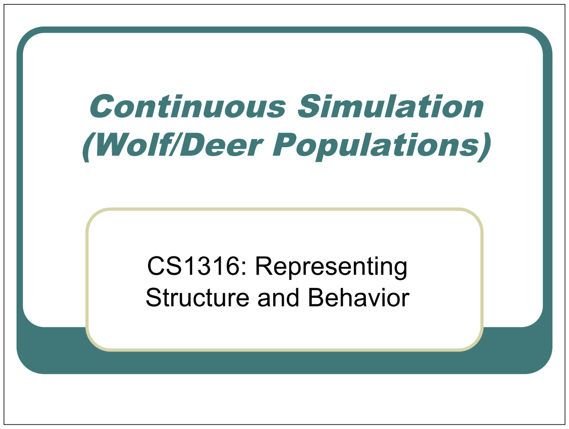
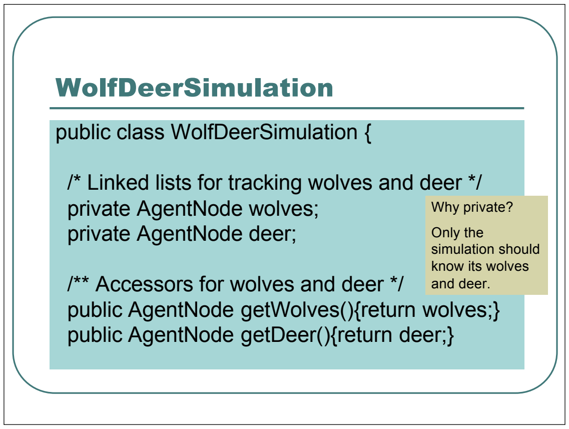


Continuous Simulation (Wolf/Deer Populations)

CS1316: Representing Structure and Behavior

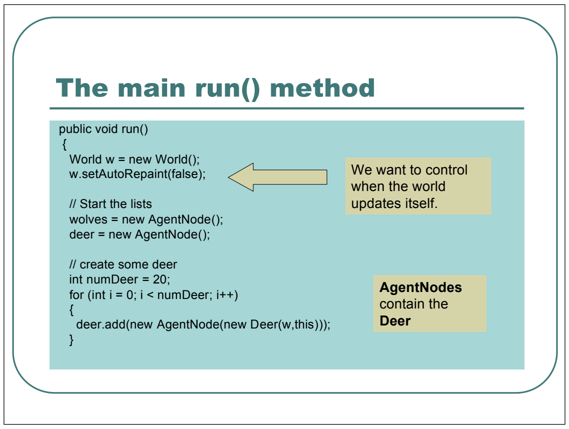


WolfDeerSimulation

```
public class WolfDeerSimulation {

  /* Linked lists for tracking wolves and deer */
  private AgentNode wolves;
  private AgentNode deer;

  /** Accessors for wolves and deer */
  public AgentNode getWolves(){return wolves;}
  public AgentNode getDeer(){return deer;}
```

Why private?

Only the simulation should know its wolves and deer.

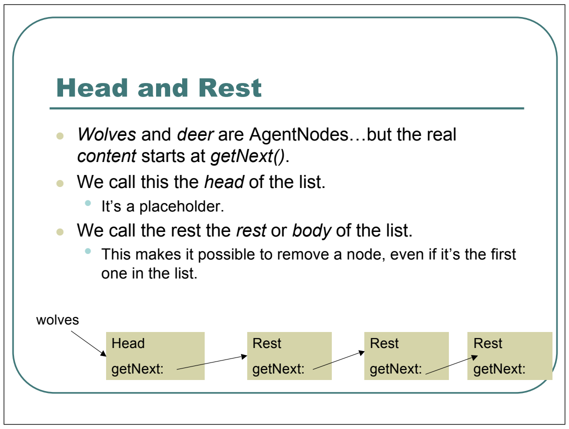


The main run() method

```
public void run()
 {
   World w = new World();
   w.setAutoRepaint(false);

   // Start the lists
   wolves = new AgentNode();
   deer = new AgentNode();

   // create some deer
   int numDeer = 20;
   for (int i = 0; i < numDeer; i++)
   {
     deer.add(new AgentNode(new Deer(w,this)));
   }
```

We want to control when the world updates itself.

AgentNodes contain the **Deer**



Head and Rest

- *Wolves* and *deer* are AgentNodes…but the real *content* starts at *getNext()*.
- We call this the *head* of the list.
  - It's a placeholder.
- We call the rest the *rest* or *body* of the list.
  - This makes it possible to remove a node, even if it's the first one in the list.

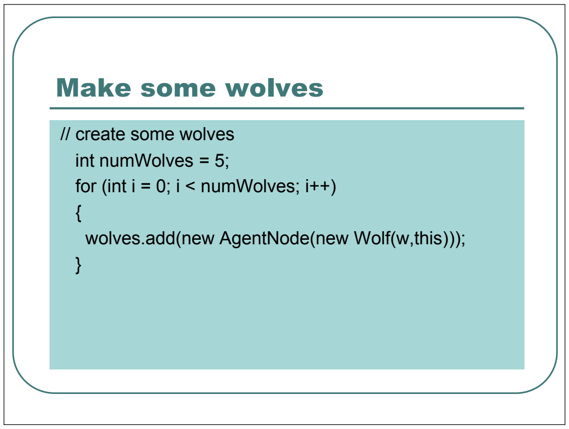
wolves

Head
getNext:

Rest
getNext:

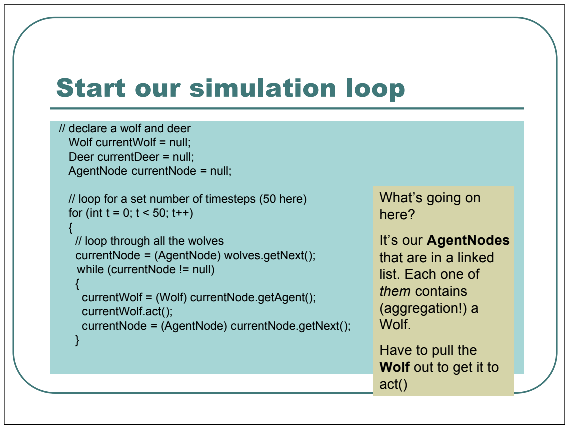
Rest
getNext:

Rest
getNext:



Make some wolves

```
// create some wolves
   int numWolves = 5;
   for (int i = 0; i < numWolves; i++)
   {
     wolves.add(new AgentNode(new Wolf(w,this)));
   }
```



Start our simulation loop

```
// declare a wolf and deer
   Wolf currentWolf = null;
   Deer currentDeer = null;
   AgentNode currentNode = null;

   // loop for a set number of timesteps (50 here)
   for (int t = 0; t < 50; t++)
   {
     // loop through all the wolves
     currentNode = (AgentNode) wolves.getNext();
      while (currentNode != null)
     {
       currentWolf = (Wolf) currentNode.getAgent();
       currentWolf.act();
       currentNode = (AgentNode) currentNode.getNext();
     }
```

What's going on here?

It's our **AgentNodes** that are in a linked list. Each one of *them* contains (aggregation!) a Wolf.

Have to pull the **Wolf** out to get it to act()

Give the deer a chance to act

```
// loop through all the deer
     currentNode = (AgentNode) deer.getNext();
     while (currentNode != null)
     {
       currentDeer = (Deer) currentNode.getAgent();
       currentDeer.act();
       currentNode = (AgentNode) currentNode.getNext();
     }
```

Same unpackaging going on here.

Show us what happened

```
      // repaint the world to show the movement
      w.repaint();

      // Let's figure out where we stand...
      System.out.println(">>> Timestep: "+t);
      System.out.println("Wolves left: "+wolves.getNext().count());
      System.out.println("Deer left: "+deer.getNext().count());

      // Wait for one second
      //Thread.sleep(1000);
    }
  }
```

Does the simulation go too fast? Make the *thread* of execution *sleep* for 1000 milliseconds

Implementing a Wolf

```
import java.awt.Color;
import java.util.Random;
import java.util.Iterator;

/**
 * Class that represents a wolf.   The wolf class
 * tracks all the living wolves with a linked list.
 *
 * @author Barb Ericson ericson@cc.gatech.edu
 */
public class Wolf extends Turtle
{
  /////////////// fields //////////////////////

  /** class constant for the color */
  private static final Color grey = new Color(153,153,153);

    /** class constant for probability of NOT turning */
    protected static final double PROB_OF_STAY = 1.0/10;
```

A *final* is something that won't change: A constant. It's used to make code more *readable* yet *easy-to-change.*

Private vs. Protected? Use Protected if your *subclasses* will need to access (new kinds of wolves?)

Constants are typically all-caps

More Wolf fields

```
/** class constant for top speed (max num steps can
    move in a timestep) */
 protected static final int maxSpeed = 60;

 /** My simulation */
 protected WolfDeerSimulation mySim;

 /** random number generator */
 protected static Random randNumGen = new Random();
```

maxSpeed should *probably* be all-caps (or did you want to make it variable? Do wolves get slower as they get hungry?

There is more than one kind of random. Treating it as an *object* makes it easier to have different kinds later.

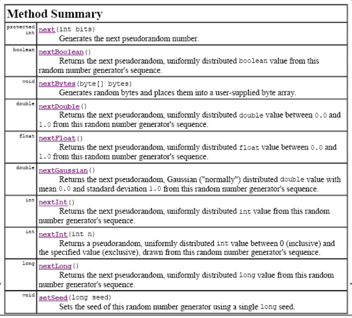
Constructors

```
///////////////////////////// Constructors //////////////////////

 /**
  * Constructor that takes the model display (the original
  * position will be randomly assigned)
  * @param modelDisplayer thing that displays the model
  * @param mySim my simulation
  */
 public Wolf (ModelDisplay modelDisplayer,WolfDeerSimulation
     thisSim)
 {
   super(randNumGen.nextInt(modelDisplayer.getWidth()),
         randNumGen.nextInt(modelDisplayer.getHeight()),
         modelDisplayer);
   init(thisSim);
 }

 /** Constructor that takes the x and y and a model
  * display to draw it on
  * @param x the starting x position
  * @param y the starting y position
  * @param modelDisplayer the thing that displays the model
  * @param mySim my simulation
  */
 public Wolf (int x, int y, ModelDisplay modelDisplayer,
             WolfDeerSimulation thisSim)
 {
   // let the parent constructor handle it
   super(x,y,modelDisplayer);
   init(thisSim);
 }
```

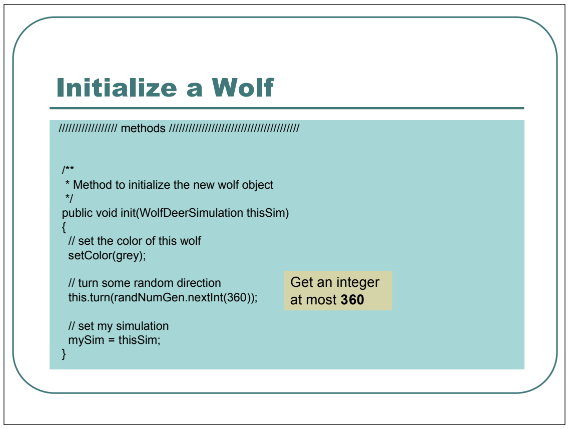
Remember that a constructor must match its superclass, if you want to use super(). These are like the ones in Turtle.

What's a ***ModelDisplay***? The abstract superclass of the **World**.

Using a Random: PseudoRandom Number Generator



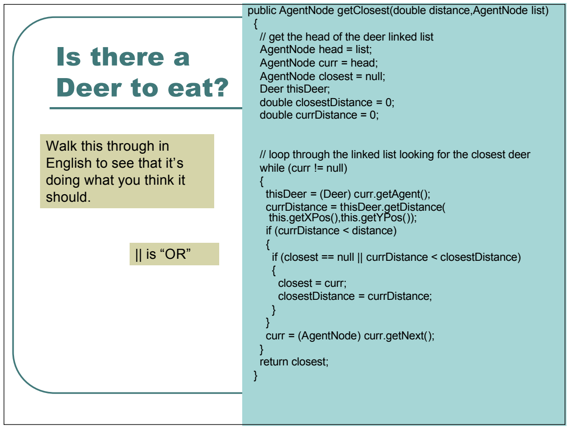
Method Summary	
protected int	next(int bits) Generates the next pseudorandom number.
boolean	nextBoolean() Returns the next pseudorandom, uniformly distributed boolean value from this random number generator's sequence.
void	nextBytes(byte[] bytes) Generates random bytes and places them into a user-supplied byte array.
double	nextDouble() Returns the next pseudorandom, uniformly distributed double value between 0.0 and 1.0 from this random number generator's sequence.
float	nextFloat() Returns the next pseudorandom, uniformly distributed float value between 0.0 and 1.0 from this random number generator's sequence.
double	nextGaussian() Returns the next pseudorandom, Gaussian ("normally") distributed double value with mean 0.0 and standard deviation 1.0 from this random number generator's sequence.
int	nextInt() Returns the next pseudorandom, uniformly distributed int value from this random number generator's sequence.
int	nextInt(int n) Returns a pseudorandom, uniformly distributed int value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence.
long	nextLong() Returns the next pseudorandom, uniformly distributed long value from this random number generator's sequence.
void	setSeed(long seed) Sets the seed of this random number generator using a single long seed.



Initialize a Wolf

```
////////////////// methods ////////////////////////////////////////


 /**
  * Method to initialize the new wolf object
  */
 public void init(WolfDeerSimulation thisSim)
 {
   // set the color of this wolf
   setColor(grey);

   // turn some random direction
   this.turn(randNumGen.nextInt(360));

   // set my simulation
   mySim = thisSim;
 }
```

Get an integer at most **360**

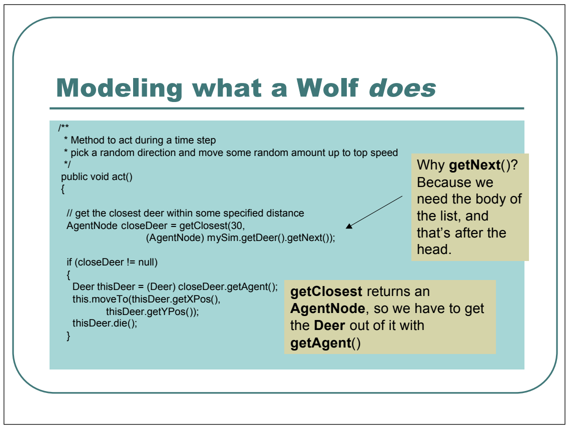


Is there a Deer to eat?

Walk this through in English to see that it's doing what you think it should.

|| is "OR"

```
public AgentNode getClosest(double distance,AgentNode list)
 {
   // get the head of the deer linked list
   AgentNode head = list;
   AgentNode curr = head;
   AgentNode closest = null;
   Deer thisDeer;
   double closestDistance = 0;
   double currDistance = 0;


   // loop through the linked list looking for the closest deer
   while (curr != null)
   {
     thisDeer = (Deer) curr.getAgent();
     currDistance = thisDeer.getDistance(
      this.getXPos(),this.getYPos());
     if (currDistance < distance)
     {
       if (closest == null || currDistance < closestDistance)
       {
         closest = curr;
         closestDistance = currDistance;
       }
     }
     curr = (AgentNode) curr.getNext();
   }
   return closest;
 }
```

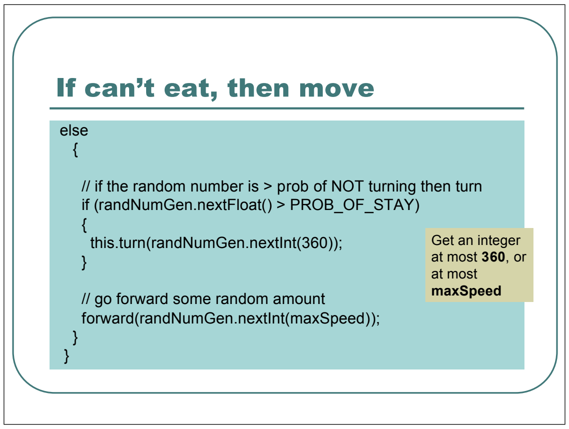


Modeling what a Wolf *does*

```
/**
  * Method to act during a time step
  * pick a random direction and move some random amount up to top speed
  */
 public void act()
 {

   // get the closest deer within some specified distance
   AgentNode closeDeer = getClosest(30,
                         (AgentNode) mySim.getDeer().getNext());

   if (closeDeer != null)
   {
     Deer thisDeer = (Deer) closeDeer.getAgent();
     this.moveTo(thisDeer.getXPos(),
                 thisDeer.getYPos());
     thisDeer.die();
   }
```

Why **getNext**()? Because we need the body of the list, and that's after the head.

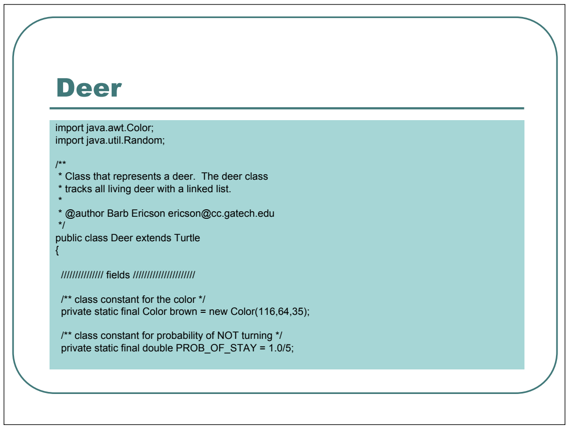
getClosest returns an **AgentNode**, so we have to get the **Deer** out of it with **getAgent**()



If can't eat, then move

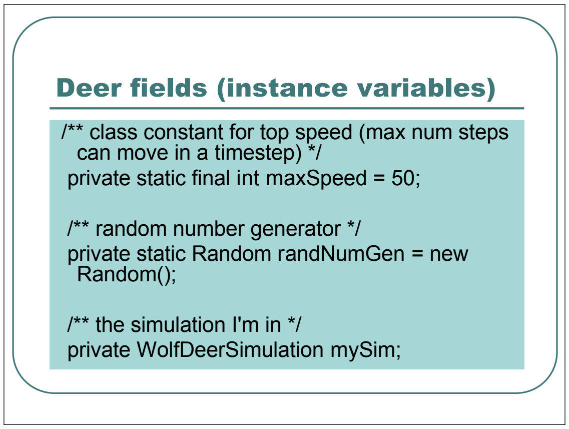
```
else
   {

    // if the random number is > prob of NOT turning then turn
    if (randNumGen.nextFloat() > PROB_OF_STAY)
    {
      this.turn(randNumGen.nextInt(360));
    }

    // go forward some random amount
    forward(randNumGen.nextInt(maxSpeed));
  }
 }
```

Get an integer at most **360**, or at most **maxSpeed**



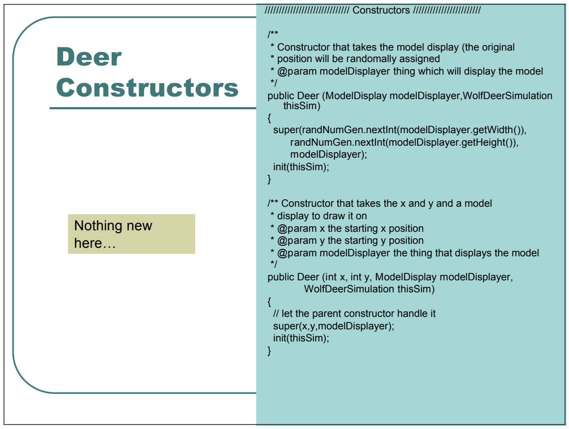
Deer

```
import java.awt.Color;
import java.util.Random;

/**
 * Class that represents a deer.  The deer class
 * tracks all living deer with a linked list.
 *
 * @author Barb Ericson ericson@cc.gatech.edu
 */
public class Deer extends Turtle
{

  /////////////// fields //////////////////////

  /** class constant for the color */
  private static final Color brown = new Color(116,64,35);

  /** class constant for probability of NOT turning */
  private static final double PROB_OF_STAY = 1.0/5;
```



Deer fields (instance variables)

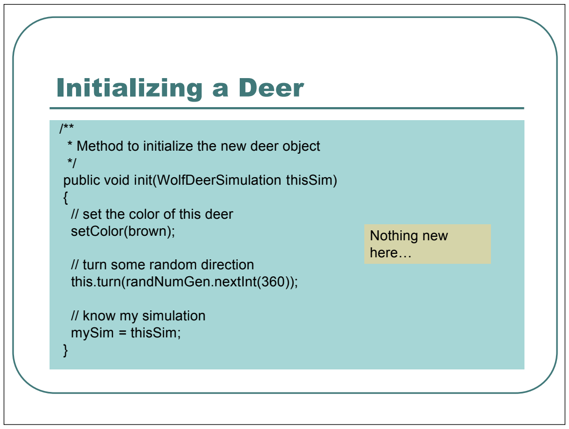
```
/** class constant for top speed (max num steps
   can move in a timestep) */
 private static final int maxSpeed = 50;

 /** random number generator */
 private static Random randNumGen = new
   Random();

 /** the simulation I'm in */
 private WolfDeerSimulation mySim;
```



Deer Constructors

```
//////////////////////////////// Constructors ///////////////////////

/**
 * Constructor that takes the model display (the original
 * position will be randomally assigned
 * @param modelDisplayer thing which will display the model
 */
public Deer (ModelDisplay modelDisplayer,WolfDeerSimulation
     thisSim)
{
  super(randNumGen.nextInt(modelDisplayer.getWidth()),
        randNumGen.nextInt(modelDisplayer.getHeight()),
        modelDisplayer);
  init(thisSim);
}

/** Constructor that takes the x and y and a model
 * display to draw it on
 * @param x the starting x position
 * @param y the starting y position
 * @param modelDisplayer the thing that displays the model
 */
public Deer (int x, int y, ModelDisplay modelDisplayer,
             WolfDeerSimulation thisSim)
{
  // let the parent constructor handle it
  super(x,y,modelDisplayer);
  init(thisSim);
}
```

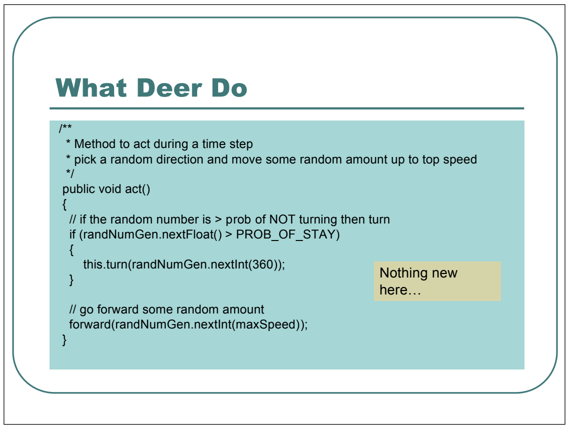
Nothing new here…



Initializing a Deer

```
/**
 * Method to initialize the new deer object
 */
public void init(WolfDeerSimulation thisSim)
{
  // set the color of this deer
  setColor(brown);

  // turn some random direction
  this.turn(randNumGen.nextInt(360));

  // know my simulation
  mySim = thisSim;
}
```

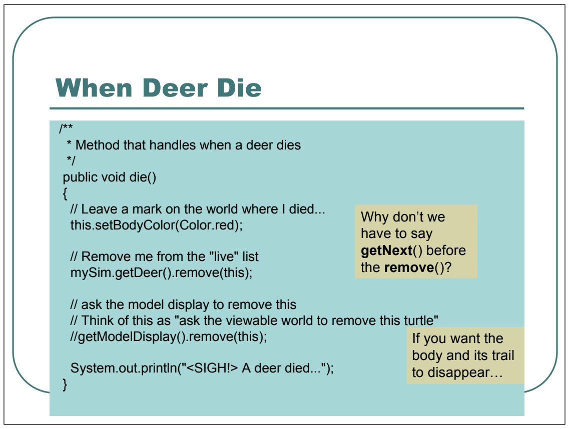
Nothing new here…



What Deer Do

```
/**
 * Method to act during a time step
 * pick a random direction and move some random amount up to top speed
 */
public void act()
{
  // if the random number is > prob of NOT turning then turn
  if (randNumGen.nextFloat() > PROB_OF_STAY)
  {
     this.turn(randNumGen.nextInt(360));
  }

  // go forward some random amount
  forward(randNumGen.nextInt(maxSpeed));
}
```

Nothing new here…

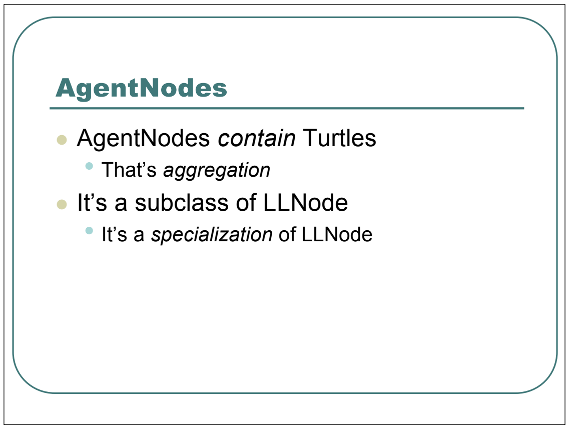


When Deer Die

```
/**
 * Method that handles when a deer dies
 */
public void die()
{
  // Leave a mark on the world where I died...
  this.setBodyColor(Color.red);

  // Remove me from the "live" list
  mySim.getDeer().remove(this);

  // ask the model display to remove this
  // Think of this as "ask the viewable world to remove this turtle"
  //getModelDisplay().remove(this);

  System.out.println("<SIGH!> A deer died...");
}
```

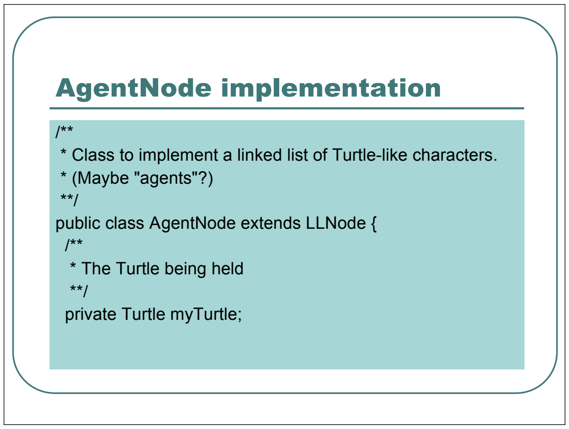
Why don't we have to say **getNext**() before the **remove**()?

If you want the body and its trail to disappear…



AgentNodes

- AgentNodes *contain* Turtles
  - That's *aggregation*
- It's a subclass of LLNode
  - It's a *specialization* of LLNode



AgentNode implementation

```
/**
 * Class to implement a linked list of Turtle-like characters.
 * (Maybe "agents"?)
 **/
public class AgentNode extends LLNode {
 /**
  * The Turtle being held
  **/
 private Turtle myTurtle;
```

AgentNode constructors

```
/** Two constructors: One for creating the head of the list
  * , with no agent
 **/
 public AgentNode() {super();}

 /**
  * One constructor for creating a node with an agent
  **/
 public AgentNode(Turtle agent){
   super();
   this.setAgent(agent);
 }
```

AgentNode getter/setter

```
/**
  * Setter for the turtle
  **/
 public void setAgent(Turtle agent){
   myTurtle = agent;
 }

 /**
  * Getter for the turtle
  **/
 public Turtle getAgent(){return myTurtle;}
```

AgentNode: Remove node where Turtle is found

```
/**
  * Remove the node where this turtle is found.
  **/
 public void remove(Turtle myTurtle) {
   // Assume we're calling on the head
   AgentNode head = this;
   AgentNode current = (AgentNode) this.getNext();

   while (current != null) {
     if (current.getAgent() == myTurtle)
     {// If found the turtle, remove that node
       head.remove(current);
     }

     current = (AgentNode) current.getNext();
   }
 }
```

It's just like other linked list removes, but now we're looking for the node that *contains* the input turtle.

Think about it…

- What if AgentNodes contained ***Object***s?
  - Object is a class that is the superclass of *all* classes (even if not explicitly *extended*).
  - AgentNodes that contain Objects could be *general* linked lists that contain *anything*
    - Just cast things as you need them as you pull them out.

Back to the simulation: What might we change?

- Wolves that aren't *always* hungry?
- Having wolves that *chase* deer? Have deer *run from* wolves?
- And how do we look at the results?

We'll deal with hunger first, then with comparing, then with running towards/away.

Creating a Hungry Wolf

```
/**
  * A class that extends the Wolf to have a Hunger level.
  * Wolves only eat when they're hungry
  **/
public class HungryWolf extends Wolf {
  /**
   * Number of cycles before I'll eat again
   **/
  private int satisfied;

  /** class constant for number of turns before hungry */
  private static final int MAX_SATISFIED = 3;
```

Need to match

```
/**
  * Constructor that takes the model display (the original
  * position will be randomly assigned)
  * @param modelDisplayer thing that displays the model
  * @param mySim my simulation
  */
 public HungryWolf (ModelDisplay
     modelDisplayer,WolfDeerSimulation thisSim)
 {
   super(modelDisplayer,thisSim);
 }

 /** Constructor that takes the x and y and a model
  * display to draw it on
  * @param x the starting x position
  * @param y the starting y position
  * @param modelDisplayer the thing that displays the model
  * @param mySim my simulation
  */
 public HungryWolf (int x, int y, ModelDisplay
     modelDisplayer,
             WolfDeerSimulation thisSim)
 {
   // let the parent constructor handle it
   super(x,y,modelDisplayer,thisSim);
 }
```

Initializing a HungryWolf

```
/**
  * Method to initialize the hungry wolf object
  */
 public void init(WolfDeerSimulation thisSim)
 {
   super.init(thisSim);

   satisfied = MAX_SATISFIED;
 }
```

What a HungryWolf does

```
/**
  * Method to act during a time step
  * pick a random direction and move some random amount up to top speed
  */
 public void act()
 {
  // Decrease satisfied time, until hungry again
  satisfied--;

  // get the closest deer within some specified distance
  AgentNode closeDeer = getClosest(30,
                         (AgentNode) mySim.getDeer().getNext());

  if (closeDeer != null)
  { // Even if deer close, only eat it if you're hungry.
    if (satisfied <= 0)
    {Deer thisDeer = (Deer) closeDeer.getAgent();
      this.moveTo(thisDeer.getXPos(),
               thisDeer.getYPos());
      thisDeer.die();
      satisfied = MAX_SATISFIED;

    }}
```

If there is a Deer near, then check if you're hungry, and only then—eat and get “full”

And if no Deer are near…

```
else
   {

     // if the randome number is > prob of turning then turn
     if (randNumGen.nextFloat() > PROB_OF_TURN)
     {
       this.turn(randNumGen.nextInt(360));
     }

     // go forward some random amount
     forward(randNumGen.nextInt(maxSpeed));

   }
 }
```

Nothing new here…

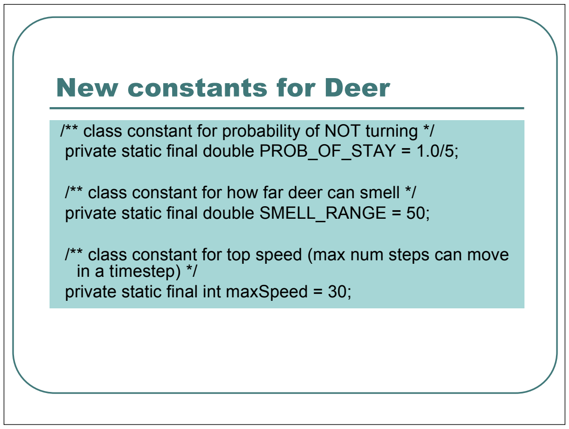
Changing the Simulation to make HungryWolves (in run())

```
// create some wolves
   int numWolves = 5;
   for (int i = 0; i < numWolves; i++)
   {
     wolves.add(new AgentNode(new
  HungryWolf(w,this)));
   }
```

Everything else just works, because **HungryWolf** is a kind of **Wolf**

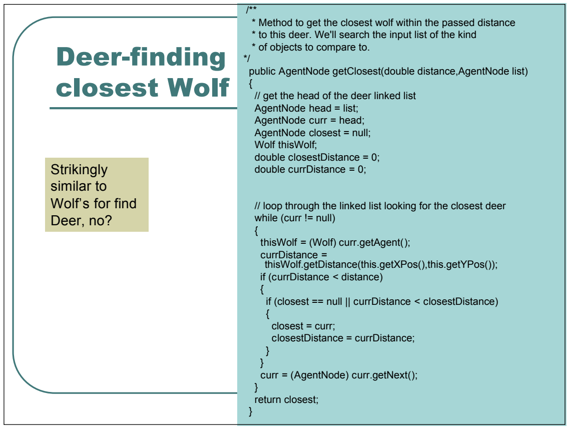
Making Wolves and Deer Run

- What we do:
  - In Deer, if there is a Wolf within our *smelling range*, run in the opposite direction (turn towards, turn 180, move)
  - In Wolf, if there is a Deer within our *smelling range*, run towards it.
  - (Stays the same) If the Wolf gets close enough, gobble up the Deer.
  - (Stays the same) For both, otherwise, wander aimlessly.



New constants for Deer

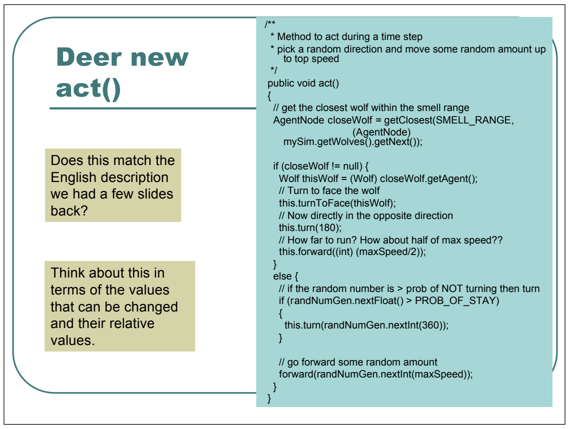
```
/** class constant for probability of NOT turning */
 private static final double PROB_OF_STAY = 1.0/5;

 /** class constant for how far deer can smell */
 private static final double SMELL_RANGE = 50;

 /** class constant for top speed (max num steps can move
    in a timestep) */
 private static final int maxSpeed = 30;
```



Deer-finding closest Wolf

Strikingly similar to Wolf's for find Deer, no?

```
/**
  * Method to get the closest wolf within the passed distance
  * to this deer. We'll search the input list of the kind
  * of objects to compare to.
*/
 public AgentNode getClosest(double distance,AgentNode list)
 {
   // get the head of the deer linked list
   AgentNode head = list;
   AgentNode curr = head;
   AgentNode closest = null;
   Wolf thisWolf;
   double closestDistance = 0;
   double currDistance = 0;


   // loop through the linked list looking for the closest deer
   while (curr != null)
   {
     thisWolf = (Wolf) curr.getAgent();
     currDistance =
      thisWolf.getDistance(this.getXPos(),this.getYPos());
     if (currDistance < distance)
     {
       if (closest == null || currDistance < closestDistance)
       {
         closest = curr;
         closestDistance = currDistance;
       }
     }
     curr = (AgentNode) curr.getNext();
   }
   return closest;
 }
```

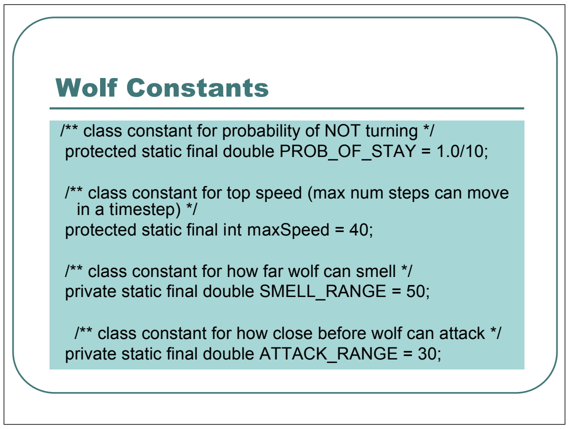


Deer new act()

Does this match the English description we had a few slides back?

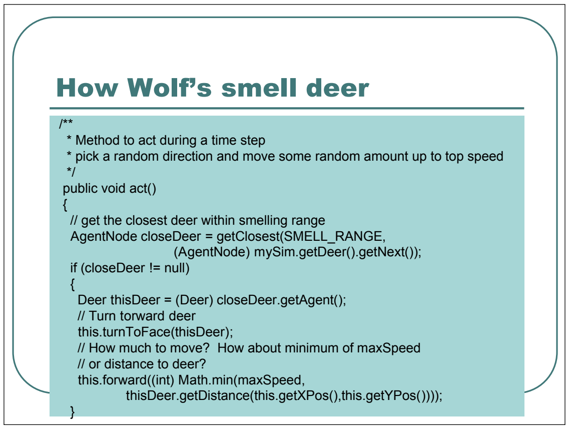
Think about this in terms of the values that can be changed and their relative values.

```
/**
  * Method to act during a time step
  * pick a random direction and move some random amount up
     to top speed
  */
 public void act()
 {
   // get the closest wolf within the smell range
   AgentNode closeWolf = getClosest(SMELL_RANGE,
                               (AgentNode)
      mySim.getWolves().getNext());

   if (closeWolf != null) {
     Wolf thisWolf = (Wolf) closeWolf.getAgent();
     // Turn to face the wolf
     this.turnToFace(thisWolf);
     // Now directly in the opposite direction
     this.turn(180);
     // How far to run? How about half of max speed??
     this.forward((int) (maxSpeed/2));
   }
   else {
     // if the random number is > prob of NOT turning then turn
     if (randNumGen.nextFloat() > PROB_OF_STAY)
     {
       this.turn(randNumGen.nextInt(360));
     }

     // go forward some random amount
     forward(randNumGen.nextInt(maxSpeed));
   }
 }
```



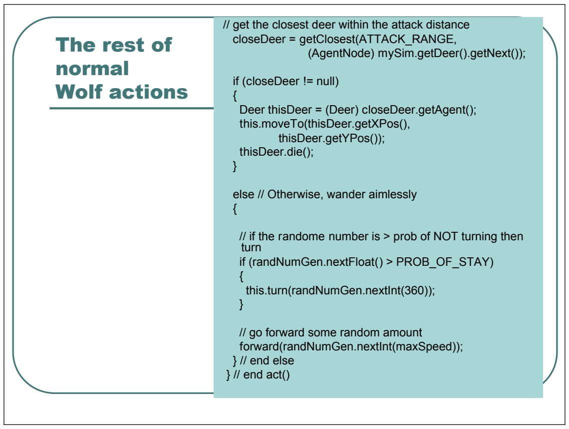
Wolf Constants

```
/** class constant for probability of NOT turning */
 protected static final double PROB_OF_STAY = 1.0/10;

 /** class constant for top speed (max num steps can move
    in a timestep) */
 protected static final int maxSpeed = 40;

 /** class constant for how far wolf can smell */
 private static final double SMELL_RANGE = 50;

   /** class constant for how close before wolf can attack */
 private static final double ATTACK_RANGE = 30;
```



How Wolf's smell deer

```
/**
  * Method to act during a time step
  * pick a random direction and move some random amount up to top speed
  */
 public void act()
 {
   // get the closest deer within smelling range
   AgentNode closeDeer = getClosest(SMELL_RANGE,
                            (AgentNode) mySim.getDeer().getNext());
   if (closeDeer != null)
   {
     Deer thisDeer = (Deer) closeDeer.getAgent();
     // Turn torward deer
     this.turnToFace(thisDeer);
     // How much to move?  How about minimum of maxSpeed
     // or distance to deer?
     this.forward((int) Math.min(maxSpeed,
                  thisDeer.getDistance(this.getXPos(),this.getYPos())));
   }
```



The rest of normal Wolf actions

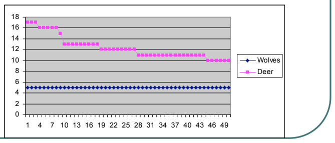
```
// get the closest deer within the attack distance
   closeDeer = getClosest(ATTACK_RANGE,
                       (AgentNode) mySim.getDeer().getNext());

   if (closeDeer != null)
   {
     Deer thisDeer = (Deer) closeDeer.getAgent();
     this.moveTo(thisDeer.getXPos(),
              thisDeer.getYPos());
     thisDeer.die();
   }

   else // Otherwise, wander aimlessly
   {

     // if the randome number is > prob of NOT turning then
      turn
     if (randNumGen.nextFloat() > PROB_OF_STAY)
     {
       this.turn(randNumGen.nextInt(360));
     }

     // go forward some random amount
     forward(randNumGen.nextInt(maxSpeed));
   } // end else
 } // end act()
```

Changes to WolfDeerSimulation…NOTHING!

- We have the same *interface* as we used to have, so *nothing* changes in WolfDeerSimulation.
- Very powerful idea:
  - *If changes to a class keep the* ***interface*** *the same, then all* ***users*** *of the class don't have to change at all.*

Running the new simulation

```
Welcome to DrJava.
> WolfDeerSimulation wds = new
    WolfDeerSimulation();
> wds.openFile("D:/cs1316/wds-
    chase.txt")
> wds.run();
```



Explorations

- What does the relative speed of Deer and Wolves matter?
  - Does it matter if Deer go faster? Wolves?
- What if Deer and Wolves can smell farther away?
  - What if one can smell better than the other?
- What's the effect of having more Deer or more Wolves?
- What if HungryWolves could starve (say at -10 satisfaction)? Do more deer live?

Doing More Simulations

- How much code would be in common in every simulation we'd build?
  - We already have lots of duplication, e.g., getClosest.
- Goal: Can we make an Agent/Actor class and Simulation class that we'd subclass with *very little* additional code to create new simulations?