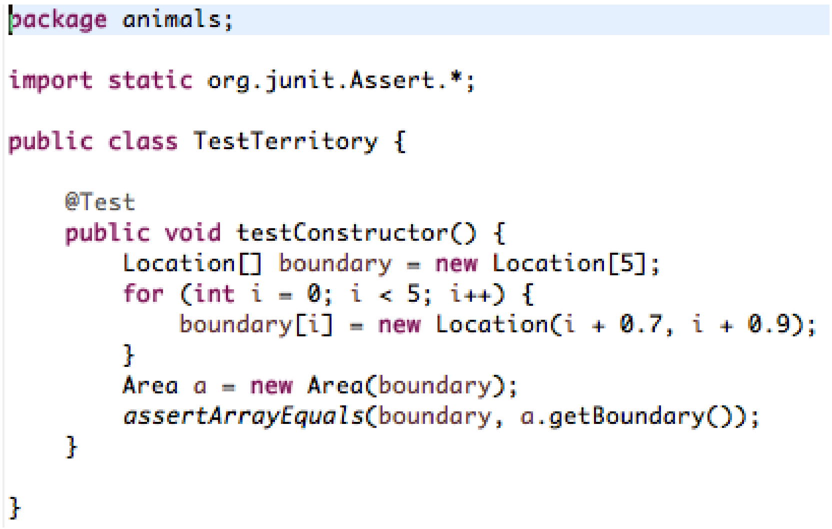
```java
package animals;

import org.junit.runner.RunWith;

@RunWith(Suite.class)
@SuiteClasses({ TestAnimals.class, TestDog.class, TestMammals.class,
    TestBird.class, TestCrow.class, TestWolf.class, TestPack.class,
    TestDeer.class, TestLocation.class, TestArea.class})
public class AllTests {

}
```

Add Territory and Regions

- Now that we have added Location and Regions we can add Territory and Region
- The base classes just extend Area
  - By making them classes, the compiler can check for semantic errors

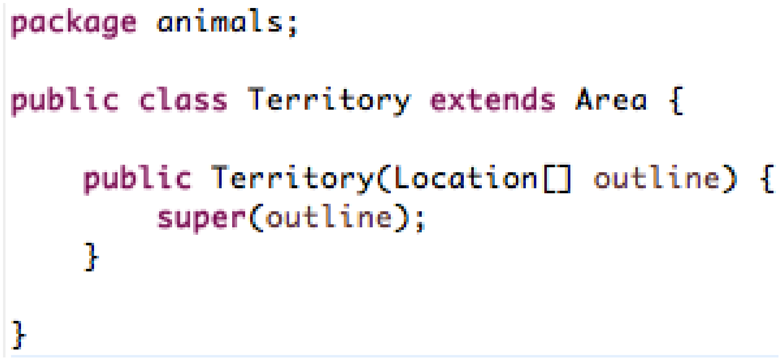
Territory

- TestTerritory



```
package animals;

import static org.junit.Assert.*;

public class TestTerritory {

    @Test
    public void testConstructor() {
        Location[] boundary = new Location[5];
        for (int i = 0; i < 5; i++) {
            boundary[i] = new Location(i + 0.7, i + 0.9);
        }
        Area a = new Area(boundary);
        assertArrayEquals(boundary, a.getBoundary());
    }

}
```

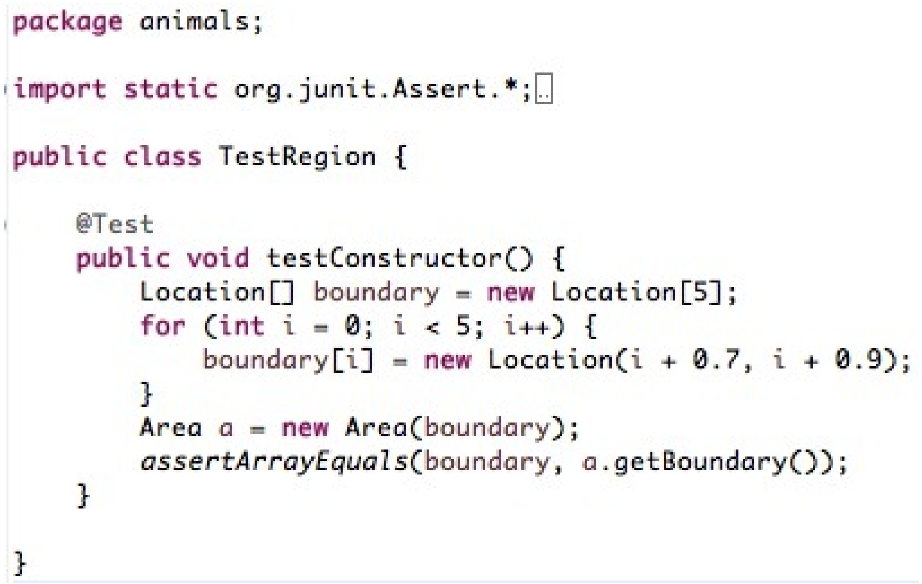
- Territory



```
package animals;

public class Territory extends Area {

    public Territory(Location[] outline) {
        super(outline);
    }

}
```

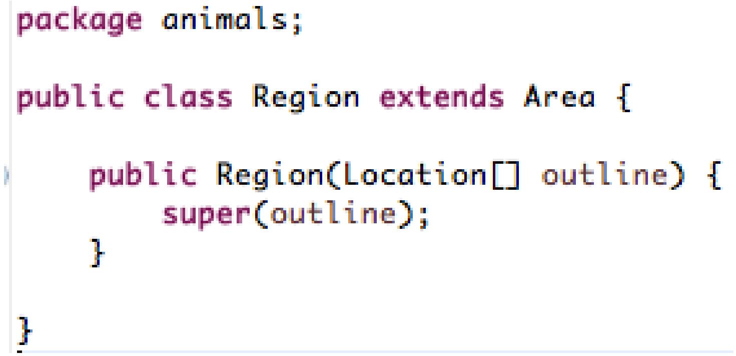
Region

- TestRegion



```
package animals;

import static org.junit.Assert.*;

public class TestRegion {

    @Test
    public void testConstructor() {
        Location[] boundary = new Location[5];
        for (int i = 0; i < 5; i++) {
            boundary[i] = new Location(i + 0.7, i + 0.9);
        }
        Area a = new Area(boundary);
        assertArrayEquals(boundary, a.getBoundary());
    }

}
```

- Region



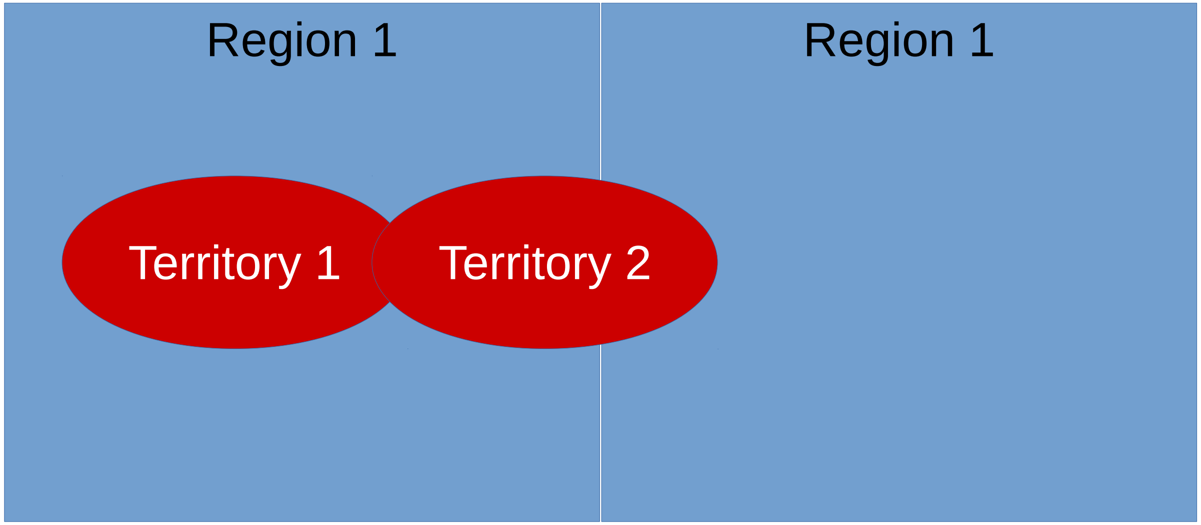
```
package animals;

public class Region extends Area {

    public Region(Location[] outline) {
        super(outline);
    }

}
```

Representing Many to Many Relationships

- We will represent the many to many class using two arrays
  - An array of Territories in Region will represent all of the territories in that region
  - An array of Regions in Territory will represent all of the regions that this particular territory overlaps

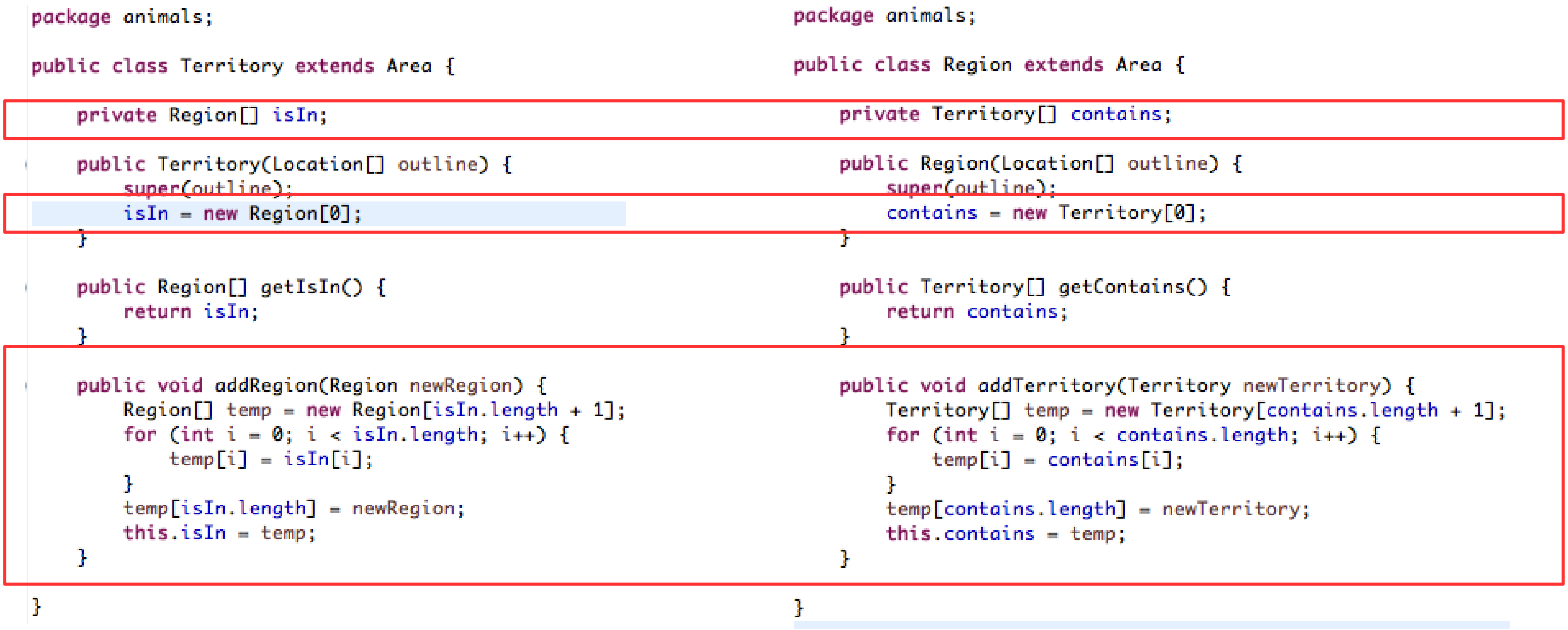
Example of Regions and Territories

- There are two territories and two regions in this example
  - Territory1 is entirely in Region1
  - Territory2 is partially in Region1 and partially in Regions 2
  - We need to represent that Territory2 is in Region1 and Region2
  - We need to represent that Region1 contains Territory1 and Territory2



Many to Many

- Territory
- Region

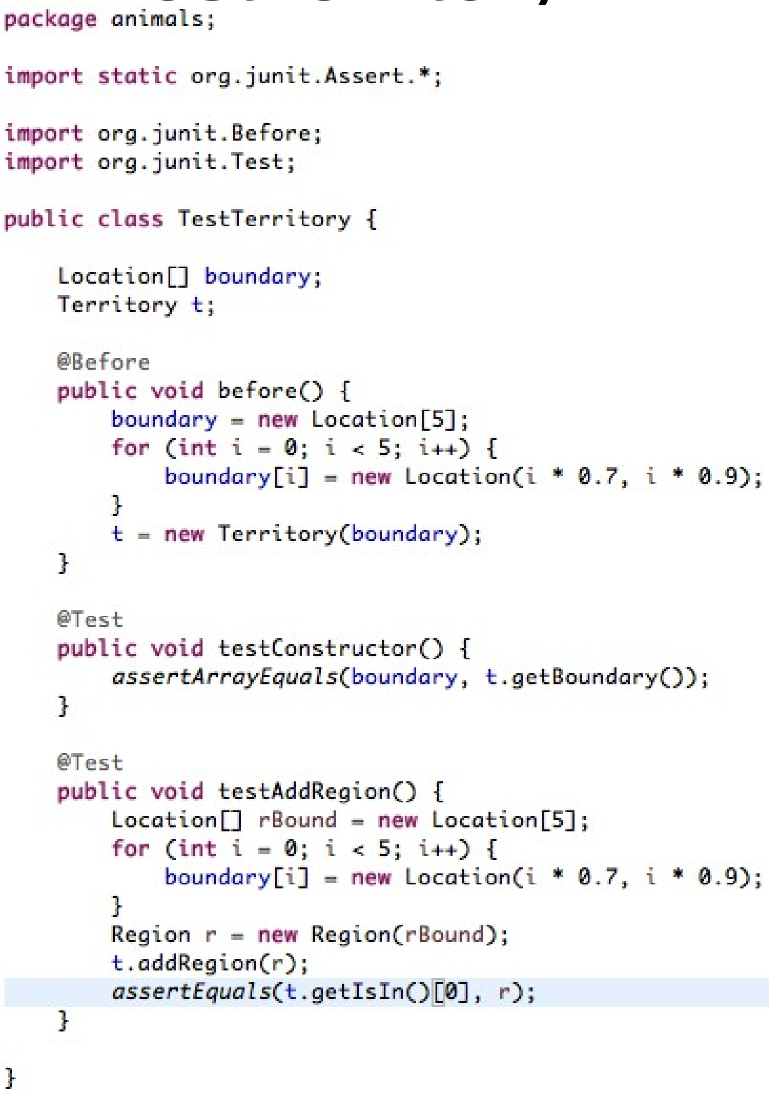


```java
package animals;

public class Territory extends Area {

    private Region[] isIn;

    public Territory(Location[] outline) {
        super(outline);
        isIn = new Region[0];
    }

    public Region[] getIsIn() {
        return isIn;
    }

    public void addRegion(Region newRegion) {
        Region[] temp = new Region[isIn.length + 1];
        for (int i = 0; i < isIn.length; i++) {
            temp[i] = isIn[i];
        }
        temp[isIn.length] = newRegion;
        this.isIn = temp;
    }

}
```

```java
package animals;

public class Region extends Area {

    private Territory[] contains;

    public Region(Location[] outline) {
        super(outline);
        contains = new Territory[0];
    }

    public Territory[] getContains() {
        return contains;
    }

    public void addTerritory(Territory newTerritory) {
        Territory[] temp = new Territory[contains.length + 1];
        for (int i = 0; i < contains.length; i++) {
            temp[i] = contains[i];
        }
        temp[contains.length] = newTerritory;
        this.contains = temp;
    }

}
```

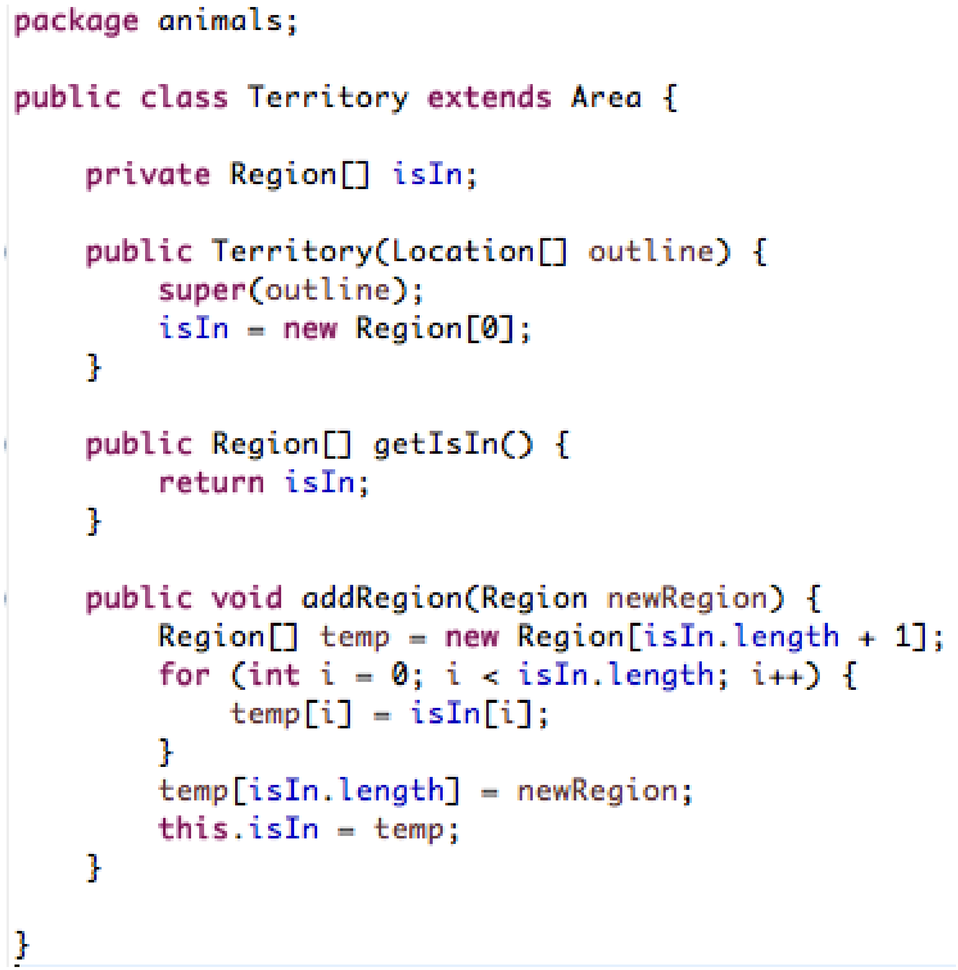
Territory

- TestTerritory



```
package animals;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

public class TestTerritory {

    Location[] boundary;
    Territory t;

    @Before
    public void before() {
        boundary = new Location[5];
        for (int i = 0; i < 5; i++) {
            boundary[i] = new Location(i * 0.7, i * 0.9);
        }
        t = new Territory(boundary);
    }

    @Test
    public void testConstructor() {
        assertArrayEquals(boundary, t.getBoundary());
    }

    @Test
    public void testAddRegion() {
        Location[] rBound = new Location[5];
        for (int i = 0; i < 5; i++) {
            boundary[i] = new Location(i * 0.7, i * 0.9);
        }
        Region r = new Region(rBound);
        t.addRegion(r);
        assertEquals(t.getIsIn()[0], r);
    }

}
```

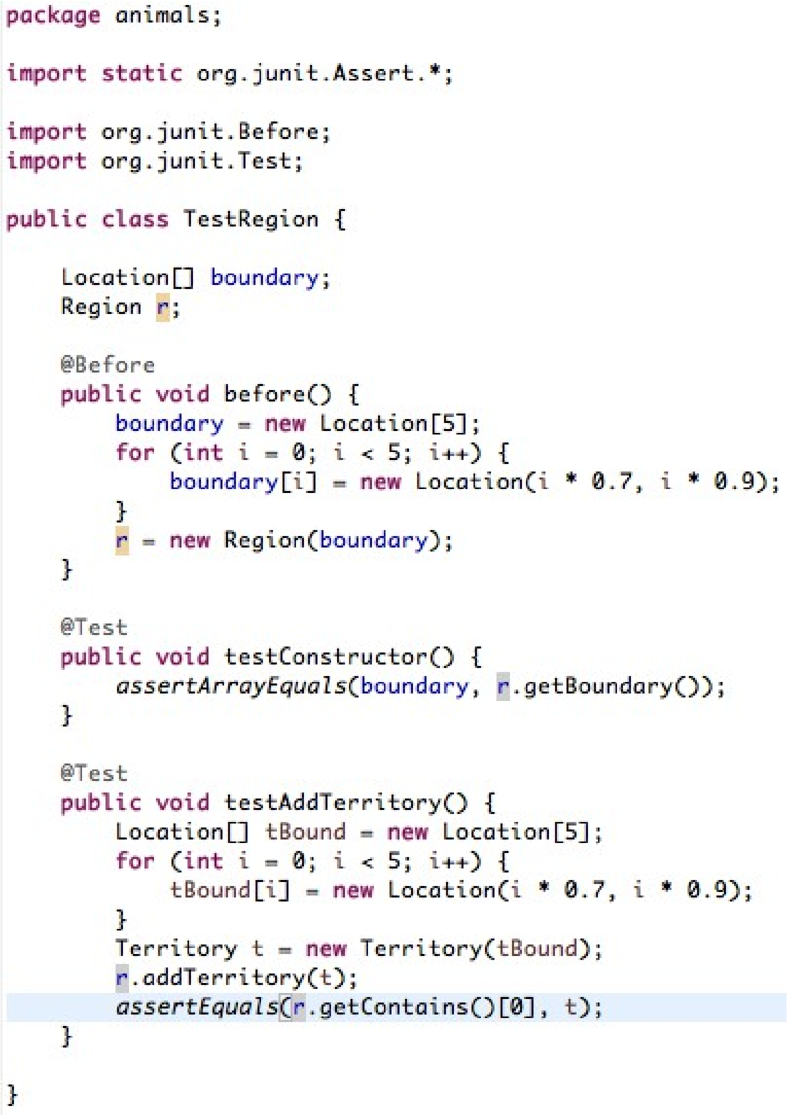
- Territory



```
package animals;

public class Territory extends Area {

    private Region[] isIn;

    public Territory(Location[] outline) {
        super(outline);
        isIn = new Region[0];
    }

    public Region[] getIsIn() {
        return isIn;
    }

    public void addRegion(Region newRegion) {
        Region[] temp = new Region[isIn.length + 1];
        for (int i = 0; i < isIn.length; i++) {
            temp[i] = isIn[i];
        }
        temp[isIn.length] = newRegion;
        this.isIn = temp;
    }

}
```

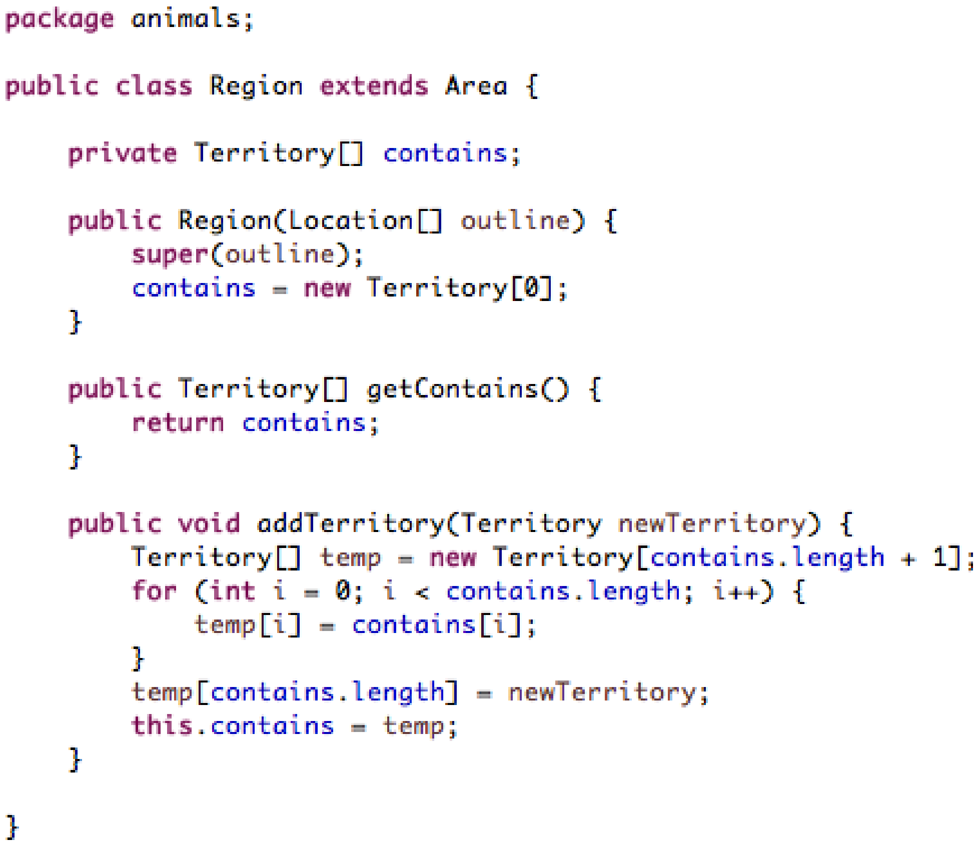
Region

- TestRegion



```
package animals;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

public class TestRegion {

    Location[] boundary;
    Region r;

    @Before
    public void before() {
        boundary = new Location[5];
        for (int i = 0; i < 5; i++) {
            boundary[i] = new Location(i * 0.7, i * 0.9);
        }
        r = new Region(boundary);
    }

    @Test
    public void testConstructor() {
        assertArrayEquals(boundary, r.getBoundary());
    }

    @Test
    public void testAddTerritory() {
        Location[] tBound = new Location[5];
        for (int i = 0; i < 5; i++) {
            tBound[i] = new Location(i * 0.7, i * 0.9);
        }
        Territory t = new Territory(tBound);
        r.addTerritory(t);
        assertEquals(r.getContains()[0], t);
    }

}
```

- Region



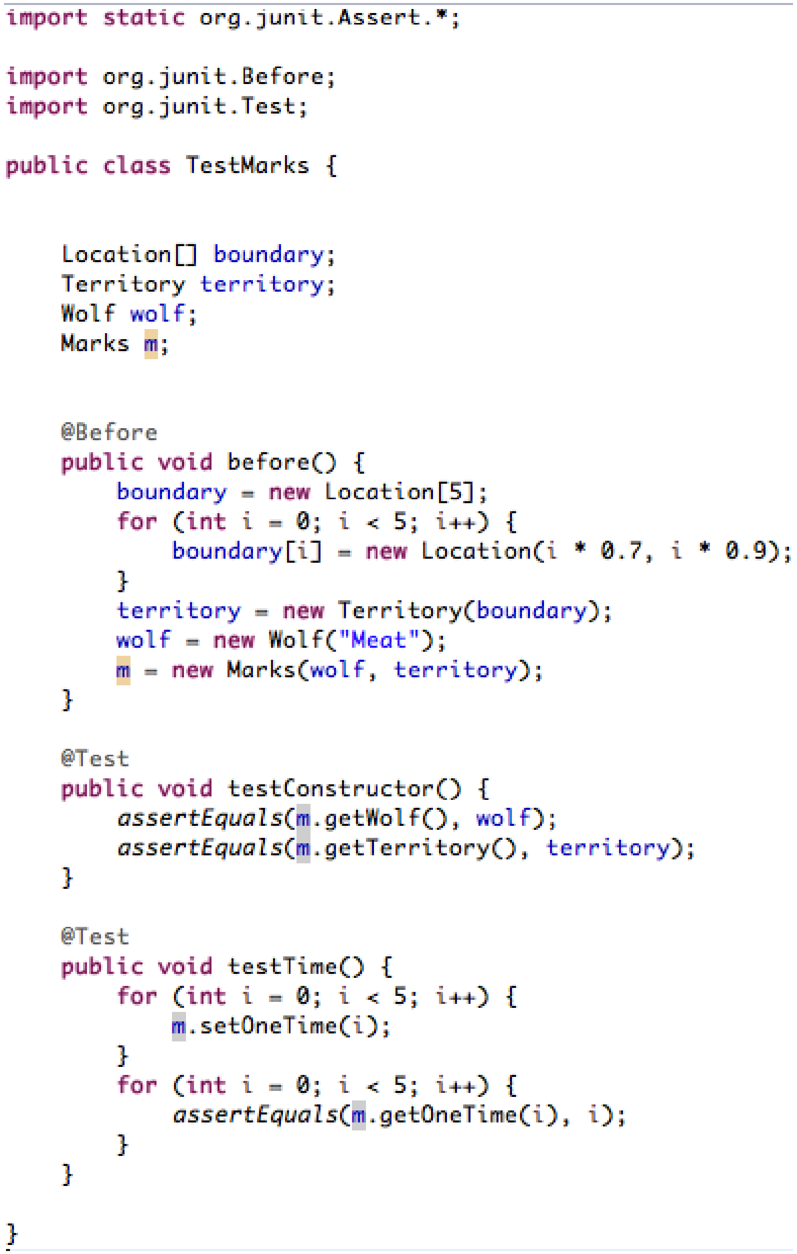
```
package animals;

public class Region extends Area {

    private Territory[] contains;

    public Region(Location[] outline) {
        super(outline);
        contains = new Territory[0];
    }

    public Territory[] getContains() {
        return contains;
    }

    public void addTerritory(Territory newTerritory) {
        Territory[] temp = new Territory[contains.length + 1];
        for (int i = 0; i < contains.length; i++) {
            temp[i] = contains[i];
        }
        temp[contains.length] = newTerritory;
        this.contains = temp;
    }

}
```

Using Classes to Represent Associations

- Sometimes Associations themselves may have properties
  - For example a Wolf marks a Territory at a certain time
  - To capture these times we need them in the association
  - To do this we can create an object called a Mark

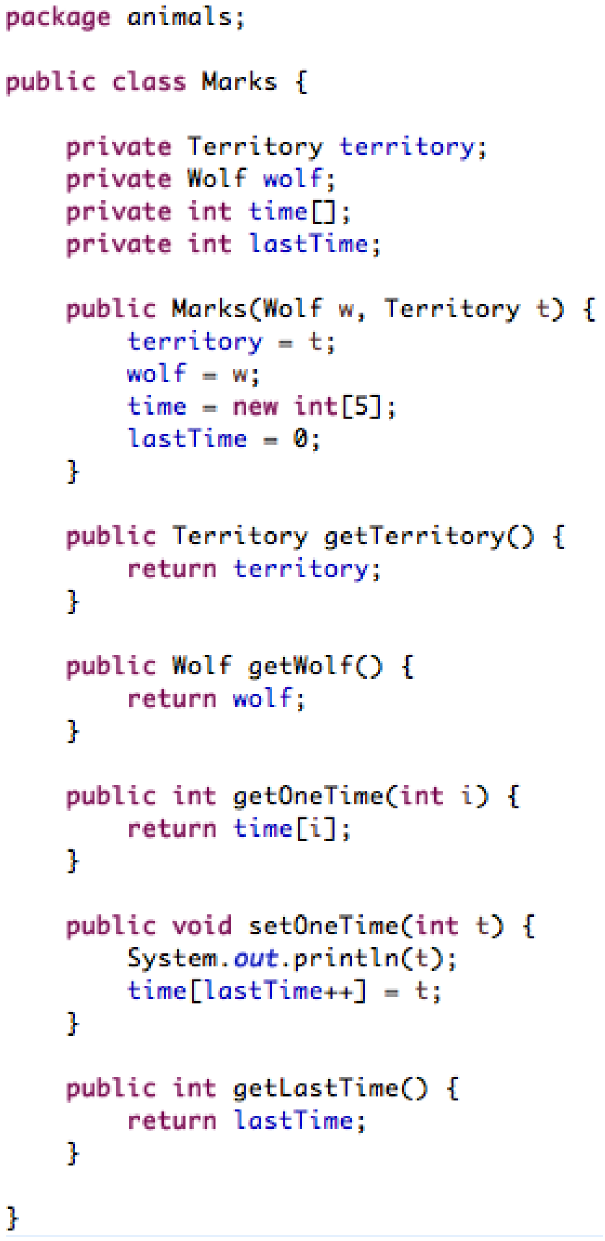
Marks

- TestMarks



```java
import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

public class TestMarks {


    Location[] boundary;
    Territory territory;
    Wolf wolf;
    Marks m;


    @Before
    public void before() {
        boundary = new Location[5];
        for (int i = 0; i < 5; i++) {
            boundary[i] = new Location(i * 0.7, i * 0.9);
        }
        territory = new Territory(boundary);
        wolf = new Wolf("Meat");
        m = new Marks(wolf, territory);
    }

    @Test
    public void testConstructor() {
        assertEquals(m.getWolf(), wolf);
        assertEquals(m.getTerritory(), territory);
    }

    @Test
    public void testTime() {
        for (int i = 0; i < 5; i++) {
            m.setOneTime(i);
        }
        for (int i = 0; i < 5; i++) {
            assertEquals(m.getOneTime(i), i);
        }
    }

}
```

- Marks



```java
package animals;

public class Marks {

    private Territory territory;
    private Wolf wolf;
    private int time[];
    private int lastTime;

    public Marks(Wolf w, Territory t) {
        territory = t;
        wolf = w;
        time = new int[5];
        lastTime = 0;
    }

    public Territory getTerritory() {
        return territory;
    }

    public Wolf getWolf() {
        return wolf;
    }

    public int getOneTime(int i) {
        return time[i];
    }

    public void setOneTime(int t) {
        System.out.println(t);
        time[lastTime++] = t;
    }

    public int getLastTime() {
        return lastTime;
    }

}
```

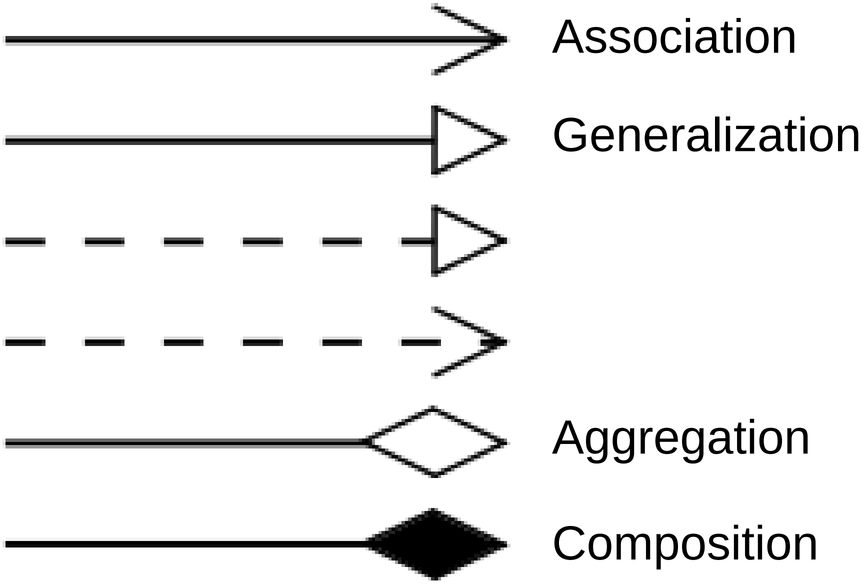
Aggregation and Composition

Aggregation and Composition

- The Pack is an association called and *aggregation*
  - An aggregation is a group of objects that have an existence outside the object
  - If the Pack goes away, the individual wolves do not
- Another type of group association is called a *composition*
  - A composition consist of objects that do go away if the object does.
  - For example, a Wolf may be a composition of body parts
    - A wolf has a leg, but the leg is part of the wolf
- As with all associations, the type of association depends on the model.
  - Theoretically a wolf leg could be transplanted to another wolf, but our model is about hunting and packs, not surgery

UML descriptions of relationships

UML Relationship Symbols



- Generalization
  - Represented by Inheritance in Java
  - Arrow points to superclass
  - Base of arrow on subclass

UML Association

- The arrow on the uni-directional association points to the class to the object of the association
  - E.g. Wolf hunts Mouse points to Mouse
- Bi-directory associations have no arrows
  - E.g. Territory has Regions and Region as Territories
- Cardinality is designated either by an integer or a range of integer
  - The integer represents that a particular number of objects is required for the association
  - The range indicates that any number of objects within the range may be required.

Cardinality Examples

- Integers
  - One Pack holds one Territory
- Ranges
  - Between 2 and any number of Wolves for a Pack
  - A Wolf hunts between 0 and any number of Mice
  - A Pack hunts between 0 and any number of Deer
  - A Territory has between 0 and any number of Regions and vice versa
  - A Wolf marks between 0 and any number of Territories

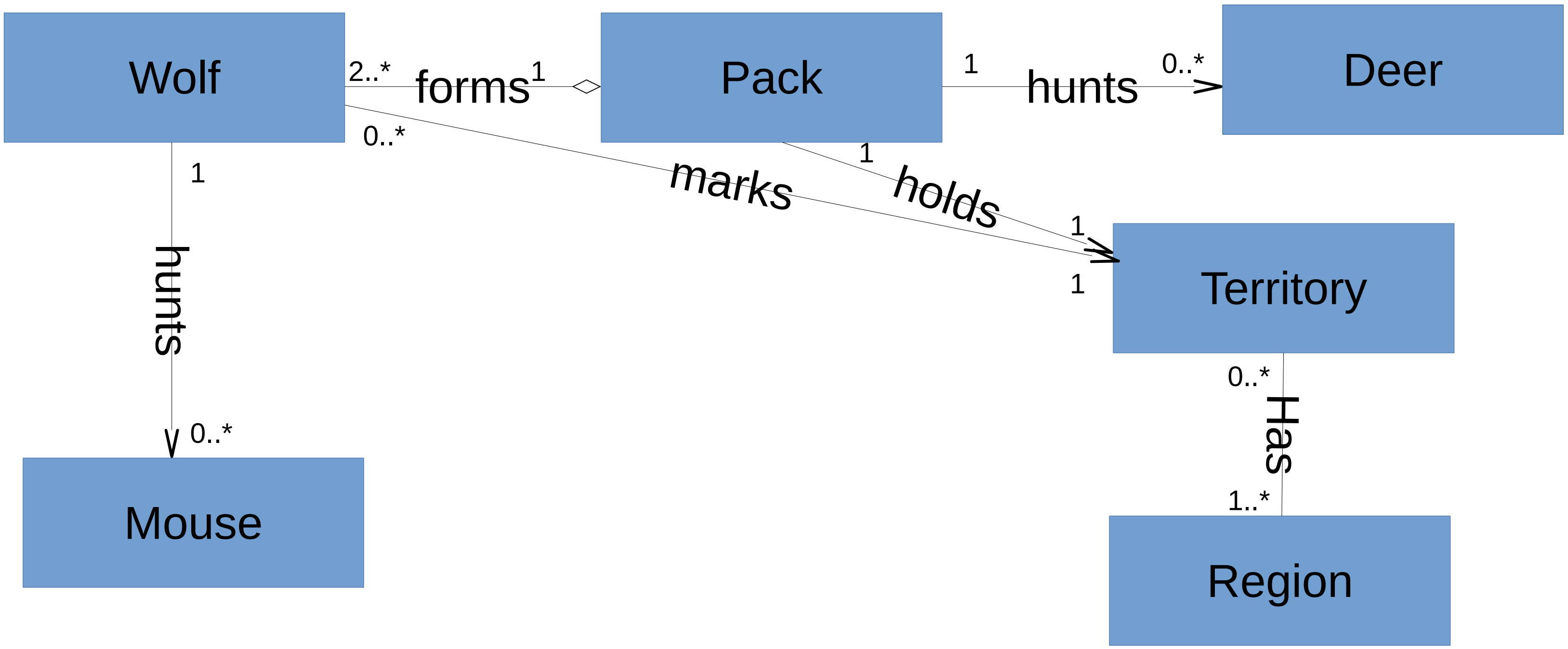
UML Aggregation

- The Pack is an aggregation of Wolves
- The open diamond on the Aggregation points to the aggregation
  - E.g., forms is an aggregation
  - Between 2 and any number of wolves aggregate into a Pack
  - We do not call a single wolf a pack

Aggregation vs Composition

- Aggregation is weaker than composition
- Objects in a composition have no existence outside the composition
- For example
  - A Car is a composition of body, wheels, engine …
  - A Pond may hold an aggregation of ducks.

Wolf Model


Wolf
2..*
forms
1
Pack
1
hunts
0..*
Deer
0..*
marks
1
holds
1
1
Territory
1
hunts
0..*
Mouse
0..*
Has
1..*
Region