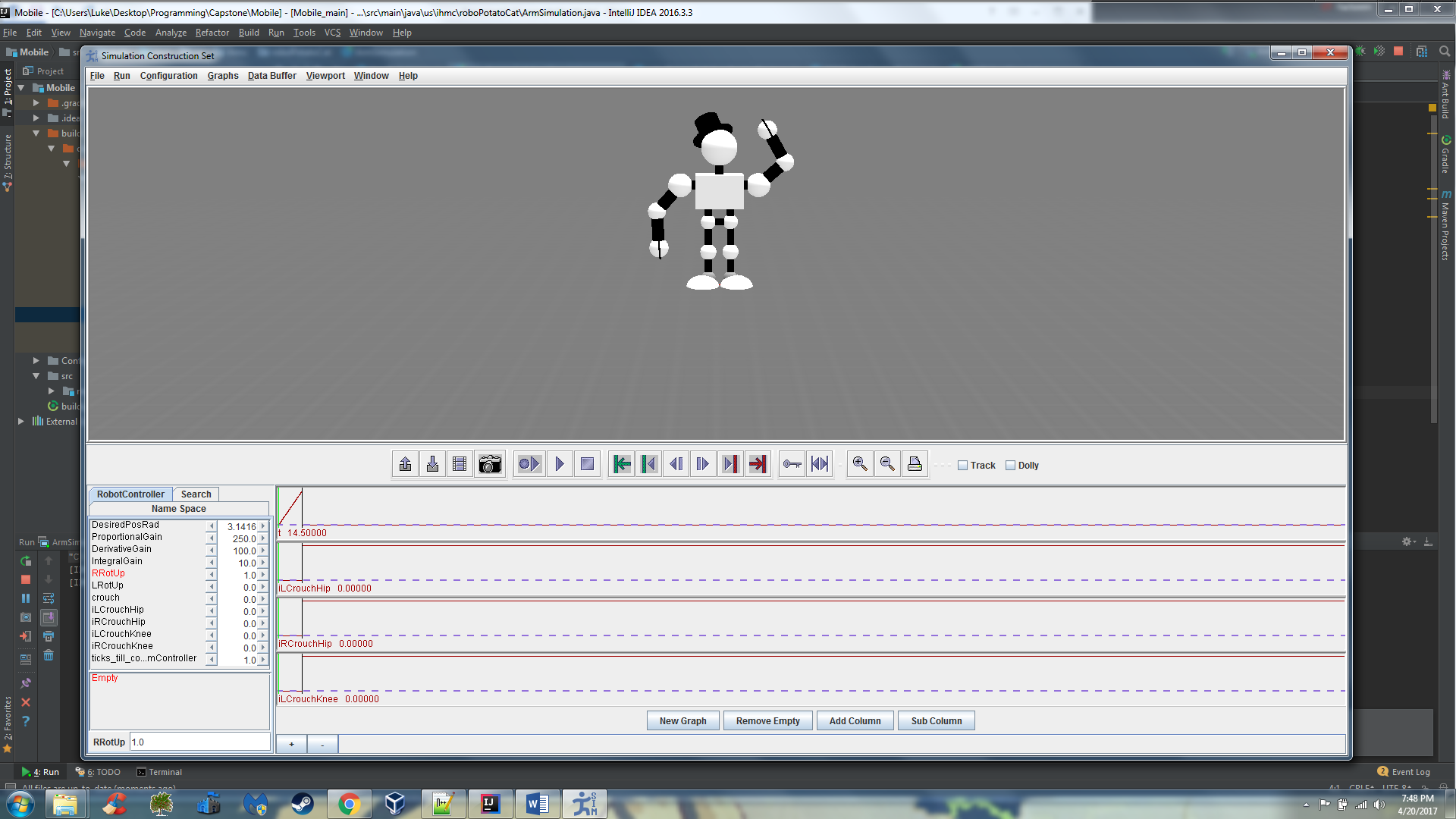
Super Amazing Awesome Tutorial For JDBot



-So you’ve done the other tutorials from IHMC, you think you know what you’re doing at this point, right?

No, of course you don’t. We sure didn’t. That’s why we’re here to help you understand what’s actually going on with this code. Hopefully by the end you will understand SCS well enough that you can create your very own robot... simulation!

Before Getting Started

-This tutorial is going to assume that you have installed SCS and have at least the pendulum tutorial done. Those can be found here:

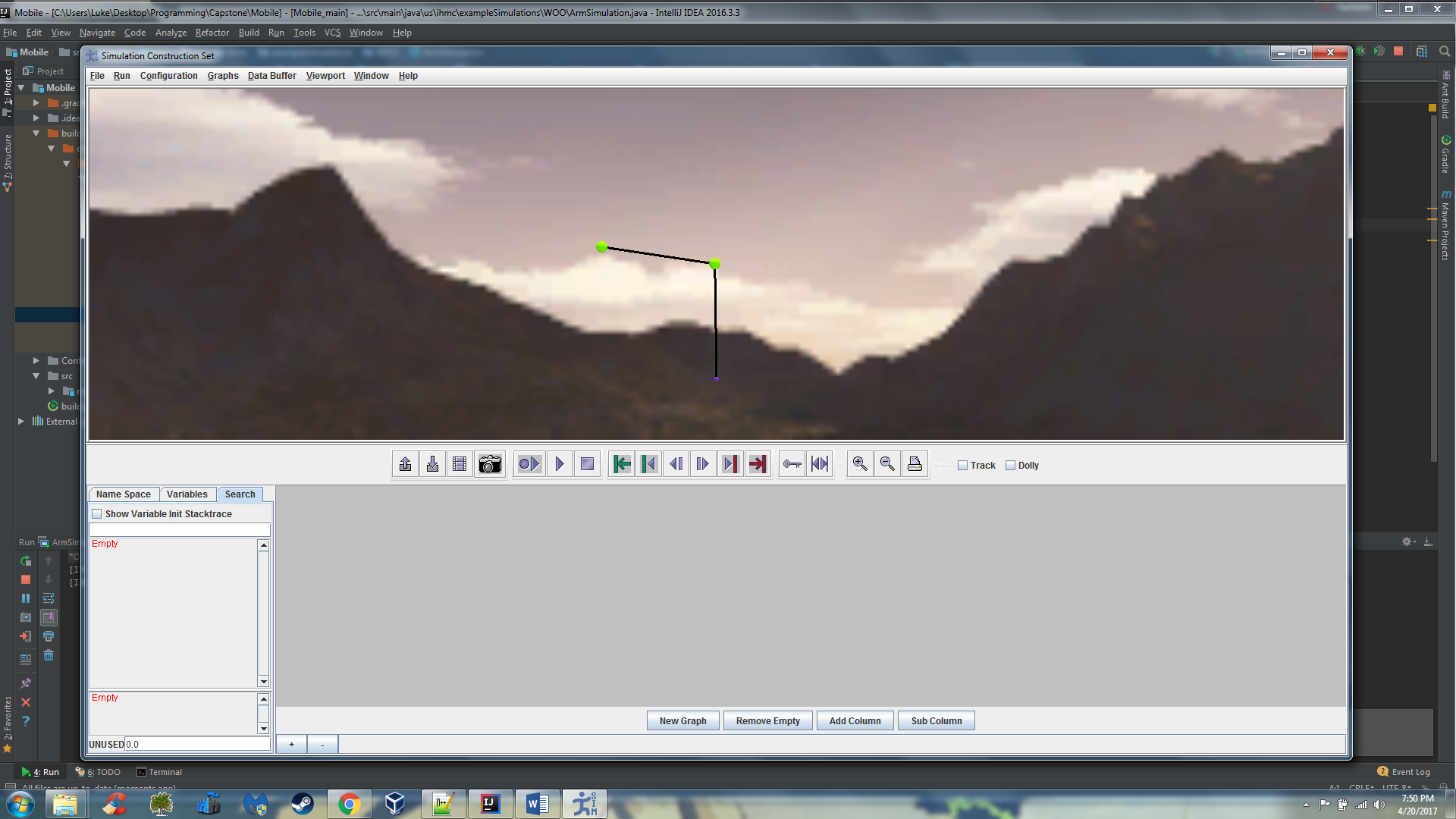
[Setting up SCS](https://ihmcrobotics.github.io/documentation/00-quickstart/00-quickstart/)

[Creating a pendulum](https://ihmcrobotics.github.io/documentation/20-scs/00-tutorials/02-creating-a-new-simulation/)

[Giving the pendulum a control](https://ihmcrobotics.github.io/documentation/20-scs/00-tutorials/03-adding-control-to-a-simulation/)

It’s also worth mentioning we created this project using IntelliJ. We like it better than Eclipse (we couldn’t figure out how to get SCS to work in Eclipse).

Creating WOOWOOBot



-WOOWOOBot will show you how to take a basic pendulum and add a second link to it, effectively creating a two-jointed pendulum. It will also show you some basic control over the individual joints.

If you have the basic pendulum already working with a control, you can use that as a base. First, right click exampleSimulations > New > Package. Name this package WOO. Now right click WOO > New > Java Class. Name this class ArmSimulation.

Now, the worst thing we did when we were learning SCS was simply copy/paste code from the tutorials without really understanding what they were doing in a desperate attempt to make things work. We didn’t fully understand how everything worked until the IHMC guys themselves explained it to us. With that in mind, we are giving you the full code to each of these classes. Please, take the time to learn about each piece of code as we go over it. You will still probably get stuck when making your own robots, but this will hopefully put you in a better place than just taking code, dropping it in, and expecting it to work (it won’t).

So here is the code for ArmSimulation:

package us.ihmc.exampleSimulations.WOO;  
  
import us.ihmc.simulationconstructionset.SimulationConstructionSet;  
import us.ihmc.simulationconstructionset.SimulationConstructionSetParameters;  
  
  
public class ArmSimulation  
{  
  
 public static final double *DT* = 0.001;  
 private SimulationConstructionSet sim;  
  
 public ArmSimulation()  
 {  
 ArmRobot robot = new ArmRobot();  
 robot.setController(new ArmController(robot));  
  
 SimulationConstructionSetParameters parameters = new SimulationConstructionSetParameters();  
 parameters.setDataBufferSize(32000);  
  
 sim = new SimulationConstructionSet(robot, parameters);  
 sim.setDT(*DT*, 20);  
 sim.setGroundVisible(false);  
 sim.setCameraPosition(0, -9.0, 0.6);  
 sim.setCameraFix(0.0, 0.0, 0.70);  
  
 sim.setSimulateDuration(60.0); // sets the simulation duration to 60 seconds  
  
 Thread myThread = new Thread(sim);  
 myThread.start();  
 }  
  
 public static void main(String[] args)  
 {  
 new ArmSimulation();  
 }  
}

This class sets up the environment for our robot, and will be acting as Main(). It determines things like the time between simulated frames, ground visibility, and camera position.

It will instantiate the other two classes we will be making.

ArmRobot robot = new ArmRobot();  
robot.setController(new ArmController(robot));

Don’t worry, the red lines are normal right now.

Next up, let’s make the ArmRobot class. Go through the same steps as before to create the class, and populate it with this code:

package us.ihmc.exampleSimulations.WOO;  
  
import us.ihmc.graphics3DAdapter.graphics.Graphics3DObject;  
import us.ihmc.graphics3DAdapter.graphics.appearances.YoAppearance;  
import us.ihmc.robotics.Axis;  
import us.ihmc.robotics.dataStructures.variable.DoubleYoVariable;  
import us.ihmc.simulationconstructionset.Link;  
import us.ihmc.simulationconstructionset.PinJoint;  
import us.ihmc.simulationconstructionset.Robot;  
  
import javax.vecmath.Vector3d;  
  
*/\*\*  
 \*  
 \* In this tutorial, lengths are expressed in meters (m), masses in kilograms (kg)  
 \*  
 \*/*public class ArmRobot extends Robot  
{  
 /\*  
 Define the parameters of the robot  
 \*/  
 public static final double *ROD\_LENGTH* = 1.0;  
 public static final double *ROD\_RADIUS* = 0.01;  
 public static final double *ROD\_MASS* = 0.00;  
  
 public static final double *FULCRUM\_RADIUS* = 0.02;  
  
 public static final double *BALL\_RADIUS* = 0.05;  
 public static final double *BALL\_MASS* = 1.0;  
  
 public static final double *FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_Y* =  
 *ROD\_LENGTH* \* *ROD\_LENGTH* \* *BALL\_MASS*; // I = mrˆ2 pendulum's resistance to changes to its rotation in kg.mˆ2  
  
 /\*  
 Initial state of the pendulum  
 \*/  
  
 private double fulcrumInitialPositionDegrees = 0.0;  
 private double fulcrumInitialPositionRadians = fulcrumInitialPositionDegrees \* Math.*PI* / 180.0;  
 private double fulcrumInitialVelocity = 0.0;  
  
 /\* Some joint state variables \*/  
 private DoubleYoVariable tau\_fulcrum, q\_fulcrum, qd\_fulcrum; // Respectively Torque, Position, Velocity  
 private DoubleYoVariable tau\_Second, q\_Second, qd\_Second; // Respectively Torque, Position, Velocity  
  
 /\*  
 Define its constructor  
 \*/  
 public ArmRobot()  
 {  
 //This names our robot  
 super("pendulum");  
  
 //Adding the initial joint  
 PinJoint fulcrumPinJoint = new PinJoint("FulcrumPin", new Vector3d(0.0, 0.0, 1.25), this, Axis.*Y*);  
 //and the second joint  
 PinJoint secondPinJoint = new PinJoint("SecondPin", new Vector3d(0.0, 0.0, -*ROD\_LENGTH*), this, Axis.*Y*);  
  
 //this is optional, it sets the position of the first pendulum, so that it will be swinging instead of pointing straight down  
 fulcrumPinJoint.setInitialState(fulcrumInitialPositionRadians, fulcrumInitialVelocity);  
  
 //this adds the graphics and physical mass of the joint, as defined in the pendulumLink() method at the bottom  
 fulcrumPinJoint.setLink(pendulumLink());  
  
 //sets a small amount of friction on the first joint  
 fulcrumPinJoint.setDamping(0.3);  
  
 //now lets do all of those things for the second joint, too  
 secondPinJoint.setInitialState(-fulcrumInitialPositionRadians, fulcrumInitialVelocity);  
  
 //we can use the same method for the second joint because it is identical  
 //to the first  
 secondPinJoint.setLink(pendulumLink());  
  
 secondPinJoint.setDamping(0.3);  
  
 //setting some physics stuff, these are variables defined above  
 q\_fulcrum = fulcrumPinJoint.getQ();  
 qd\_fulcrum = fulcrumPinJoint.getQD();  
 tau\_fulcrum = fulcrumPinJoint.getTau();  
  
 q\_Second = secondPinJoint.getQ();  
 qd\_Second = secondPinJoint.getQD();  
 tau\_Second = secondPinJoint.getTau();  
  
 //now we connect the joints together  
 fulcrumPinJoint.addJoint(secondPinJoint);  
  
 //and now we add the root joint  
 this.addRootJoint(fulcrumPinJoint);  
 }  
  
 //these methods are for the controller, which we will go over shortly  
 */\*\*  
 \* Fulcrum's angular position in radians  
 \** ***@return*** *angular position in radians  
 \*/* public double getFulcrumAngularPosition()  
 {  
 return q\_fulcrum.getDoubleValue();  
 }  
 public double getSecondAngularPosition()  
 {  
 return q\_Second.getDoubleValue();  
 }  
  
 */\*\*  
 \* Fulcrum's angular velocity in radians per seconds  
 \** ***@return*** *angular velocity in radians per seconds  
 \*/* public double getFulcrumAngularVelocity()  
 {  
 return qd\_fulcrum.getDoubleValue();  
 }  
 public double getSecondAngularVelocity()  
 {  
 return qd\_Second.getDoubleValue();  
 }  
  
 */\*\*  
 \* Fulcrum's torque in Newton meter  
 \** ***@return*** *Torque in Newton meter  
 \*/* public double getFulcrumTorque()  
 {  
 return tau\_fulcrum.getDoubleValue();  
 }  
  
 */\*\*  
 \* Set Fulcrum's torque in Newton meter  
 \** ***@return*** *Torque in Newton meter  
 \*/* public void setFulcrumTorque(double tau)  
 {  
 this.tau\_fulcrum.set(tau);  
 }  
  
 public void setSecondTorque(double tau)  
 {  
 this.tau\_Second.set(tau);  
 }  
  
 */\*\*  
 \* Create the first link for the DoublePendulumRobot.  
 \*/* //This creates graphics and physics for our joints  
 private Link pendulumLink()  
 {  
 //instantiating the "Link"  
 Link pendulumLink = new Link("PendulumLink");  
 pendulumLink.setMomentOfInertia(0.0, *FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_Y*, 0.0);  
 //creating mass  
 pendulumLink.setMass(*BALL\_MASS*);//currently 1  
  
 //instantiating a graphics object  
 Graphics3DObject pendulumGraphics = new Graphics3DObject();  
 //creates the small red sphere at the center point  
 pendulumGraphics.addSphere(*FULCRUM\_RADIUS*, YoAppearance.*Crimson*());  
 //moves us to the end of the rod  
 pendulumGraphics.translate(0.0, 0.0, -*ROD\_LENGTH*);  
 //creates the black cylinder that makes up the rod  
 pendulumGraphics.addCylinder(*ROD\_LENGTH*, *ROD\_RADIUS*, YoAppearance.*Black*());  
 //adds the big yellow sphere at the end  
 pendulumGraphics.addSphere(*BALL\_RADIUS*, YoAppearance.*Chartreuse*());  
 pendulumLink.setLinkGraphics(pendulumGraphics);  
  
 return pendulumLink;  
 }  
  
}

So, this one is a lot bigger.

We explain what most of the individual pieces of code are doing with our in-code comments, but the gist is that this class is what creates our robot. We use this class to create each joint, assign those joints graphics and physics, and set up some variables for use by the controllers (more on that momentarily). Take the time to read through this one, it is where you will spend the majority of your time when making your own bot.

public ArmRobot()  
{  
 //This names our robot  
 super("pendulum");  
  
 //Adding the initial joint  
 PinJoint fulcrumPinJoint = new PinJoint("FulcrumPin", new Vector3d(0.0, 0.0, 1.25), this, Axis.*Y*);  
 //and the second joint  
 PinJoint secondPinJoint = new PinJoint("SecondPin", new Vector3d(0.0, 0.0, -*ROD\_LENGTH*), this, Axis.*Y*);  
  
 //this is optional, it sets the position of the first pendulum, so that it will be swinging instead of pointing straight down  
 fulcrumPinJoint.setInitialState(fulcrumInitialPositionRadians, fulcrumInitialVelocity);  
  
 //this adds the graphics and physical mass of the joint, as defined in the pendulumLink() method at the bottom  
 fulcrumPinJoint.setLink(pendulumLink());  
  
 //sets a small amount of friction on the first joint  
 fulcrumPinJoint.setDamping(0.3);  
  
 //now lets do all of those things for the second joint, too  
 secondPinJoint.setInitialState(-fulcrumInitialPositionRadians, fulcrumInitialVelocity);  
  
 //we can use the same method for the second joint because it is identical  
 //to the first  
 secondPinJoint.setLink(pendulumLink());  
  
 secondPinJoint.setDamping(0.3);  
  
 //setting some physics stuff, these are variables defined above  
 q\_fulcrum = fulcrumPinJoint.getQ();  
 qd\_fulcrum = fulcrumPinJoint.getQD();  
 tau\_fulcrum = fulcrumPinJoint.getTau();  
  
 q\_Second = secondPinJoint.getQ();  
 qd\_Second = secondPinJoint.getQD();  
 tau\_Second = secondPinJoint.getTau();  
  
 //now we connect the joints together  
 fulcrumPinJoint.addJoint(secondPinJoint);  
  
 //and now we add the root joint  
 this.addRootJoint(fulcrumPinJoint);  
}

This constructor is where most of the magic happens. We go through a few key steps:

1. Create our joints
2. Give our joints starting positions and friction
3. Calling a method to give our joints graphics and mass
4. Set some variables that determine the physics of our joints
5. Link our joints together
6. Declare our root joint (more details later)
7. We don’t do this here, but later there will be a step to add ground contact points to each of our joints so they can fall on a surface, rather than the void.

//This creates graphics and physics for our joints  
private Link pendulumLink()  
{  
 //instantiating the "Link"  
 Link pendulumLink = new Link("PendulumLink");  
 pendulumLink.setMomentOfInertia(0.0, *FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_Y*, 0.0);  
 //creating mass  
 pendulumLink.setMass(*BALL\_MASS*);//currently 1  
  
 //instantiating a graphics object  
 Graphics3DObject pendulumGraphics = new Graphics3DObject();  
 //creates the small red sphere at the center point  
 pendulumGraphics.addSphere(*FULCRUM\_RADIUS*, YoAppearance.*Crimson*());  
 //moves us to the end of the rod  
 pendulumGraphics.translate(0.0, 0.0, -*ROD\_LENGTH*);  
 //creates the black cylinder that makes up the rod  
 pendulumGraphics.addCylinder(*ROD\_LENGTH*, *ROD\_RADIUS*, YoAppearance.*Black*());  
 //adds the big yellow sphere at the end  
 pendulumGraphics.addSphere(*BALL\_RADIUS*, YoAppearance.*Chartreuse*());  
 pendulumLink.setLinkGraphics(pendulumGraphics);  
  
 return pendulumLink;  
}

This method is at the very bottom of the class. It gives our joints graphics and mass on the balls at the end. You need to understand this part if you want to give your robot a top hat.

If you are coming straight from the pendulum tutorial, you will notice that a lot of the code and many of the names are identical to that project. That is because this is literally just Simple Pendulum with one extra piece and some swinging action.

This is the last piece: the ArmController.

package us.ihmc.exampleSimulations.WOO;  
  
 import us.ihmc.robotics.dataStructures.registry.YoVariableRegistry;  
 import us.ihmc.robotics.dataStructures.variable.DoubleYoVariable;  
 import us.ihmc.simulationconstructionset.robotController.RobotController;  
  
  
public class ArmController implements RobotController  
{  
 // A name for this controller  
 private final String name = "pendulumController";  
  
 // This line instantiates a registry that will contain relevant controller variables that will be accessible from the simulation panel.  
 private final YoVariableRegistry registry = new YoVariableRegistry("PendulumController");  
 //private final YoVariableRegistry secondRegistry = new YoVariableRegistry("SecondController");  
  
 // This is a reference to the SimplePendulumRobot that enables the controller to access this robot's variables.  
 private ArmRobot robot;  
  
 /\* Control variables: \*/  
  
 // Target angle  
 private DoubleYoVariable desiredPositionRadians;  
  
 // Controller parameter variables  
 private DoubleYoVariable p\_gain, d\_gain, i\_gain;  
  
 // This is the desired torque that we will apply to the fulcrum joint (PinJoint)  
 private double torque;  
  
 /\* Constructor:  
 Where we instantiate and initialize control variables  
 \*/  
 public ArmController(ArmRobot robot)  
 {  
 this.robot = robot;  
 desiredPositionRadians = new DoubleYoVariable("DesiredPosRad", registry);  
 desiredPositionRadians.set(Math.*PI*);  
  
 p\_gain = new DoubleYoVariable("ProportionalGain", registry);  
 p\_gain.set(250.0);  
 d\_gain = new DoubleYoVariable("DerivativeGain", registry);  
 d\_gain.set(100.0);  
 i\_gain = new DoubleYoVariable("IntegralGain", registry);  
 i\_gain.set(10.0);  
 }  
  
 public void initialize()  
 {  
  
 }  
  
 private double positionError = 0;  
 private double integralError = 0;  
 private int i = 0; //counter  
  
 //this is a loop that calls our controllers. We have one for each joint,   
 //but there are other ways this could be set up  
 public void doControl()  
 {  
 fulcrumController();  
 secondController();  
 }  
  
 //this is the meat of this class  
 public void fulcrumController()  
 {  
 //This looks a little tricky, but it's simpler than it seems  
   
 // ERROR term: Compute the difference between the desired position the pendulum and its current position  
 positionError = desiredPositionRadians.getDoubleValue() - (robot.getFulcrumAngularPosition());  
 // ^desired position is currently defined as Pi. When you want to change the angle of the joint,  
 // this is where you do it.  
  
 // INTEGRAL term: Compute a simple numerical integration of the position error  
 integralError += positionError \* ArmSimulation.*DT*; //  
  
 // P.I.D  
 torque = p\_gain.getDoubleValue() \* positionError +  
 i\_gain.getDoubleValue() \* integralError +  
 d\_gain.getDoubleValue() \* (0 - robot.getFulcrumAngularVelocity());  
  
 //here we set the torque to apply to the joint based on our desired position, this will push the joint and hold  
 //it where we want it  
 robot.setFulcrumTorque(torque);  
 }  
  
 public void secondController()  
 {  
 //This one actually is a little tricky, because we a re changing our desired position between two different   
 //values over time.  
 if (i == 1000)  
 {  
 i = 2000;  
 }  
 else if (i == 1001)  
 {  
 i = 0;  
 }  
 if (i > 1000)  
 {  
  
 positionError = desiredPositionRadians.getDoubleValue() - (robot.getSecondAngularPosition()+1);  
 i--;  
 }  
 if (i < 1000)  
 {  
 positionError = desiredPositionRadians.getDoubleValue() - (robot.getSecondAngularPosition()+5);  
 i++;  
 }  
  
 // INTEGRAL term: Compute a simple numerical integration of the position error  
 integralError += positionError \* ArmSimulation.*DT*; //  
  
 // P.I.D  
 torque = p\_gain.getDoubleValue() \* positionError +  
 i\_gain.getDoubleValue() \* integralError +  
 d\_gain.getDoubleValue() \* (0 - robot.getSecondAngularVelocity());  
  
 //here we set the torque for our second joint  
 robot.setSecondTorque(torque);  
 }  
  
 public YoVariableRegistry getYoVariableRegistry()  
 {  
 return registry;  
 }  
  
 public String getName()  
 {  
 return name;  
 }  
  
 public String getDescription()  
 {  
 return name;  
 }  
}

This one looks more complex than it is. There are a few moving parts here that you don’t necessarily need to understand in order to proceed, but you should understand them anyway. It may bite you in the butt if you don’t.

//this is a loop that calls our controllers. We have one for each joint,  
//but there are other ways this could be set up  
public void doControl()  
{  
 fulcrumController();  
 secondController();  
}

This method is where the magic happens. It loops through once every simulated frame, calling the methods that tell our joints where they should be.

//this is the meat of this class  
public void fulcrumController()  
{  
 //This looks a little tricky, but it's simpler than it seems  
  
 // ERROR term: Compute the difference between the desired position the pendulum and its current position  
 positionError = desiredPositionRadians.getDoubleValue() - (robot.getFulcrumAngularPosition());  
 // ^desired position is currently defined as Pi. When you want to change the angle of the joint,  
 // this is where you do it.  
  
 // INTEGRAL term: Compute a simple numerical integration of the position error  
 integralError += positionError \* ArmSimulation.*DT*; //  
  
 // P.I.D  
 torque = p\_gain.getDoubleValue() \* positionError +  
 i\_gain.getDoubleValue() \* integralError +  
 d\_gain.getDoubleValue() \* (0 - robot.getFulcrumAngularVelocity());  
  
 //here we set the torque to apply to the joint based on our desired position, this will push the joint and hold  
 //it where we want it  
 robot.setFulcrumTorque(torque);  
}

Let’s take a look at this method…

It has a bunch of funky math. Let’s fix that by explaining what is going on.

If you have taken a precalculus course in highschool then you have the basic knowledge to understand this. Remember integrals? In the simplest terms, this method is doing math to form an integral curve. It pulls in the position we want (desiredPositionRadians), the position we are at (getFulcrumAngularPosition), and our current velocity (getFulcrumAngularVelocity) in order to produce a certain quantity of torque to push us where we need to go (setFulcrumTorque). p\_gain, i\_gain, and d\_gain deal with position, velocity, and acceleration respectively.

Remember that getFulcrumAngularPosition, getFulcrumAngularVelocity, and setFulcrumTorque are all methods that were defined in ArmRobot specifically for the first joint. The second joint methods have different names.

If you didn’t understand any of that, it’s okay. All you really need to know for this tutorial is that adjusting desiredPositionRadians will change the angle of the joint.

Our desired position is currently defined as Pi. That sets the first joint’s angle straight up in the air. If we were to set our second joint to Pi in this situation, it would point straight down into the center. That is because the position is based on the position of the first joint. For the second joint, straight up is out, and straight down is in. If we were to set the position of the second joint to zero, both pendulums would point straight up.

public void secondController()  
{  
 //This one actually is a little tricky, because we a re changing our desired position between two different  
 //values over time.  
 if (i == 1000)  
 {  
 i = 2000;  
 }  
 else if (i == 1001)  
 {  
 i = 0;  
 }  
 if (i > 1000)  
 {  
  
 positionError = (desiredPositionRadians.getDoubleValue()+1) - robot.getSecondAngularPosition();  
 i--;  
 }  
 if (i < 1000)  
 {  
 positionError = (desiredPositionRadians.getDoubleValue()+5) - robot.getSecondAngularPosition();  
 i++;  
 }  
  
 // INTEGRAL term: Compute a simple numerical integration of the position error  
 integralError += positionError \* ArmSimulation.*DT*; //  
  
 // P.I.D  
 torque = p\_gain.getDoubleValue() \* positionError +  
 i\_gain.getDoubleValue() \* integralError +  
 d\_gain.getDoubleValue() \* (0 - robot.getSecondAngularVelocity());  
  
 //here we set the torque for our second joint  
 robot.setSecondTorque(torque);  
}

The second controller is a bit different. All the first joint has to do is hold itself straight up. This one has to move back and forth in the signature dance of the WOOWOO tribe.

i is a variable that we created near the top of the class. It acts as a counter, which counts each frame of simulation. Because of our current settings, that means 1000 is equal one second. Using some coding magic (if statements), we can use this to set the desired position between

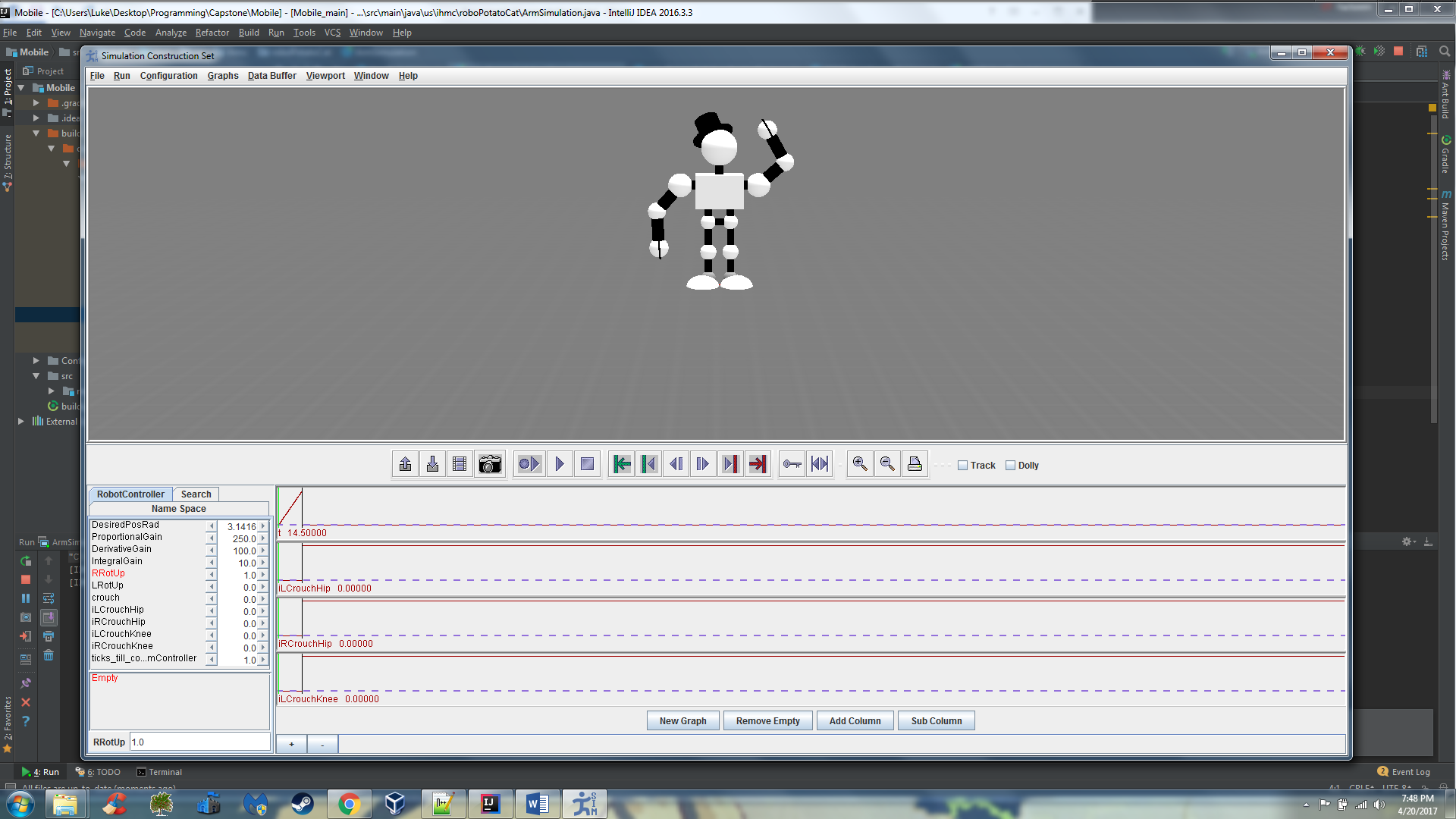
(desiredPositionRadians.getDoubleValue()+1)

and

(desiredPositionRadians.getDoubleValue()+5)

Were you wondering why all of the classes are named Arm? It’s because you just built a robot arm. That’s it. You have a shoulder joint and an elbow joint. Now let’s attach it to the rest of the robot.

Creating JD



- With that you should be able to run WOOWOOBot and see his sick moves. This has given you a basic idea of how most of the pieces of JD will work. He has the same three classes (still named Arm – yes we are lazy), and most of the stuff you need to understand is done. There are a few things left, though, and they’re fairly sizable. We need a floor, and we need JD to not fall through the floor, and we need to understand a little bit more about root joints. First, let’s create another new package under exampleSimulations called JD. Next add the three classes: ArmSimulation, ArmRobot, and ArmController. Here is ArmSimulation:

package us.ihmc.exampleSimulations.JD;  
  
import us.ihmc.graphics3DAdapter.graphics.Graphics3DObject;  
import us.ihmc.simulationconstructionset.SimulationConstructionSet;  
import us.ihmc.simulationconstructionset.SimulationConstructionSetParameters;  
  
  
public class ArmSimulation  
{  
  
 public static final double *DT* = 0.001;  
 private SimulationConstructionSet sim;  
  
 public ArmSimulation()  
 {  
 ArmRobot robot = new ArmRobot();  
 robot.setController(new ArmController(robot));  
  
 SimulationConstructionSetParameters parameters = new SimulationConstructionSetParameters();  
 parameters.setDataBufferSize(32000);  
  
 sim = new SimulationConstructionSet(robot, parameters);  
 sim.setDT(*DT*, 20);  
 sim.setGroundVisible(true);  
 sim.setCameraPosition(0, -50.0, 8);  
 sim.setCameraFix(0.0, 0.0, 0.70);  
  
 sim.setSimulateDuration(60.0); // sets the simulation duration to 60 seconds  
  
 // x, y, z = red, white, blue  
 Graphics3DObject coordinateSystem = new Graphics3DObject();  
 coordinateSystem.addCoordinateSystem(0.5);  
 sim.addStaticLinkGraphics(coordinateSystem);  
  
 Thread myThread = new Thread(sim);  
 myThread.start();  
 }  
  
 public static void main(String[] args)  
 {  
 new ArmSimulation();  
 }  
}

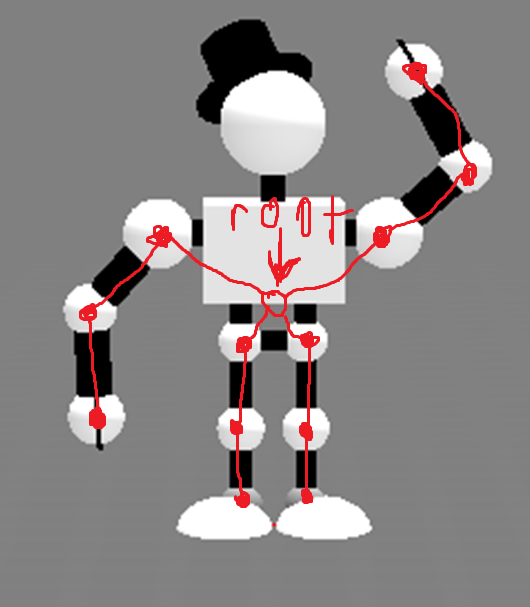
There are only a few differences here, the main one being that we have enabled graphics for the ground.

Now for the big one, ArmRobot:

package us.ihmc.exampleSimulations.JD;  
  
import us.ihmc.graphics3DAdapter.GroundProfile3D;  
import us.ihmc.graphics3DAdapter.graphics.Graphics3DObject;  
import us.ihmc.graphics3DAdapter.graphics.appearances.YoAppearance;  
import us.ihmc.robotics.Axis;  
import us.ihmc.robotics.dataStructures.variable.DoubleYoVariable;  
import us.ihmc.simulationconstructionset.\*;  
import us.ihmc.simulationconstructionset.util.LinearGroundContactModel;  
import us.ihmc.simulationconstructionset.util.ground.FlatGroundProfile;  
  
import javax.vecmath.Vector3d;  
  
*/\*\*  
 \*  
 \* lengths are expressed in meters (m), masses in kilograms (kg)  
 \*  
 \*/*public class ArmRobot extends Robot  
{  
 /\*  
 Define the parameters of the robot  
 \*/  
  
  
 public static final double *SERVO\_JOINT\_LENGTH* = 1; //1 = 60mm everything is extrapolated from that  
 //we multiply by INCH\_TO\_MILLIMETER to get however many inches we need  
 public static final double *INCH\_TO\_MILLIMETER* = (1 \* 25.4)/60;  
 public static final double *ROD\_RADIUS* = 0.01;  
 public static final double *ROD\_MASS* = 0.00;  
  
 public static final double *FULCRUM\_RADIUS* = 0.02;  
  
 public static final double *BALL\_RADIUS* = 0.05;  
 public static final double *BALL\_MASS* = 1.0;  
  
 public static final double *FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X* =  
 *SERVO\_JOINT\_LENGTH* \* *SERVO\_JOINT\_LENGTH* \* *BALL\_MASS*; // I = mrˆ2 pendulum's resistance to changes to its rotation in kg.mˆ2  
  
 private double fulcrumInitialPositionDegrees = 90.0;  
 private double fulcrumInitialPositionRadians = fulcrumInitialPositionDegrees \* Math.*PI* / 180.0;  
 private double fulcrumInitialVelocity = 0.0;  
  
 /\* Some joint state variables \*/  
 private DoubleYoVariable tau\_LRotator, q\_LRotator, qd\_LRotator; // Respectively Torque, Position, Velocity  
 private DoubleYoVariable tau\_RRotator, q\_RRotator, qd\_RRotator;  
 private DoubleYoVariable tau\_LFlapper, q\_LFlapper, qd\_LFlapper;  
 private DoubleYoVariable tau\_RFlapper, q\_RFlapper, qd\_RFlapper;  
 private DoubleYoVariable tau\_LElbow, q\_LElbow, qd\_LElbow;  
 private DoubleYoVariable tau\_RElbow, q\_RElbow, qd\_RElbow;  
 private DoubleYoVariable tau\_LHip, q\_LHip, qd\_LHip;  
 private DoubleYoVariable tau\_RHip, q\_RHip, qd\_RHip;  
 private DoubleYoVariable tau\_LKnee, q\_LKnee, qd\_LKnee;  
 private DoubleYoVariable tau\_RKnee, q\_RKnee, qd\_RKnee;  
 private DoubleYoVariable tau\_LAnkle, q\_LAnkle, qd\_LAnkle;  
 private DoubleYoVariable tau\_RAnkle, q\_RAnkle, qd\_RAnkle;  
  
  
  
 /\*  
 Define its constructor  
 \*/  
 public ArmRobot()  
 {  
  
 //our name  
 super("JD");  
  
 //instantiating our root joint, located at the bottom center of JD's box body  
 FloatingJoint rootJoint = new FloatingJoint("FulcrumPin", new Vector3d(), this);  
 rootJoint.setPosition(0, 0,3);  
  
 //use this if you would like to put his root in a 2D planar  
// FloatingPlanarJoint rootJoint = new FloatingPlanarJoint("FulcrumPin", this, XZ);  
// rootJoint.changeOffsetVector(0, 0, 3); //2.65 is right at ground  
  
 //instantiate new joints here - the vector3d is the point in space that the new part exists in in relation to the previous joint  
 PinJoint rightShoulderRotator = new PinJoint("rightShoulderRotator", new Vector3d(3\**INCH\_TO\_MILLIMETER*, 0.0, .75), this, Axis.*X*);//make sure to measure jd and adjust these Zs  
 PinJoint leftShoulderRotator = new PinJoint("leftShoulderRotator", new Vector3d(-3\**INCH\_TO\_MILLIMETER*, 0.0, 0.75), this, Axis.*X*);  
 PinJoint rightShoulderFlapper = new PinJoint("rightShoulderFlapper", new Vector3d(0.0625\**INCH\_TO\_MILLIMETER*, 0.0, 0), this, Axis.*Y*);  
 PinJoint leftShoulderFlapper = new PinJoint("leftShoulderFlapper", new Vector3d(-0.0625\**INCH\_TO\_MILLIMETER*, 0.0, 0), this, Axis.*Y*);  
 PinJoint rightElbow = new PinJoint("rightElbow", new Vector3d(2.75\**INCH\_TO\_MILLIMETER*, 0.0, 0), this, Axis.*Y*);  
 PinJoint leftElbow = new PinJoint("leftElbow", new Vector3d(-2.75\**INCH\_TO\_MILLIMETER*, 0.0, 0), this, Axis.*Y*);  
 PinJoint rightHand = new PinJoint("rightHand", new Vector3d(2.875\**INCH\_TO\_MILLIMETER*, 0.0, 0), this, Axis.*X*);  
 PinJoint leftHand = new PinJoint("leftHand", new Vector3d(-2.875\**INCH\_TO\_MILLIMETER*, 0.0, 0), this, Axis.*X*);  
  
 PinJoint rightHip = new PinJoint("rightHip", new Vector3d(0.875\**INCH\_TO\_MILLIMETER*, 0.0, -1.125\**INCH\_TO\_MILLIMETER*), this, Axis.*X*);  
 PinJoint leftHip = new PinJoint("leftHip", new Vector3d(-0.875\**INCH\_TO\_MILLIMETER*, 0.0, -1.125\**INCH\_TO\_MILLIMETER*), this, Axis.*X*);  
 PinJoint rightKnee = new PinJoint("rightKnee", new Vector3d(0, 0, -2.375\**INCH\_TO\_MILLIMETER*), this, Axis.*X*);  
 PinJoint leftKnee = new PinJoint("leftKnee", new Vector3d(0, 0, -2.375\**INCH\_TO\_MILLIMETER*), this, Axis.*X*);  
 PinJoint rightAnkle = new PinJoint("rightAnkle", new Vector3d(0, 0.0, -2.125\**INCH\_TO\_MILLIMETER*), this, Axis.*Y*);  
 PinJoint leftAnkle = new PinJoint("leftAnkle", new Vector3d(0, 0.0, -2.125\**INCH\_TO\_MILLIMETER*), this, Axis.*Y*);  
  
 //for foot contact points  
 PinJoint r1 = new PinJoint("r1", new Vector3d(-1\**INCH\_TO\_MILLIMETER*, -3\**INCH\_TO\_MILLIMETER*, -0.5\**INCH\_TO\_MILLIMETER*), this, Axis.*Y*);  
 PinJoint r2 = new PinJoint("r2", new Vector3d(-1\**INCH\_TO\_MILLIMETER*, 0.0, -0.5\**INCH\_TO\_MILLIMETER*), this, Axis.*Y*);  
 PinJoint r3 = new PinJoint("r3", new Vector3d(-1\**INCH\_TO\_MILLIMETER*, 1.5\**INCH\_TO\_MILLIMETER*, -0.5\**INCH\_TO\_MILLIMETER*), this, Axis.*Y*);  
 PinJoint r4 = new PinJoint("r4", new Vector3d(1.75\**INCH\_TO\_MILLIMETER*, 1.75\**INCH\_TO\_MILLIMETER*, -0.5\**INCH\_TO\_MILLIMETER*), this, Axis.*Y*);  
 PinJoint r5 = new PinJoint("r5", new Vector3d(1.75\**INCH\_TO\_MILLIMETER*, 0.0, -0.5\**INCH\_TO\_MILLIMETER*), this, Axis.*Y*);  
 PinJoint r6 = new PinJoint("r6", new Vector3d(1\**INCH\_TO\_MILLIMETER*, -3\**INCH\_TO\_MILLIMETER*, -0.5\**INCH\_TO\_MILLIMETER*), this, Axis.*Y*);  
  
 PinJoint l1 = new PinJoint("l1", new Vector3d(1\**INCH\_TO\_MILLIMETER*, -3\**INCH\_TO\_MILLIMETER*, -0.5\**INCH\_TO\_MILLIMETER*), this, Axis.*Y*);  
 PinJoint l2 = new PinJoint("l2", new Vector3d(1\**INCH\_TO\_MILLIMETER*, 0.0, -0.5\**INCH\_TO\_MILLIMETER*), this, Axis.*Y*);  
 PinJoint l3 = new PinJoint("l3", new Vector3d(1\**INCH\_TO\_MILLIMETER*, 1.5\**INCH\_TO\_MILLIMETER*, -0.5\**INCH\_TO\_MILLIMETER*), this, Axis.*Y*);  
 PinJoint l4 = new PinJoint("l4", new Vector3d(-1.75\**INCH\_TO\_MILLIMETER*, 1.75\**INCH\_TO\_MILLIMETER*, -0.5\**INCH\_TO\_MILLIMETER*), this, Axis.*Y*);  
 PinJoint l5 = new PinJoint("l5", new Vector3d(-1.75\**INCH\_TO\_MILLIMETER*, 0.0, -0.5\**INCH\_TO\_MILLIMETER*), this, Axis.*Y*);  
 PinJoint l6 = new PinJoint("l6", new Vector3d(-1\**INCH\_TO\_MILLIMETER*, -3\**INCH\_TO\_MILLIMETER*, -0.5\**INCH\_TO\_MILLIMETER*), this, Axis.*Y*);  
  
 //limit stops are like controls, they create boundaries that the joints can't cross accidentally  
 leftShoulderFlapper.setLimitStops(-Math.*PI*/2,Math.*PI*/2,10,50);  
 leftElbow.setLimitStops(-Math.*PI*/2,Math.*PI*/2,10,50);  
 rightShoulderFlapper.setLimitStops(-Math.*PI*/2,Math.*PI*/2,10,50);  
 rightElbow.setLimitStops(-Math.*PI*/2,Math.*PI*/2,10,50);  
  
 rightHip.setLimitStops(-Math.*PI*/2,Math.*PI*/2, 10, 50);  
 leftHip.setLimitStops(-Math.*PI*/2,Math.*PI*/2, 10, 50);  
 rightKnee.setLimitStops(-Math.*PI*/2,Math.*PI*/2, 10, 100);  
 leftKnee.setLimitStops(-Math.*PI*/2,Math.*PI*/2, 10, 100);  
  
  
 //damping = how tight the joints are  
 rightShoulderRotator.setDamping(0.3);  
 leftShoulderRotator.setDamping(0.3);  
 rightShoulderFlapper.setDamping(0.3);  
 leftShoulderFlapper.setDamping(0.3);  
 rightElbow.setDamping(0.3);  
 leftElbow.setDamping(0.3);  
 rightHand.setDamping(0.3);  
 leftHand.setDamping(0.3);  
  
 rightHip.setDamping(0.3);  
 leftHip.setDamping(0.3);  
 rightKnee.setDamping(0.3);  
 leftKnee.setDamping(0.3);  
 rightAnkle.setDamping(0.3);  
 leftAnkle.setDamping(0.3);  
  
 //assign a graphic and mass  
 rightShoulderRotator.setLink(servoPinAxisGraphicR());  
 leftShoulderRotator.setLink(servoPinAxisGraphicL());  
 rightShoulderFlapper.setLink(testSphereArmThighR());  
 leftShoulderFlapper.setLink(testSphereArmThighL());  
 rightElbow.setLink(testSphereForearmR());  
 leftElbow.setLink(testSphereForearmL());  
 rightHand.setLink(testSphereHand());  
 leftHand.setLink(testSphereHand());  
  
 rightHip.setLink(testSphereThighR());  
 leftHip.setLink(testSphereThighL());  
 rightKnee.setLink(testSphereLeg());  
 leftKnee.setLink(testSphereLeg());  
 rightAnkle.setLink(testSphereFootR());  
 leftAnkle.setLink(testSphereFootL());  
  
 r1.setLink(footsies());  
 r2.setLink(footsies());  
 r3.setLink(footsies());  
 r4.setLink(footsies());  
 r5.setLink(footsies());  
 r6.setLink(footsies());  
  
 l1.setLink(footsies());  
 l2.setLink(footsies());  
 l3.setLink(footsies());  
 l4.setLink(footsies());  
 l5.setLink(footsies());  
 l6.setLink(footsies());  
  
 //attach joints to each other  
 rootJoint.addJoint(rightShoulderRotator);  
 rootJoint.addJoint(leftShoulderRotator);  
 rootJoint.addJoint(leftHip);  
 rootJoint.addJoint(rightHip);  
  
 rightShoulderRotator.addJoint(rightShoulderFlapper);  
 leftShoulderRotator.addJoint(leftShoulderFlapper);  
 rightShoulderFlapper.addJoint(rightElbow);  
 leftShoulderFlapper.addJoint(leftElbow);  
 rightElbow.addJoint(rightHand);  
 leftElbow.addJoint(leftHand);  
  
 rightHip.addJoint(rightKnee);  
 leftHip.addJoint(leftKnee);  
 rightKnee.addJoint(rightAnkle);  
 leftKnee.addJoint(leftAnkle);  
  
 rightAnkle.addJoint(r1);  
 rightAnkle.addJoint(r2);  
 rightAnkle.addJoint(r3);  
 rightAnkle.addJoint(r4);  
 rightAnkle.addJoint(r5);  
 rightAnkle.addJoint(r6);  
  
 leftAnkle.addJoint(l1);  
 leftAnkle.addJoint(l2);  
 leftAnkle.addJoint(l3);  
 leftAnkle.addJoint(l4);  
 leftAnkle.addJoint(l5);  
 leftAnkle.addJoint(l6);  
  
 //sets the position of the joint from the controllers  
 q\_LRotator = leftShoulderRotator.getQ();  
 qd\_LRotator = leftShoulderRotator.getQD();  
 tau\_LRotator = leftShoulderRotator.getTau();  
  
 q\_RRotator = rightShoulderRotator.getQ();  
 qd\_RRotator = rightShoulderRotator.getQD();  
 tau\_RRotator = rightShoulderRotator.getTau();  
  
 q\_LFlapper = leftShoulderFlapper.getQ();  
 qd\_LFlapper = leftShoulderFlapper.getQD();  
 tau\_LFlapper = leftShoulderFlapper.getTau();  
  
 q\_RFlapper = rightShoulderFlapper.getQ();  
 qd\_RFlapper = rightShoulderFlapper.getQD();  
 tau\_RFlapper = rightShoulderFlapper.getTau();  
  
 q\_LElbow = leftElbow.getQ();  
 qd\_LElbow = leftElbow.getQD();  
 tau\_LElbow= leftElbow.getTau();  
  
 q\_RElbow = rightElbow.getQ();  
 qd\_RElbow = rightElbow.getQD();  
 tau\_RElbow = rightElbow.getTau();  
  
 q\_LHip = leftHip.getQ();  
 qd\_LHip = leftHip.getQD();  
 tau\_LHip = leftHip.getTau();  
  
 q\_RHip = rightHip.getQ();  
 qd\_RHip = rightHip.getQD();  
 tau\_RHip = rightHip.getTau();  
  
 q\_LKnee = leftKnee.getQ();  
 qd\_LKnee = leftKnee.getQD();  
 tau\_LKnee = leftKnee.getTau();  
  
 q\_RKnee = rightKnee.getQ();  
 qd\_RKnee = rightKnee.getQD();  
 tau\_RKnee = rightKnee.getTau();  
  
 q\_LAnkle = leftAnkle.getQ();  
 qd\_LAnkle = leftAnkle.getQD();  
 tau\_LAnkle = leftAnkle.getTau();  
  
 q\_RAnkle = rightAnkle.getQ();  
 qd\_RAnkle = rightAnkle.getQD();  
 tau\_RAnkle = rightAnkle.getTau();  
  
  
  
 //initial positions of joints  
 //rightShoulderRotator.setInitialState(fulcrumInitialPositionRadians, fulcrumInitialVelocity);  
 //rightHip.setInitialState(fulcrumInitialPositionRadians, fulcrumInitialVelocity);  
 //rightKnee.setInitialState(fulcrumInitialPositionRadians, fulcrumInitialVelocity);  
  
 rootJoint.setLink(coreGraphic());  
 this.addRootJoint(rootJoint);  
  
 //each new contact point needs a new GroundContactPoint as below  
 GroundContactPoint groundContactPointRSR = new GroundContactPoint("rightShoulderRotator", this);  
 //and it will also need to be attached to a joint or link as below  
 rightShoulderRotator.addGroundContactPoint(groundContactPointRSR);  
 //so ONE ground contact point starts here  
 GroundContactPoint groundContactPointLSR = new GroundContactPoint("leftShoulderRotator", this);  
 leftShoulderRotator.addGroundContactPoint(groundContactPointLSR);  
 //and ends here  
 GroundContactPoint groundContactPointRSF = new GroundContactPoint("rightShoulderFlapper", this);  
 rightShoulderFlapper.addGroundContactPoint(groundContactPointRSF);  
  
 GroundContactPoint groundContactPointLSF = new GroundContactPoint("leftShoulderFlapper", this);  
 leftShoulderFlapper.addGroundContactPoint(groundContactPointLSF);  
  
 GroundContactPoint groundContactPointRE = new GroundContactPoint("rightElbow", this);  
 rightElbow.addGroundContactPoint(groundContactPointRE);  
  
 GroundContactPoint groundContactPointLE = new GroundContactPoint("leftElbow", this);  
 leftElbow.addGroundContactPoint(groundContactPointLE);  
  
 GroundContactPoint groundContactPointRHn = new GroundContactPoint("rightHand", this);  
 rightHand.addGroundContactPoint(groundContactPointRHn);  
  
 GroundContactPoint groundContactPointLHn = new GroundContactPoint("leftHand", this);  
 leftHand.addGroundContactPoint(groundContactPointLHn);  
  
  
 GroundContactPoint groundContactPointRH = new GroundContactPoint("rightHip", this);  
 rightHip.addGroundContactPoint(groundContactPointRH);  
  
 GroundContactPoint groundContactPointLH = new GroundContactPoint("leftHip", this);  
 leftHip.addGroundContactPoint(groundContactPointLH);  
  
 GroundContactPoint groundContactPointRK = new GroundContactPoint("rightKnee", this);  
 rightKnee.addGroundContactPoint(groundContactPointRK);  
  
 GroundContactPoint groundContactPointLK = new GroundContactPoint("leftKnee", this);  
 leftKnee.addGroundContactPoint(groundContactPointLK);  
  
 GroundContactPoint groundContactPointRA = new GroundContactPoint("rightAnkle", this);  
 rightAnkle.addGroundContactPoint(groundContactPointRA);  
  
 GroundContactPoint groundContactPointLA = new GroundContactPoint("leftAnkle", this);  
 leftAnkle.addGroundContactPoint(groundContactPointLA);  
  
 GroundContactPoint groundContactPointR1 = new GroundContactPoint("r1", this);  
 r1.addGroundContactPoint(groundContactPointR1);  
 GroundContactPoint groundContactPointR2 = new GroundContactPoint("r2", this);  
 r2.addGroundContactPoint(groundContactPointR2);  
 GroundContactPoint groundContactPointR3 = new GroundContactPoint("r3", this);  
 r3.addGroundContactPoint(groundContactPointR3);  
 GroundContactPoint groundContactPointR4 = new GroundContactPoint("r4", this);  
 r4.addGroundContactPoint(groundContactPointR4);  
 GroundContactPoint groundContactPointR5 = new GroundContactPoint("r5", this);  
 r5.addGroundContactPoint(groundContactPointR5);  
 GroundContactPoint groundContactPointR6 = new GroundContactPoint("r6", this);  
 r6.addGroundContactPoint(groundContactPointR6);  
  
 GroundContactPoint groundContactPointL1 = new GroundContactPoint("l1", this);  
 l1.addGroundContactPoint(groundContactPointL1);  
 GroundContactPoint groundContactPointL2 = new GroundContactPoint("l2", this);  
 l2.addGroundContactPoint(groundContactPointL2);  
 GroundContactPoint groundContactPointL3 = new GroundContactPoint("l3", this);  
 l3.addGroundContactPoint(groundContactPointL3);  
 GroundContactPoint groundContactPointL4 = new GroundContactPoint("l4", this);  
 l4.addGroundContactPoint(groundContactPointL4);  
 GroundContactPoint groundContactPointL5 = new GroundContactPoint("l5", this);  
 l5.addGroundContactPoint(groundContactPointL5);  
 GroundContactPoint groundContactPointL6 = new GroundContactPoint("l6", this);  
 l6.addGroundContactPoint(groundContactPointL6);  
  
 //This creates a flat surface for JD to stand on  
 GroundContactModel groundModel = new LinearGroundContactModel(this, 1422, 150.6, 50.0, 1000.0,  
 this.getRobotsYoVariableRegistry());  
 GroundProfile3D profile = new FlatGroundProfile();  
 groundModel.setGroundProfile3D(profile);  
 this.setGroundContactModel(groundModel);  
 }  
  
 */\*\*  
 \* Fulcrum's angular position in radians  
 \** ***@return*** *angular position in radians  
 \*/* public double getLRotatorAngularPosition() { return q\_LRotator.getDoubleValue(); }  
 public double getRRotatorAngularPosition() { return q\_RRotator.getDoubleValue(); }  
 public double getLFlapperAngularPosition()  
 {  
 return q\_LFlapper.getDoubleValue();  
 }  
 public double getRFlapperAngularPosition() { return q\_RFlapper.getDoubleValue(); }  
 public double getLElbowAngularPosition()  
 {  
 return q\_LElbow.getDoubleValue();  
 }  
 public double getRElbowAngularPosition() { return q\_RElbow.getDoubleValue(); }  
 public double getLHipAngularPosition()  
 {  
 return q\_LHip.getDoubleValue();  
 }  
 public double getRHipAngularPosition()  
 {  
 return q\_RHip.getDoubleValue();  
 }  
 public double getLKneeAngularPosition()  
 {  
 return q\_LKnee.getDoubleValue();  
 }  
 public double getRKneeAngularPosition()  
 {  
 return q\_RKnee.getDoubleValue();  
 }  
 public double getLAnkleAngularPosition()  
 {  
 return q\_LAnkle.getDoubleValue();  
 }  
 public double getRAnkleAngularPosition()  
 {  
 return q\_RAnkle.getDoubleValue();  
 }  
  
  
 */\*\*  
 \* Fulcrum's angular velocity in radians per seconds  
 \** ***@return*** *angular velocity in radians per seconds  
 \*/* public double getLRotatorAngularVelocity() { return qd\_LRotator.getDoubleValue(); }  
 public double getRRotatorAngularVelocity() { return qd\_RRotator.getDoubleValue(); }  
 public double getLFlapperAngularVelocity()  
 {  
 return qd\_LFlapper.getDoubleValue();  
 }  
 public double getRFlapperAngularVelocity() { return qd\_RFlapper.getDoubleValue(); }  
 public double getLElbowAngularVelocity()  
 {  
 return qd\_LElbow.getDoubleValue();  
 }  
 public double getRElbowAngularVelocity() { return qd\_RElbow.getDoubleValue(); }  
 public double getLHipAngularVelocity()  
 {  
 return qd\_LHip.getDoubleValue();  
 }  
 public double getRHipAngularVelocity()  
 {  
 return qd\_RHip.getDoubleValue();  
 }  
 public double getLKneeAngularVelocity()  
 {  
 return qd\_LKnee.getDoubleValue();  
 }  
 public double getRKneeAngularVelocity()  
 {  
 return qd\_RKnee.getDoubleValue();  
 }  
 public double getLAnkleAngularVelocity()  
 {  
 return qd\_LAnkle.getDoubleValue();  
 }  
 public double getRAnkleAngularVelocity()  
 {  
 return qd\_RAnkle.getDoubleValue();  
 }  
 */\*\*  
 \* Fulcrum's torque in Newton meter  
 \** ***@return*** *Torque in Newton meter  
 \*/  
 /\*\*  
 \* Set Fulcrum's torque in Newton meter  
 \** ***@return*** *Torque in Newton meter  
 \*/* public void setLRotatorTorque(double tau)  
 {  
 this.tau\_LRotator.set(tau);  
 }  
 public void setRRotatorTorque(double tau)  
 {  
 this.tau\_RRotator.set(tau);  
 }  
 public void setLFlapperTorque(double tau) { this.tau\_LFlapper.set(tau); }  
 public void setRFlapperTorque(double tau)  
 {  
 this.tau\_RFlapper.set(tau);  
 }  
 public void setLElbowTorque(double tau)  
 {  
 this.tau\_LElbow.set(tau);  
 }  
 public void setRElbowTorque(double tau)  
 {  
 this.tau\_RElbow.set(tau);  
 }  
 public void setLHipTorque(double tau)  
 {  
 this.tau\_LHip.set(tau);  
 }  
 public void setRHipTorque(double tau)  
 {  
 this.tau\_RHip.set(tau);  
 }  
 public void setLKneeTorque(double tau)  
 {  
 this.tau\_LKnee.set(tau);  
 }  
 public void setRKneeTorque(double tau)  
 {  
 this.tau\_RKnee.set(tau);  
 }  
 public void setLAnkleTorque(double tau)  
 {  
 this.tau\_LAnkle.set(tau);  
 }  
 public void setRAnkleTorque(double tau)  
 {  
 this.tau\_RAnkle.set(tau);  
 }  
  
 //many many graphics methods  
  
 //used for the shoulder rotators, which are not visible  
 private Link servoPinAxisGraphicL()  
 {  
 Link servo = new Link("servoPin");  
 servo.setMomentOfInertia(*FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*, *FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*, *FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*);  
 servo.setMass(*BALL\_MASS*);  
  
 Graphics3DObject servoHeadGraphics = new Graphics3DObject();  
  
 servoHeadGraphics.rotate((Math.*PI*/2), Axis.*Y*);  
 servoHeadGraphics.translate(0.0, 0.0, 0.0);//0.0835 is one half of .167(the cylinders height) setting this value in the x pos negative centers the graphic on the center of the virtual object  
 servoHeadGraphics.addCylinder(.6, .15, YoAppearance.*Black*());  
 servo.setLinkGraphics(servoHeadGraphics);  
  
 return servo;  
 }  
  
 private Link servoPinAxisGraphicR()  
 {  
 Link servo = new Link("servoPin");  
 servo.setMomentOfInertia(*FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*, *FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*, *FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*);  
 servo.setMass(*BALL\_MASS*);  
  
 Graphics3DObject servoHeadGraphics = new Graphics3DObject();  
  
 servoHeadGraphics.rotate((Math.*PI*/2), Axis.*Y*);  
 servoHeadGraphics.translate(0.0, 0.0, -.5);//0.0835 is one half of .167(the cylinders height) setting this value in the x pos negative centers the graphic on the center of the virtual object  
 servoHeadGraphics.addCylinder(.6, .15, YoAppearance.*Black*());  
 servo.setLinkGraphics(servoHeadGraphics);  
  
 return servo;  
 }  
  
 //this is the graphic stuff for the body and head, which is all attached to the root joint, and JD's head is indeed  
 //fake. It's just a graphic  
 private Link coreGraphic()  
 {  
 Link body = new Link("body");  
 body.setMomentOfInertia(*FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*, *FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*, *FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*);  
 body.setMass(*BALL\_MASS*);  
  
 Graphics3DObject bodyGraphics = new Graphics3DObject();  
 bodyGraphics.addCube(3.75\**INCH\_TO\_MILLIMETER*, *SERVO\_JOINT\_LENGTH*, 2.625\**INCH\_TO\_MILLIMETER*, YoAppearance.*White*());//x=width y=depth z=height looking at the robot  
 bodyGraphics.rotate((Math.*PI*/2), Axis.*Z*);  
 bodyGraphics.translate(0.0, 0.875\**INCH\_TO\_MILLIMETER*, -.5);  
 bodyGraphics.addCylinder(.5, .13, YoAppearance.*Black*());  
 bodyGraphics.translate(0.0, -0.875\**INCH\_TO\_MILLIMETER*\*2, 0.0);  
 bodyGraphics.addCylinder(.5, .13, YoAppearance.*Black*());  
 bodyGraphics.translate(0.0, 0.875\**INCH\_TO\_MILLIMETER*, 0.5);  
 bodyGraphics.translate(0.0, 0.0, 1);  
 bodyGraphics.addCylinder(.5, .13, YoAppearance.*Black*());  
 //changing this translate affects the head  
 bodyGraphics.translate(0.0, 0.0, 1);  
 bodyGraphics.addSphere(.6, YoAppearance.*White*());//the actual head  
 bodyGraphics.rotate((Math.*PI*\*0.2), Axis.*Z*);  
 bodyGraphics.rotate((Math.*PI*\*0.2), Axis.*Y*);  
 bodyGraphics.translate(0.0, 0.05, .44);  
 bodyGraphics.addCylinder(.7, .4, YoAppearance.*Black*());  
 bodyGraphics.addCylinder(.05, .7, YoAppearance.*Black*());  
 body.setLinkGraphics(bodyGraphics);  
  
 return body;  
 }  
  
 private Link testSphereThighR()  
 {  
 Link servo = new Link("servoPin");  
 servo.setMomentOfInertia(*FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*, *FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*, *FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*);  
 servo.setMass(*BALL\_MASS*);  
  
 Graphics3DObject servoHeadGraphics = new Graphics3DObject();  
  
 servoHeadGraphics.addSphere(0.25, YoAppearance.*White*());  
 servoHeadGraphics.translate(0.0, 0.0, -2.375\**INCH\_TO\_MILLIMETER*);  
 servoHeadGraphics.addCylinder(2.375\**INCH\_TO\_MILLIMETER*, 0.12, YoAppearance.*Black*());  
 servoHeadGraphics.addSphere(.2, YoAppearance.*Chartreuse*());  
 servo.setLinkGraphics(servoHeadGraphics);  
  
 return servo;  
 }  
  
 private Link testSphereThighL()  
 {  
 Link servo = new Link("servoPin");  
 servo.setMomentOfInertia(*FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*, *FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*, *FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*);  
 servo.setMass(*BALL\_MASS*);  
  
 Graphics3DObject servoHeadGraphics = new Graphics3DObject();  
  
 servoHeadGraphics.addSphere(0.25, YoAppearance.*White*());  
 //adding the groin bar  
 servoHeadGraphics.rotate((Math.*PI*/2), Axis.*Y*);  
 servoHeadGraphics.addCylinder(.6, 0.12, YoAppearance.*Black*());  
 servoHeadGraphics.rotate(-(Math.*PI*/2), Axis.*Y*);  
 servoHeadGraphics.translate(0.0, 0.0, -2.375\**INCH\_TO\_MILLIMETER*);  
 servoHeadGraphics.addCylinder(2.375\**INCH\_TO\_MILLIMETER*, 0.12, YoAppearance.*Black*());  
 servoHeadGraphics.addSphere(.2, YoAppearance.*Chartreuse*());  
 servo.setLinkGraphics(servoHeadGraphics);  
  
 return servo;  
 }  
  
 //we only needed one of these because the legs were similar enough  
 private Link testSphereLeg()  
 {  
 Link servo = new Link("servoPin");  
 servo.setMomentOfInertia(*FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*, *FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*, *FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*);  
 servo.setMass(*BALL\_MASS*);  
  
 Graphics3DObject servoHeadGraphics = new Graphics3DObject();  
  
 servoHeadGraphics.addSphere(.27, YoAppearance.*White*());  
  
 servoHeadGraphics.translate(0.0, 0.0, -2.125\**INCH\_TO\_MILLIMETER*);  
 servoHeadGraphics.addCylinder(2.125\**INCH\_TO\_MILLIMETER*, 0.12, YoAppearance.*Black*());  
 servoHeadGraphics.addSphere(.2, YoAppearance.*Chartreuse*());  
 servo.setLinkGraphics(servoHeadGraphics);  
  
 return servo;  
 }  
  
 private Link testSphereFootR()  
 {  
 Link servo = new Link("servoPin");  
 servo.setMomentOfInertia(*FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*, *FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*, *FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*);  
 servo.setMass(*BALL\_MASS*);  
  
 Graphics3DObject servoHeadGraphics = new Graphics3DObject();  
  
 servoHeadGraphics.addSphere(.25, YoAppearance.*Gray*());  
 servoHeadGraphics.translate(0.19, -0.22, -0.625\**INCH\_TO\_MILLIMETER*);  
 servoHeadGraphics.addHemiEllipsoid(0.55, 0.85, 0.4, YoAppearance.*White*());  
 servo.setLinkGraphics(servoHeadGraphics);  
  
 return servo;  
 }  
  
 private Link testSphereFootL()  
 {  
 Link servo = new Link("servoPin");  
 servo.setMomentOfInertia(*FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*, *FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*, *FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*);  
 servo.setMass(*BALL\_MASS*);  
  
 Graphics3DObject servoHeadGraphics = new Graphics3DObject();  
  
 servoHeadGraphics.addSphere(.25, YoAppearance.*Gray*());  
 servoHeadGraphics.translate(-0.19, -0.22, -0.625\**INCH\_TO\_MILLIMETER*);  
 servoHeadGraphics.addHemiEllipsoid(.55, .85, .4, YoAppearance.*White*());  
 servo.setLinkGraphics(servoHeadGraphics);  
  
 return servo;  
 }  
  
 //these are the upper arms. We are... bad at names  
 private Link testSphereArmThighL()  
 {  
 Link servo = new Link("servoPin");  
 servo.setMomentOfInertia(*FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*, *FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*, *FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*);  
 servo.setMass(*BALL\_MASS*);  
  
 Graphics3DObject servoHeadGraphics = new Graphics3DObject();  
  
 servoHeadGraphics.rotate((Math.*PI*/2), Axis.*Y*);  
 servoHeadGraphics.addSphere(.4, YoAppearance.*White*());  
 servoHeadGraphics.translate(0.0, 0.0, -2.75\**INCH\_TO\_MILLIMETER*);  
 servoHeadGraphics.addCylinder(2.75\**INCH\_TO\_MILLIMETER*, .2, YoAppearance.*Black*());  
 servoHeadGraphics.addSphere(*BALL\_RADIUS*, YoAppearance.*Chartreuse*());  
 servo.setLinkGraphics(servoHeadGraphics);  
  
 return servo;  
 }  
  
 private Link testSphereArmThighR()  
 {  
 Link servo = new Link("servoPin");  
 servo.setMomentOfInertia(*FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*, *FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*, *FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*);  
 servo.setMass(*BALL\_MASS*);  
  
 Graphics3DObject servoHeadGraphics = new Graphics3DObject();  
  
 servoHeadGraphics.rotate((Math.*PI*/2), Axis.*Y*);  
 servoHeadGraphics.addSphere(.4, YoAppearance.*White*());  
 servoHeadGraphics.translate(0.0, 0.0, 0);  
 servoHeadGraphics.addCylinder(2.75\**INCH\_TO\_MILLIMETER*, .2, YoAppearance.*Black*());  
 servoHeadGraphics.addSphere(*BALL\_RADIUS*, YoAppearance.*Chartreuse*());  
 servo.setLinkGraphics(servoHeadGraphics);  
  
 return servo;  
 }  
  
 private Link testSphereForearmR()  
 {  
 Link servo = new Link("servoPin");  
 servo.setMomentOfInertia(*FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*, *FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*, *FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*);  
 servo.setMass(*BALL\_MASS*);  
  
 Graphics3DObject servoHeadGraphics = new Graphics3DObject();  
  
 servoHeadGraphics.rotate((Math.*PI*/2), Axis.*Y*);  
 servoHeadGraphics.addSphere(.3, YoAppearance.*White*());  
 servoHeadGraphics.translate(0.0, 0.0, 0.0);  
 servoHeadGraphics.addCylinder(2.75\**INCH\_TO\_MILLIMETER*, .2, YoAppearance.*Black*());  
 servoHeadGraphics.addSphere(*BALL\_RADIUS*, YoAppearance.*Chartreuse*());  
 servo.setLinkGraphics(servoHeadGraphics);  
  
 return servo;  
 }  
  
 private Link testSphereForearmL()  
 {  
 Link servo = new Link("servoPin");  
 servo.setMomentOfInertia(*FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*, *FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*, *FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*);  
 servo.setMass(*BALL\_MASS*);  
  
 Graphics3DObject servoHeadGraphics = new Graphics3DObject();  
  
 servoHeadGraphics.rotate((Math.*PI*/2), Axis.*Y*);  
 servoHeadGraphics.addSphere(.3, YoAppearance.*White*());  
 servoHeadGraphics.translate(0.0, 0.0, -2.75\**INCH\_TO\_MILLIMETER*);  
 servoHeadGraphics.addCylinder(2.75\**INCH\_TO\_MILLIMETER*, .2, YoAppearance.*Black*());  
 servoHeadGraphics.addSphere(*BALL\_RADIUS*, YoAppearance.*Chartreuse*());  
 servo.setLinkGraphics(servoHeadGraphics);  
  
 return servo;  
 }  
  
 private Link testSphereHand()  
 {  
 Link servo = new Link("servoPin");  
 servo.setMomentOfInertia(*FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*, *FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*, *FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*);  
 servo.setMass(*BALL\_MASS*);  
  
 Graphics3DObject servoHeadGraphics = new Graphics3DObject();  
  
 servoHeadGraphics.addSphere(.32, YoAppearance.*White*());  
 servoHeadGraphics.addCylinder(.05, .4, YoAppearance.*Black*());  
 servo.setLinkGraphics(servoHeadGraphics);  
  
 return servo;  
 }  
  
 //these are invisible joints to create contact points all around the feet. Uncomment the section within to see  
 //where they actually are.  
 private Link footsies()  
 {  
 Link servo = new Link("servoPin");  
 servo.setMomentOfInertia(*FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*, *FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*, *FULCRUM\_MOMENT\_OF\_INERTIA\_ABOUT\_X*);  
 servo.setMass(.1);  
  
// Graphics3DObject servoHeadGraphics = new Graphics3DObject();  
//  
// servoHeadGraphics.addSphere(.1, YoAppearance.White());  
// servoHeadGraphics.addCylinder(.05, .4, YoAppearance.Black());  
// servo.setLinkGraphics(servoHeadGraphics);  
  
 return servo;  
 }  
}

Yeah…

Now we need to talk about root joints. Root joints are the initial joint of your robot. Every robot has one, and only one. The first Pendulum of WOOWOO acted as its root. Here the root joint is invisible, and it exists at the bottom center of JD’s rectangle. New joints can be built off of the root, but they can never link back in from another location. This means that joints will always branch out. In our case it looks something like this:



Where most of the other joints have pendulum-like graphic segments attached to them, the root actually has the graphic for the body, head, and hat all attached to it.

You should also be aware that you can put a root joint in a planar. This makes the root sit in a two-dimensional field instead of a three-dimensional one. You can use this to keep your robot stable if you need it, but that’s cheating and JD is no cheater (JK use it if you need it).

Now let’s look at ground contact points:

//each new contact point needs a new GroundContactPoint as below  
GroundContactPoint groundContactPointRSR = new GroundContactPoint("rightShoulderRotator", this);  
//and it will also need to be attached to a joint or link as below  
rightShoulderRotator.addGroundContactPoint(groundContactPointRSR);  
//so ONE ground contact point starts here  
GroundContactPoint groundContactPointLSR = new GroundContactPoint("leftShoulderRotator", this);  
leftShoulderRotator.addGroundContactPoint(groundContactPointLSR);  
//and ends here  
GroundContactPoint groundContactPointRSF = new GroundContactPoint("rightShoulderFlapper", this);  
rightShoulderFlapper.addGroundContactPoint(groundContactPointRSF);  
  
GroundContactPoint groundContactPointLSF = new GroundContactPoint("leftShoulderFlapper", this);  
leftShoulderFlapper.addGroundContactPoint(groundContactPointLSF);  
  
GroundContactPoint groundContactPointRE = new GroundContactPoint("rightElbow", this);  
rightElbow.addGroundContactPoint(groundContactPointRE);  
  
GroundContactPoint groundContactPointLE = new GroundContactPoint("leftElbow", this);  
leftElbow.addGroundContactPoint(groundContactPointLE);  
  
GroundContactPoint groundContactPointRHn = new GroundContactPoint("rightHand", this);  
rightHand.addGroundContactPoint(groundContactPointRHn);  
  
GroundContactPoint groundContactPointLHn = new GroundContactPoint("leftHand", this);  
leftHand.addGroundContactPoint(groundContactPointLHn);  
  
  
GroundContactPoint groundContactPointRH = new GroundContactPoint("rightHip", this);  
rightHip.addGroundContactPoint(groundContactPointRH);  
  
GroundContactPoint groundContactPointLH = new GroundContactPoint("leftHip", this);  
leftHip.addGroundContactPoint(groundContactPointLH);  
  
GroundContactPoint groundContactPointRK = new GroundContactPoint("rightKnee", this);  
rightKnee.addGroundContactPoint(groundContactPointRK);  
  
GroundContactPoint groundContactPointLK = new GroundContactPoint("leftKnee", this);  
leftKnee.addGroundContactPoint(groundContactPointLK);  
  
GroundContactPoint groundContactPointRA = new GroundContactPoint("rightAnkle", this);  
rightAnkle.addGroundContactPoint(groundContactPointRA);  
  
GroundContactPoint groundContactPointLA = new GroundContactPoint("leftAnkle", this);  
leftAnkle.addGroundContactPoint(groundContactPointLA);  
  
GroundContactPoint groundContactPointR1 = new GroundContactPoint("r1", this);  
r1.addGroundContactPoint(groundContactPointR1);  
GroundContactPoint groundContactPointR2 = new GroundContactPoint("r2", this);  
r2.addGroundContactPoint(groundContactPointR2);  
GroundContactPoint groundContactPointR3 = new GroundContactPoint("r3", this);  
r3.addGroundContactPoint(groundContactPointR3);  
GroundContactPoint groundContactPointR4 = new GroundContactPoint("r4", this);  
r4.addGroundContactPoint(groundContactPointR4);  
GroundContactPoint groundContactPointR5 = new GroundContactPoint("r5", this);  
r5.addGroundContactPoint(groundContactPointR5);  
GroundContactPoint groundContactPointR6 = new GroundContactPoint("r6", this);  
r6.addGroundContactPoint(groundContactPointR6);  
  
GroundContactPoint groundContactPointL1 = new GroundContactPoint("l1", this);  
l1.addGroundContactPoint(groundContactPointL1);  
GroundContactPoint groundContactPointL2 = new GroundContactPoint("l2", this);  
l2.addGroundContactPoint(groundContactPointL2);  
GroundContactPoint groundContactPointL3 = new GroundContactPoint("l3", this);  
l3.addGroundContactPoint(groundContactPointL3);  
GroundContactPoint groundContactPointL4 = new GroundContactPoint("l4", this);  
l4.addGroundContactPoint(groundContactPointL4);  
GroundContactPoint groundContactPointL5 = new GroundContactPoint("l5", this);  
l5.addGroundContactPoint(groundContactPointL5);  
GroundContactPoint groundContactPointL6 = new GroundContactPoint("l6", this);  
l6.addGroundContactPoint(groundContactPointL6);  
  
//This creates a flat surface for JD to stand on  
GroundContactModel groundModel = new LinearGroundContactModel(this, 1422, 150.6, 50.0, 1000.0,  
 this.getRobotsYoVariableRegistry());  
GroundProfile3D profile = new FlatGroundProfile();  
groundModel.setGroundProfile3D(profile);  
this.setGroundContactModel(groundModel);

This was added in at the bottom of the constructor. They are fairly straightforward. The bit at the very bottom creates the flat-ground surface JD stands on.

In WOOWOOBot we only used one method to apply graphics. We could do that because of the simplicity of the robot. That won’t be the case here, however. Because many of the pieces have to be flipped symmetrically, the numbers determining their placement must also be flipped. We solved this by creating a new method for each joint, though there are more clever ways this could be accomplished.

You may notice joints labeled r1-r6 and l1-l6. These are invisible and exist around the perimeter of JD’s feet. They act as the ground contacts that keep JD from falling over.

I think that covers ArmRobot fairly well. Let’s move on to ArmController:

package us.ihmc.exampleSimulations.JD;  
  
import us.ihmc.robotics.dataStructures.registry.YoVariableRegistry;  
import us.ihmc.robotics.dataStructures.variable.DoubleYoVariable;  
import us.ihmc.robotics.dataStructures.variable.IntegerYoVariable;  
import us.ihmc.simulationconstructionset.robotController.RobotController;  
  
  
public class ArmController implements RobotController  
{  
 // A name for this controller  
 private final String name = "pendulumController";  
  
 // This line instantiates a registry that will contain relevant controller variables that will be accessible from the simulation panel.  
 private final YoVariableRegistry registry = new YoVariableRegistry("RobotController");  
  
 // This is a reference to the SimplePendulumRobot that enables the controller to access this robot's variables.  
 private ArmRobot robot;  
  
 /\* Control variables: \*/  
  
 // Target angle  
 private DoubleYoVariable desiredPositionRadians;  
  
 // Controller parameter variables  
 private DoubleYoVariable p\_gain, d\_gain, i\_gain;  
  
 private IntegerYoVariable RRotUp;  
 private IntegerYoVariable LRotUp;  
 private IntegerYoVariable crouch;  
  
 private int iR = 0;  
 private int iL = 0;  
 private DoubleYoVariable iLCrouchHip;  
 private DoubleYoVariable iRCrouchHip;  
 private DoubleYoVariable iLCrouchKnee;  
 private DoubleYoVariable iRCrouchKnee;  
  
  
 private double theNumberWeAreCountingUpTo = 5000;  
  
 // This is the desired torque that we will apply to the fulcrum joint (PinJoint)  
 private double torque;  
  
 /\* Constructor:  
 Where we instantiate and initialize control variables  
 \*/  
 public ArmController(ArmRobot robot)  
 {  
 this.robot = robot;  
 desiredPositionRadians = new DoubleYoVariable("DesiredPosRad", registry);  
 desiredPositionRadians.set(Math.*PI*);  
  
 p\_gain = new DoubleYoVariable("ProportionalGain", registry);  
 p\_gain.set(250.0);  
 d\_gain = new DoubleYoVariable("DerivativeGain", registry);  
 d\_gain.set(100.0);  
 i\_gain = new DoubleYoVariable("IntegralGain", registry);  
 i\_gain.set(10.0);  
  
 //we created these so you can control JD during simulation  
 RRotUp = new IntegerYoVariable("RRotUp", registry);  
 RRotUp.set(0);  
 LRotUp = new IntegerYoVariable("LRotUp", registry);  
 LRotUp.set(0);  
 crouch = new IntegerYoVariable("crouch", registry);  
 crouch.set(0);  
  
 iLCrouchHip = new DoubleYoVariable("iLCrouchHip", registry);  
 iLCrouchHip.set(1);  
 iRCrouchHip = new DoubleYoVariable("iRCrouchHip", registry);  
 iRCrouchHip.set(1);  
 iLCrouchKnee = new DoubleYoVariable("iLCrouchKnee", registry);  
 iLCrouchKnee.set(1);  
 iRCrouchKnee = new DoubleYoVariable("iRCrouchKnee", registry);  
 iRCrouchKnee.set(1);  
 }  
  
 public void initialize()  
 {  
  
 }  
  
 private double positionError = 0;  
 private double integralError = 0;  
  
 //doControl() the all-powerful  
 public void doControl()  
 {  
 LRotatorController();  
 RRotatorController();  
 LFlapperController();  
 RFlapperController();  
 LElbowController();  
 RElbowController();  
 LHipController();  
 RHipController();  
 LKneeController();  
 RKneeController();  
 LAnkleController();  
 RAnkleController();  
 }  
  
 //these are the shoulder rotator joints  
 public void LRotatorController()  
 {  
 // ERROR term: Compute the difference between the desired position the pendulum and its current position  
 if(LRotUp.getIntegerValue() != 0)  
 {  
 positionError = (0) - robot.getLRotatorAngularPosition();  
 }  
 else  
 {  
 positionError = (desiredPositionRadians.getDoubleValue()) - robot.getLRotatorAngularPosition();  
 }  
 // INTEGRAL term: Compute a simple numerical integration of the position error  
 integralError += positionError \* ArmSimulation.*DT*; //  
  
 // P.I.D  
 torque = p\_gain.getDoubleValue() \* positionError +  
 i\_gain.getDoubleValue() \* integralError +  
 d\_gain.getDoubleValue() \* (0 - robot.getLRotatorAngularVelocity());  
  
 robot.setLRotatorTorque(torque);  
 }  
 public void RRotatorController()  
 {  
 // ERROR term: Compute the difference between the desired position the pendulum and its current position  
 if(RRotUp.getIntegerValue() != 0)  
 {  
 positionError = -1\*(desiredPositionRadians.getDoubleValue()) - robot.getRRotatorAngularPosition();  
 }  
 else  
 {  
 positionError = (0) - robot.getRRotatorAngularPosition();  
 }  
  
 // INTEGRAL term: Compute a simple numerical integration of the position error  
 integralError += positionError \* ArmSimulation.*DT*; //  
  
 // P.I.D  
 torque = p\_gain.getDoubleValue() \* positionError +  
 i\_gain.getDoubleValue() \* integralError +  
 d\_gain.getDoubleValue() \* (0 - robot.getRRotatorAngularVelocity());  
  
 robot.setRRotatorTorque(torque);  
 }  
  
 //the second joints on the shoulders, used to simulate a ball joint  
 public void LFlapperController()  
 {  
 // ERROR term: Compute the difference between the desired position the pendulum and its current position  
 positionError = (desiredPositionRadians.getDoubleValue() \* .25) - robot.getLFlapperAngularPosition();  
  
 // INTEGRAL term: Compute a simple numerical integration of the position error  
 integralError += positionError \* ArmSimulation.*DT*; //  
  
 // P.I.D  
 torque = p\_gain.getDoubleValue() \* positionError +  
 i\_gain.getDoubleValue() \* integralError +  
 d\_gain.getDoubleValue() \* (0 - robot.getLFlapperAngularVelocity());  
  
 robot.setLFlapperTorque(torque);  
 }  
 public void RFlapperController()  
 {  
 // ERROR term: Compute the difference between the desired position the pendulum and its current position  
 positionError = (desiredPositionRadians.getDoubleValue() \* .25) - robot.getRFlapperAngularPosition();  
  
 // INTEGRAL term: Compute a simple numerical integration of the position error  
 integralError += positionError \* ArmSimulation.*DT*; //  
  
 // P.I.D  
 torque = p\_gain.getDoubleValue() \* positionError +  
 i\_gain.getDoubleValue() \* integralError +  
 d\_gain.getDoubleValue() \* (0 - robot.getRFlapperAngularVelocity());  
  
 robot.setRFlapperTorque(torque);  
 }  
  
  
 public void LElbowController()  
 {  
 // ERROR term: Compute the difference between the desired position the pendulum and its current position  
 if(LRotUp.getIntegerValue() != 0)  
 {  
 if (iL == 1000)  
 {  
 iL = 2000;  
 }  
 else if (iL == 1001)  
 {  
 iL = 0;  
 }  
 if (iL > 1000)  
 {  
  
 positionError = (desiredPositionRadians.getDoubleValue()\*0.25) - (robot.getLElbowAngularPosition());  
 iL--;  
 }  
 if (iL < 1000)  
 {  
 positionError = (desiredPositionRadians.getDoubleValue()\*0.5) - (robot.getLElbowAngularPosition());  
 iL++;  
 }  
 }  
 else  
 {  
 positionError = (desiredPositionRadians.getDoubleValue() \* 0.25) - robot.getLElbowAngularPosition();  
 }  
  
 // INTEGRAL term: Compute a simple numerical integration of the position error  
 integralError += positionError \* ArmSimulation.*DT*; //  
  
 // P.I.D  
 torque = p\_gain.getDoubleValue() \* positionError +  
 i\_gain.getDoubleValue() \* integralError +  
 d\_gain.getDoubleValue() \* (0 - robot.getLElbowAngularVelocity());  
  
 robot.setLElbowTorque(torque);  
 }  
 public void RElbowController()  
 {  
 // ERROR term: Compute the difference between the desired position the pendulum and its current position  
 if(RRotUp.getIntegerValue() != 0)  
 {  
 if (iR == 1000)  
 {  
 iR = 2000;  
 }  
 else if (iR == 1001)  
 {  
 iR = 0;  
 }  
 if (iR > 1000)  
 {  
  
 positionError = (desiredPositionRadians.getDoubleValue()\*.25) - (robot.getRElbowAngularPosition());  
 iR--;  
 }  
 if (iR < 1000)  
 {  
 positionError = (desiredPositionRadians.getDoubleValue()\*0.5) - (robot.getRElbowAngularPosition());  
 iR++;  
 }  
 }  
 else  
 {  
 positionError = (desiredPositionRadians.getDoubleValue() \* 0.25) - robot.getRElbowAngularPosition();  
 }  
  
 // INTEGRAL term: Compute a simple numerical integration of the position error  
 integralError += positionError \* ArmSimulation.*DT*; //  
  
 // P.I.D  
 torque = p\_gain.getDoubleValue() \* positionError +  
 i\_gain.getDoubleValue() \* integralError +  
 d\_gain.getDoubleValue() \* (0 - robot.getRElbowAngularVelocity());  
  
 robot.setRElbowTorque(torque);  
 }  
  
  
 public void LHipController()  
 {  
 // ERROR term: Compute the difference between the desired position the pendulum and its current position  
 if(crouch.getIntegerValue() != 0)  
 {  
 if(iLCrouchHip.getDoubleValue() < theNumberWeAreCountingUpTo)  
 {  
 positionError = -(((desiredPositionRadians.getDoubleValue())\*(iLCrouchHip.getDoubleValue()/theNumberWeAreCountingUpTo))\*0.75) - robot.getLHipAngularPosition();  
 iLCrouchHip.add(1);  
 }  
 else  
 {  
 positionError = -(desiredPositionRadians.getDoubleValue()\*0.75) - robot.getLHipAngularPosition();  
 }  
 }  
 else  
 {  
 positionError = (desiredPositionRadians.getDoubleValue()\*0) - robot.getLHipAngularPosition();  
 iLCrouchHip.set(0);  
 }  
  
 // INTEGRAL term: Compute a simple numerical integration of the position error  
 integralError += positionError \* ArmSimulation.*DT*; //  
  
 // P.I.D  
 torque = p\_gain.getDoubleValue() \* positionError +  
 i\_gain.getDoubleValue() \* integralError +  
 d\_gain.getDoubleValue() \* (0 - robot.getLHipAngularVelocity());  
  
 robot.setLHipTorque(torque);  
 }  
 public void RHipController()  
 {  
 // ERROR term: Compute the difference between the desired position the pendulum and its current position  
 if(crouch.getIntegerValue() != 0)  
 {  
 if(iRCrouchHip.getDoubleValue() < theNumberWeAreCountingUpTo)  
 {  
 positionError = -(((desiredPositionRadians.getDoubleValue())\*(iRCrouchHip.getDoubleValue()/theNumberWeAreCountingUpTo))\*0.75) - robot.getRHipAngularPosition();  
 iRCrouchHip.add(1);  
 }  
 else  
 {  
 positionError = -(desiredPositionRadians.getDoubleValue()\*0.75) - robot.getRHipAngularPosition();  
  
 }  
 }  
 else  
 {  
 positionError = (desiredPositionRadians.getDoubleValue()\*0) - robot.getRHipAngularPosition();  
 iRCrouchHip.set(0);  
 }  
  
 // INTEGRAL term: Compute a simple numerical integration of the position error  
 integralError += positionError \* ArmSimulation.*DT*; //  
  
 // P.I.D  
 torque = p\_gain.getDoubleValue() \* positionError +  
 i\_gain.getDoubleValue() \* integralError +  
 d\_gain.getDoubleValue() \* (0 - robot.getRHipAngularVelocity());  
  
 robot.setRHipTorque(torque);  
 }  
  
  
 public void LKneeController()  
 {  
 // ERROR term: Compute the difference between the desired position the pendulum and its current position  
 if(crouch.getIntegerValue() != 0)  
 {  
 if(iLCrouchKnee.getDoubleValue() < theNumberWeAreCountingUpTo)  
 {  
 positionError = (((desiredPositionRadians.getDoubleValue()\*((iLCrouchKnee.getDoubleValue()/theNumberWeAreCountingUpTo)))/2)) - robot.getLKneeAngularPosition();  
 iLCrouchKnee.add(1);  
 }  
 else  
 {  
 positionError = (desiredPositionRadians.getDoubleValue()\*0.5) - robot.getLKneeAngularPosition();  
  
 }  
 }  
 else  
 {  
 positionError = (desiredPositionRadians.getDoubleValue()\*0) - robot.getLKneeAngularPosition();  
 iLCrouchKnee.set(0);  
 }  
  
 // INTEGRAL term: Compute a simple numerical integration of the position error  
 integralError += positionError \* ArmSimulation.*DT*; //  
  
 // P.I.D  
 torque = p\_gain.getDoubleValue() \* positionError +  
 i\_gain.getDoubleValue() \* integralError +  
 d\_gain.getDoubleValue() \* (0 - robot.getLKneeAngularVelocity());  
  
 robot.setLKneeTorque(torque);  
 }  
 public void RKneeController()  
 {  
 // ERROR term: Compute the difference between the desired position the pendulum and its current position  
 if(crouch.getIntegerValue() != 0)  
 {  
 if(iRCrouchKnee.getDoubleValue() < theNumberWeAreCountingUpTo)  
 {  
 positionError = ((desiredPositionRadians.getDoubleValue()\*(((iRCrouchKnee.getDoubleValue()/theNumberWeAreCountingUpTo)))/2)) - robot.getLKneeAngularPosition();  
 iRCrouchKnee.add(1);  
 }  
 else  
 {  
 positionError = (desiredPositionRadians.getDoubleValue()\*0.5) - robot.getLKneeAngularPosition();  
  
 }  
 }  
 else  
 {  
 positionError = (desiredPositionRadians.getDoubleValue()\*0) - robot.getLKneeAngularPosition();  
 iRCrouchKnee.set(0);  
 }  
  
 // INTEGRAL term: Compute a simple numerical integration of the position error  
 integralError += positionError \* ArmSimulation.*DT*; //  
  
 // P.I.D  
 torque = p\_gain.getDoubleValue() \* positionError +  
 i\_gain.getDoubleValue() \* integralError +  
 d\_gain.getDoubleValue() \* (0 - robot.getRKneeAngularVelocity());  
  
 robot.setRKneeTorque(torque);  
 }  
  
  
 public void LAnkleController()  
 {  
 // ERROR term: Compute the difference between the desired position the pendulum and its current position  
 positionError = (0) - robot.getLAnkleAngularPosition();  
  
 // INTEGRAL term: Compute a simple numerical integration of the position error  
 integralError += positionError \* ArmSimulation.*DT*; //  
  
 // P.I.D  
 torque = p\_gain.getDoubleValue() \* positionError +  
 i\_gain.getDoubleValue() \* integralError +  
 d\_gain.getDoubleValue() \* (0 - robot.getLAnkleAngularVelocity());  
  
 robot.setLAnkleTorque(torque);  
 }  
 public void RAnkleController()  
 {  
 // ERROR term: Compute the difference between the desired position the pendulum and its current position  
 positionError = (0) - robot.getRAnkleAngularPosition();  
  
 // INTEGRAL term: Compute a simple numerical integration of the position error  
 integralError += positionError \* ArmSimulation.*DT*; //  
  
 // P.I.D  
 torque = p\_gain.getDoubleValue() \* positionError +  
 i\_gain.getDoubleValue() \* integralError +  
 d\_gain.getDoubleValue() \* (0 - robot.getRAnkleAngularVelocity());  
  
 robot.setRAnkleTorque(torque);  
 }  
  
 public YoVariableRegistry getYoVariableRegistry()  
 {  
 return registry;  
 }  
  
 public String getName()  
 {  
 return name;  
 }  
  
 public String getDescription()  
 {  
 return name;  
 }  
}

Here we have a controller method for each joint that holds JD steady.

Most of them simply hold him still, but a few have a little extra:

public void LRotatorController()  
{  
 // ERROR term: Compute the difference between the desired position the pendulum and its current position  
 if(LRotUp.getIntegerValue() != 0)  
 {  
 positionError = (0) - robot.getLRotatorAngularPosition();  
 }  
 else  
 {  
 positionError = (desiredPositionRadians.getDoubleValue()) - robot.getLRotatorAngularPosition();  
 }  
 // INTEGRAL term: Compute a simple numerical integration of the position error  
 integralError += positionError \* ArmSimulation.*DT*; //  
  
 // P.I.D  
 torque = p\_gain.getDoubleValue() \* positionError +  
 i\_gain.getDoubleValue() \* integralError +  
 d\_gain.getDoubleValue() \* (0 - robot.getLRotatorAngularVelocity());  
  
 robot.setLRotatorTorque(torque);  
}

This is the controller for the left shoulder rotator joint. Notice the if/else statement that changes with the variable LRotUp. This is a YoVariable. These can be edited through the UI while simulating. LRotUp changes LRotator and LElbow to make JD wave. While the program is running, look in the box in the bottom left of the UI. Type “LRotUp” into its search bar. You should see the variable appear with a value of 0. Start the simulation and change the variables value to 1.

Freaking.

Magic.

After watching the universe being born before your eyes change the value back to 0. Now type in “crouch”. When the variable appears change its value to 1 and run the simulation.

Oh my God.

The waving is accomplished almost identically to WOOWOO’s epic dancing. The crouch is slightly more complex.

public void LHipController()  
{  
 // ERROR term: Compute the difference between the desired position the pendulum and its current position  
 if(crouch.getIntegerValue() != 0)  
 {  
 if(iLCrouchHip.getDoubleValue() < theNumberWeAreCountingUpTo)  
 {  
 positionError = -(((desiredPositionRadians.getDoubleValue())\*(iLCrouchHip.getDoubleValue()/theNumberWeAreCountingUpTo))\*0.75) - robot.getLHipAngularPosition();  
 iLCrouchHip.add(1);  
 }  
 else  
 {  
 positionError = -(desiredPositionRadians.getDoubleValue()\*0.75) - robot.getLHipAngularPosition();  
 }  
 }  
 else  
 {  
 positionError = (desiredPositionRadians.getDoubleValue()\*0) - robot.getLHipAngularPosition();  
 iLCrouchHip.set(0);  
 }  
  
 // INTEGRAL term: Compute a simple numerical integration of the position error  
 integralError += positionError \* ArmSimulation.*DT*; //  
  
 // P.I.D  
 torque = p\_gain.getDoubleValue() \* positionError +  
 i\_gain.getDoubleValue() \* integralError +  
 d\_gain.getDoubleValue() \* (0 - robot.getLHipAngularVelocity());  
  
 robot.setLHipTorque(torque);  
}

We have a counter, iLCrouchHip and iRCrouchHip for the left and right sides, and we have a number that we are counting up to, theNumberWeAreCountingUpTo. theNumberWeAreCountingUpTo is set to 5000. We divide the counters by 5000, so they create a ratio growing closer to 1 as they are simulated. This allows JD to crouch slowly instead of all at once. He stands up all at once, though, and that is A-OK because he doesn’t fall over. So there you have it. A 100% capable Terminator.

If you liked this tutorial, we have some things you can try on your own:

-Make JD’s head real. Give it an actual joint and mass.

-Make JD stand up slowly, and put his arms out in front while crouching to balance better.

-Come up with a better way to create the Graphic methods, one that doesn’t involve one per joint.

-Make him WALK.